МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
им. М.В. Ломоносова

На правах рукописи

БУЛЫЧЁВ Пётр Евгеньевич

АЛГОРИТМЫ ВЫЧИСЛЕНИЯ ОТНОШЕНИЙ ПОДОБИЯ В ЗАДАЧАХ ВЕРИФИКАЦИИ И РЕСТРУКТУРИЗАЦИИ ПРОГРАММ

05.13.11 — математическое и программное обеспечение
вычислительных машин, комплексов и компьютерных сетей

ДИССЕРТАЦИЯ

на соискание ученой степени кандидата физико-математических наук

Научные руководители:
доктор физ.-мат. наук,
академик РАЕН,
профессор Р. Л. Смелянский;
кандидат физ.-мат. наук,
доцент В. А. Захаров.

Москва — 2010

Оглавление

Введение 6

1. Проблема обнаружения программных клонов 10
2. Отношения симуляции между конечными автоматами (структурами Крипке) 14
3. Отношения симуляций между временными автоматами 17

Глава 1. Обнаружение подобных фрагментов в исходных кодах программ 21

1.1. Постановка задачи 21

1.2. Обзор методов и средств автоматического обнаружения клонов 27

1.3. Основные определения 32

1.4. Алгоритм поиска клонов 37

1.5. Теоретическое сравнение возможностей разработанного и существующих алгоритмов 45

1.6. Описание практической реализации 48

1.7. Результаты экспериментальных исследований 50

1.8. Область применимости 59

1.9. Заключение 60

Глава 2. Проверка отношений подобия между размеченными системами переходов (структурами Крипке) 61

2.1. Постановка задачи 61

2.2. Обзор литературы 73

2.3. Универсальный язык задания симуляций 78

2.4. Универсальный символьный алгоритм проверки симуляций . . 90

2.5. Описание практической реализации 96
2.6. Применение средства проверки симуляции для инкрементального построения модели 97
2.7. Область применимости 101
2.8. Заключение 102

Глава 3. Проверка отношений подобия между временными автоматами 104
3.1. Постановка задачи 106
3.2. Обзор существующих подходов 115
3.3. Определение слабой альтернирующей временной симуляции . . 118
3.4. Алгоритм проверки слабой альтернирующей временной симуляции 127
3.5. Описание практической реализации 144
3.6. Результаты экспериментальных исследований 145
3.7. Область применимости 149

Заключение 150

Литература 153

Приложение А. Руководство по использованию разработанных программных средств 168
А.1. Запуск Clone Digger 168
А.2. Запуск универсальной среды проверки симуляции 169

Список иллюстраций

1.1 Фрагмент кода 22

1.2 Клон, удовлетворяющий определению, используемому в данной работе, и его процедурная абстракция 24

1.3 Пример абстрактного синтаксического дерева (АСД) 24

1.4 Обычное и линеаризированное представление АСД 25

1.5 Пример бесполезного клона, обнаруженного на лексическом уровне 30

1.6 Пример применения операции антиунификации 34

1.7 Алгоритм первого этапа кластеризации 39

1.8 Пример информации о клоне, обнаруженном средством Clone Digger 50

1.9 Сравнение производительности Clone Digger и CloneDR 52

1.10 Количество всех найденных клонов и клонов, классифицированных вручную 54

1.11 Сравнение полноты обнаружения клонов 58

1.12 Сравнение точности обнаружения клонов 58

2.1 $M_1 \preceq_{strong} M_2$ 69

2.2 Верхние оценки сложности вычисления строгой и прореженной симуляции и бисимуляции 77

2.3 Алгоритм решения игры 82

2.4 Пример игры для проверки прореженной симуляции 84

2.5 Алгоритм построения редуцированной игры 93

2.6 Алгоритм нахождения выигрышных состояний игрока Spoiler . 94

2.7 Результаты исследований на моделях универсального средства проверки симуляций, реализованного в рамках данной работы . 100

3.1 Операторы up, $down$, $reset$, $free$ 135
3.2 Проверка симуляции в средстве UPPAAL Tiga 144
3.3 Временные автоматы, на которых проводилось экспериментальное исследование разработанного алгоритма 147
3.4 Результаты экспериментального исследования разработанного алгоритма проверки twa-симуляции 147

Введение

Отношения подобия и различные их проявления играют большую роль как в математике, естествознании, так и в обыденной практике.

Словарь Даля определяет понятие подобия как схожесть, близость, принадлежность к одному виду, образу или обладание одинаковыми качествами. Похожее определение приводится и в других словарях: Ожегова, Оксфордском, Лопатникова и др.

Подобие лежит в основе аналогии, т. е. виде умозаключения, при котором знания, полученные в ходе рассмотрения какого-либо объекта A, переносятся на другой подобный ему объект B, менее изученный и в данный момент изучаемый. Помимо использования в аналогии отношения подобия обладают и другим полезным свойством: в некоторых случаях они позволяют не только установить, в чём схожи два объекта, но и понять, в чём они отличаются.

Аналогия служит источником научных гипотез, справедливость которых должна быть доказана теоретически или проверена экспериментально. Однако, если отношение подобия и объекты сравнения A и B определены формально, то на основании существования подобия между A и B и на основании известных свойств объекта A можно получить достоверные заключения и о свойствах изучаемого объекта B.

Отношения подобия позволяют вводить новые сущности, объединяя новые объекты в один класс. Таким образом возникают онтологии и появляется возможность описывать свойства не только отдельных объектов, но и свойства целых классов объектов.

В данной работе исследуется применение отношений подобия в задачах автоматического анализа программ. Среди методов анализа программ можно выделить методы, работающие с исходными текстами программ, и методы,

в которых проверяются динамические свойства (поведение) программ. Мы будем рассматривать применение отношений подобия к анализу программ в обоих этих ракурсах. Таким образом, мы будем исследовать схожесть в устройстве программ (синтаксическое подобие) и схожесть их поведения (семантическое подобие).

Во-первых, мы рассмотрим проблему обнаружения синтаксически похожих фрагментов (клонов, дубликатов кода) в исходных кодах программ. Результаты статистических исследований, о которых говорится в работах [8, 11, 83], свидетельствуют о том, что суммарный объём клонов в больших проектах обычно составляет 7–20%, а в некоторых случаях доходит и до 59% [33]. Наличие большого числа клонов приводит к увеличению стоимости поддержки кода [73, 96], к возрастанию ресурсов, необходимых для компиляции и хранения скомпилированной программы [59, 93]. Кроме того, сознательное дублирование кода (т. е. использование операции copy&paste) приводит к увеличению вероятности возникновения новых и распространения существующих ошибок [26, 31, 60]. Поэтому от избыточного дублирования кода целесообразно избавляться при помощи существующих методов реорганизации (рефакторинга) кода, например, при помощи процедурной абстракции, также называемой “Выделением метода” (Extract Method) в терминологии Фаулера [121]. В данной работе ставится задача разработи и реализации алгоритма нахождения клонов, от которых легко можно избавиться при помощи таких методов рефакторинга.

Не менее важное практическое значение имеют и отношения семантического подобия, проявляющиеся в схожести функциональных свойств вычислительных систем. Во второй части рассмотрены симуляции — отношения подобия специального вида, возникающие в задачах верификации программ. Отношения симуляции — это бинарные отношения, заданные на множестве

моделей вычисления (автоматов, сетей Петри, программ), которые проявляются в том, что структура дерева вычислений одной модели оказывается сходной структуре дерева вычислений другой модели. Интересны эти отношения тем, что они сохраняют выполнимость спецификаций, заданных формулами темпоральных логик, т. е. если модели A и B находятся в отношении симуляции (обозначается $B \preceq A$), то любая спецификация, выполнимая для A, выполняется и для B. Благодаря этому качеству отношения симуляции часто используются в методах формальной верификации программ, позволяя сводить проверку программ к анализу их абстрактных моделей, сохраняющих исследуемые свойства программ.

Начиная с основополагающей статьи Миллнера [74] отношения симуляции хорошо исследованы для случая, когда модели задаются структурами Крипке. По мере возникновения новых задач и методов верификации моделей программ появляются новые отношения. Библиография статей, посвящённых отношениям симуляции, вероятно, обширна. Наиболее распространённые отношения симуляции и способы их проверки описаны в работах [15, 17, 35, 43, 47, 50, 54, 65, 89, 95, 105]. Более того, постоянно появляются новые отношения, поэтому существует необходимость в создании универсальной системы проверки симуляции, которая могла бы быть использована как единая платформа для проверки различных отношений, в зависимости от потребностей пользователя. С научной точки зрения разработка данного средства интересно тем, что оно позволит дать ответ на вопрос о том, можно ли сконструировать универсальный алгоритм проверки симуляций, не уступающий по эффективности существующим индивидуальным алгоритмам. С практической точки зрения это средство будет полезно тем, что оно может быть приспособлено к разным системам верификации и для проверки различных (в т. ч. новых) отношений симуляции. В данной работе ставится задача

разработки такой системы.

Помимо конечных автоматов в верификации программ на моделях используются и более выразительные формализмы описания программ, в частности, временные автоматы [5]. По существу, временные автоматы — это конечные автоматы, снабжённые конечным множеством таймеров и ограничениями на значения таймеров, при которых допустимы те или иные переходы. Пока автомат находится в одном состоянии, значения всех таймеров растут с одинаковой скоростью, при переходе из состояния в состояние значения некоторых таймеров могут сбрасываться в 0. Временные автоматы могут использоваться в задаче синтеза контроллера (устройства управления). В этом случае автомат, фактически, описывает взаимодействие двух игроков: контроллера и среды. Сложность временных автоматов настолько велика, что практическое применение этих методов невозможно без применения абстракции. Поэтому разработка подходящего отношения симуляции и алгоритма его проверки — ключевая задача в разработке любой инструментальной системы верификации временных автоматов. Однако, отношения симуляции между временными автоматами не так хорошо исследованы, как отношения симуляции между конечными автоматами. В данной работе ставится задача разработки формального определения временной симуляции между игровыми времеными автоматами и построение алгоритма проверки этой симуляции.

Таким образом, в данной работе рассматриваются три задачи, связанные с исследованием подобия, возникающим в анализе программ:

1. задача обнаружения клонов, в которой задействована идея синтаксического подобия,

2. задача построения универсального алгоритма проверки отношений симуляции (семантического подобия) между моделями дискретных про-

грамм,

3. задача разработки отношения симуляции для временных игровых автоматов и построения алгоритма проверки этого отношения.

Опишем каждую из трёх поставленных задач более подробно.

1. Проблема обнаружения программных клонов

Нередко в процессе разработки программного обеспечения оказывается, что функциональные возможности фрагмента, который необходимо реализовать программисту, похожи на те, что были реализованы ранее в другом фрагменте программы. В этом случае разработчик может использовать операцию "копирование и вставка" (copy&paste), т. е. скопировать существующий фрагмент кода в новое место. Применение данной операции приводит к появлению больших и похожих фрагментов кода, также называемыми клонами или дубликатами кода. В обзоре [93] проведён анализ этого явления на многочисленных примерах из практики. Частое использование операции "копирование и вставка" может говорить о низкой культуре программирования, плохом знании создаваемого программного продукта и стремлении увеличить скорость разработки или обьём создаваемого кода в ущерб его качеству. К сожалению, эти явления неизбежно сопутствуют разработке всякого программного проекта в том случае, когда критическим фактором является время создания программного продукта. В большинстве случаев вместо операции "копирование и вставка" следует использовать более глубокую реорганизацию кода, которая заключается в перемещении общего кода в новую функцию, или создании абстрактного класса [121]. Однако существуют ситуации, когда применение операции "копирование и вставка" является оправданным. Например, как известно, в настоящее время при разработке критических программных систем

(встроенных, финансовых и др.) стоимость их проверки (тестирования, формальной верификации) существенно больше стоимости разработки [29]. Для таких систем после повторного использования работоспособного кода требуется потратить меньше ресурсов на проверку всей системы, чем после глубокой реорганизации кода. Например, модификация и повторное использование работоспособного кода требует меньших затрат на проверку всей системы, чем глубокая реорганизация кода. Кроме того, программисты могут сознательно дублировать существующий код, зная, что в дальнейшем две ветви будут развиваться независимо друг от друга [63].

Похожие фрагменты кода могут появляться и в результате непреднамеренного дублирования. Два разных программиста могут создать похожие фрагменты при реализации одного и того же алгоритма. Более того, такого рода фрагменты могут порождаться и одним программистом, поскольку разработчики программного обеспечения часто помнят шаблоны решения типовых задач, и применяют их в похожих ситуациях. Другой причиной непреднамеренного дублирования кода может являться использование прикладного программного интерфейса (application programming interface, API) некоторой библиотеки, которое приводит к появлению похожих последовательностей вызовов библиотечных функций [63].

Из-за действия описанных выше факторов практически во всех больших программных системах существуют похожие фрагменты, которые называются клонами. В отличие от многих других широко используемых в программировании понятий (например, процедуры, алгоритма) в настоящее время не существует строгого математического определения понятия клона. Но все исследователи сходятся в том, что клоном называются фрагменты, имеющие незначительные синтаксические отличия друг от друга. Однако на счёт того, какие именно синтаксические отличия можно признать незначительными, ис-

следователи пока не имеют общего мнения. Это и не удивительно, поскольку эта область исследований находится в самом начале своего развития. Однако даже в случае использования наиболее требовательных определений суммарный объём может составлять до 20% от исходных кодов программ [93].

Наличие большого числа клонов приводит к следующим проблемам:

- трудности модификации программы [59, 73],
- клонирование способствует распространению ошибок [31, 60],
- повышение требований к оборудованию, на котором будет компилироваться и исполняться данный код [59, 93].

Рассмотрим эти проблемы подробнее.

Если в программе имеются несколько похожих фрагментов, то не исключено, что они реализуют одну и ту же функциональность и при внесении существенных изменений в один из них потребуется изменить и другие. Если же программист забудет это сделать, то тем самым он может привести код в рассогласованное состояние. Таким образом, для корректного изменения содержащего клоны кода необходимо вручную анализировать все фрагменты, которые похожи на изменяемый участок. Это, в свою очередь, приводит к увеличению временных затрат на поддержку кода.

При клонировании фрагментов, уже содержащих ошибки, эти ошибки присутствуют и в вставленных фрагментах. В частности, в работе [42] проводится анализ причин появления ошибок в исходных кодах операционной системы Linux. Установлено, что в одном-единственном файле 34 из 35 ошибок были результатом преднамеренного дублирования кода. Одна ошибка была скопирована в 10 мест, а другая — в 24.

Присутствие клонов в программе также приводит к увеличению размера исполняемого файла, что может являться негативным фактором при исполь-

зовании данной программы во встроенных системах [28]. Кроме того, требуется дополнительное время на компиляцию исходных текстов программы.

Таким образом, наличие клонов в программах приводит к увеличению стоимости их поддержки. Следовательно, детекторы клонов могут использоваться для нахождения клонов, которые в дальнейшем будут устранены. Однако, это не единственная область применения такого рода средств.

Детекторы клонов могут использоваться для выделения библиотек часто используемых функций [19]. Кроме того, найденные клоны могут использоваться для нахождения шаблонов использования классов, и таким образом, для добавления новых методов в эти классы. При добавлении новой функций можно проверять с помощью детектора клонов, не существует ли уже функции, которая имеет ту же функциональность [8]. Детекторы клонов могут помочь обнаружить вирусы, добавляющие свой код в программы, написанные на интерпретируемых языках программирования (таких как JavaScript) [108]. Относительное количество кода, покрытого клонами, может использоваться в качестве одной из метрик качества кода [96]. Ещё одной областью применимости детекторов клонов является выявление нарушения авторских прав [62, 108].

Таким образом, автоматические детекторы клонов являются важным классом средств анализа программного обеспечения.

К сожалению, специалисты в области анализа программного обеспечения ещё не выработали общепринятого формального определения понятия клона. Основной причиной этого, на наш взгляд, является то, что эта область исследований до сих пор активно развивается. Однако, поскольку в большинстве случаев автоматические детекторы клонов используются в процессе улучшения качества исходного кода путём реорганизации кода (рефакторинга), то разумно предположить, что средство обнаружения клонов, которые могут

быть устранены посредством рефакторинга, будет востребовано разработчиками программного обеспечения. С одной стороны подобное средство должно находить как можно больше таких клонов, а с другой стороны, для удобства использования оно должно иметь как можно меньшее число ложных срабатываний.

Любое средство обнаружения клонов в своей работе должно опираться на некоторое формальное определение клона. Используемое нами определение было получено на основе анализа того, какого рода отличия могут допускаться между фрагментами, составляющими клон, от которого можно избавиться при помощи рефакторинга. Такое определение не было рассмотрено ранее в литературе и, вместе с тем, оно обобщает существующие определения. Задача, которая ставится в данной диссертационной работе, заключается в разработке и реализации алгоритма нахождения клонов, удовлетворяющих этому определению.

2. Отношения симуляции между конечными автоматами (структурами Крипке)

Проверка правильности программ — это одна из наиболее трудных задач программирования. Широко используемое на практике тестирование программного обеспечения не позволяет установить корректность функционирования анализируемой программы, а лишь помогает выявить наличие в ней ошибок.

Поэтому для проверки правильности программных систем, цена ошибки в которых велика (жизненно-важное ПО, встроенные системы), целесообразно использовать формальные методы верификации, позволяющие получить строгие математические доказательства отсутствия дефектов в поведении

программ.

Одно из наиболее активно развивающихся направлений в области формальных методов проверки правильности программ — это верификация моделей программ [118]. Суть этого подхода состоит в том, что проводится проверка не самой программы, а её математической модели. Проверяемые спецификации обычно задаются формулами темпоральных логик. Важным этапом метода является построение модели. Модели могут отражать поведение программы с разной степенью подробности, они могут строиться как вручную, так и автоматически [30]. Чрезмерно упрощённая модель может давать слишком грубое приближение реальной программы, и на ней не будут проявляться те свойства вычислений, которые присущи реальной программе. С другой стороны, при верификации чрезмерно подробных моделей приходится преодолевать эффект “комбинаторного взрыва” числа состояний, из-за которого анализ модели становится практически неосуществимым. Поэтому для применения метода верификации моделей удобно иметь средство, позволяющее сравнивать устройство моделей программ между собой. Наиболее важными отношениями между моделями программ являются отношения симуляции, которые проявляются в том, что устройство одной модели оказывается подобным устройству другой. Такое свойство приводит к тому, что выполнимость спецификации на одной модели неизбежно влечёт их выполнимость и на другой. Отношения подобия (симуляции) находят широкое применение при проверке правильности поведения сложных программ.

Понятие отношения симуляции между моделями программ было впервые введено Миллнером (Millner) в 1971 году в работе [74]. Миллнер рассматривал симуляции только между детерменированными моделями программ и предлагал использовать эти отношения как формализм, позволящий проверять эквивалентность программ на некотором уровне абстракции. Позднее, в

1980 году Миллнер обобщил это понятие симуляционной эквивалентности на случай недетерменированных программ и ввёл понятия строгой и слабой бисимуляций [75]. В 1981 году Парк (Park) обобщил понятие симуляции на случай недетерменированных моделей программ, задаваемыми автоматами Бюхи [81]. Важным свойством отношения симуляции, по словам Парка, является то, что с одной стороны отношение симуляции является более сильным, чем широко используемое отношение трассового (языкового) включения [104], а с другой стороны, проверка выполнимости отношения симуляции является алгоритмически более простой задачей, чем задача проверки трассового включения. В дальнейшем в литературе было приведено множество примеров использования отношений симуляций. В частности, отношения симуляции использовались в процессе инкрементального построения моделей, при котором каждая новая версия модели является уточнением предыдущей [21]. Кроме того, отношения симуляции могут использоваться для вычисления инвариантов бесконечных параметризованных семейств моделей [65], для нахождения симметрии в моделях [106], для автоматического уменьшения размеров моделей [20, 111], для проверки свойств конфиденциальности многопоточных программ [94].

Разнообразие отношений симуляции велико; разные отношения сохраняют выполнимость формул разных темпоральных логик и используются в различных случаях. Кроме того, возможно, в будущем у исследователей в области верификации моделей программ возникнет необходимость в проверке новых отношений симуляции. Поэтому хотелось бы иметь универсальную систему проверки симуляции, которая могла бы быть использована для проверки различных отношений, в зависимости от потребностей пользователя. В данной диссертационной работе ставится задача разработки такой системы. Универсальная система проверки симуляций должна состоять из двух компо-

нент: формального языка, позволяющего описывать различные симуляции, и программно-инструментального средства, которое по двум размеченным системам переходов и описанию симуляции проверяет, выполняется ли заданная симуляция между этими системами переходов.

В качестве формализма задания моделей нами были выбраны структуры Крипке (которые, фактически, являются конечными автоматами) по двум причинам. Во-первых, структуры Крипке используются во многих популярных средствах проверки моделей программ, в частности, в средствах NuSMV [77] и Spin [57]. С другой стороны, отношения симуляции между структурами Крипке достаточно хорошо исследованы, и для всех известных нам отношений симуляций на множестве структур Крипке существуют алгоритмы их проверки.

3. Отношения симуляций между временными автоматами

Несмотря на то, что конечные автоматы (структуры Крипке) были использованы во многих задачах верификации моделей программ, в некоторых случаях этот формализм оказывается недостаточно выразительным для адекватного описания анализируемых программ. В таких случаях используются различные расширения класса конечных автоматов, например, гибридные автоматы.

Гибридные автоматы [1] являются удобным формализмом для описания так называемых гибридных систем, т. е. систем, в которых дискретные управляющие устройства взаимодействуют с непрерывным физическим окружением. Гибридный автомат состоит из системы переходов, каждое состояние которой снабжено системой дифференциальных уравнений над некоторым множе-

ством вещественных переменных. Вычисление гибридного автомата состоит из последовательности дискретных переходов, перемежаемой обновлениями значений вещественных переменных в соответствии с дифференциальными уравнениями.

Гибридные автоматы часто используются при имитационном моделировании, однако для формального анализа в общем виде они слишком сложны. В частности, даже задача проверки достижимости заданного состояния в гибридном автомате является алгоритмически неразрешимой [55].

Исследователи уделили много внимания поиску частных случаев гибридных автоматов, которые с одной стороны были бы достаточно выразительными для описания управляющих систем, функционирующих в реальном времени, а с другой стороны задачи их анализа были бы алгоритмически разрешимыми. Одним из таких частных случаев являются временные автоматы. Во временных автоматах системы дифференциальных уравнений, описывающих изменения переменных, имеют вид $\dot{\overline{X}} = (1, 1, \ldots, 1)$, т. е. все переменные растут с одинаковой скоростью. Таким образом, формализм временных автоматов является достаточно выразительным для описания систем, в которых единственным непрерывно изменяемым параметром являются часы. Для временных автоматов является разрешимой задача проверки достижимости заданного состояния [5], более того, для таких автоматов разрешима проблема проверки выполнимости формул темпоральной логики Timed CTL(TCTL) [4].

Временные автоматы также используются в качестве удобной математической модели при решении задачи синтеза систем автоматического управления исходя из спецификации требуемого поведения устройства управления и возможного поведения среды [71]. Эта задача синтеза состоит в построении такого алгоритма работы устройства управления (контроллера), что объект управления будет корректно функционировать в независимости от действий

среды. Например, автопилот должен вести самолёт по курсу вне зависимости от внешних условий. При решении этой задачи система автоматического управления может быть представлена автоматом, множество переходов которого состоит из переходов двух типов, контролируемых и неконтролируемых устройством управления. Такой автомат фактически представляет собой описание игры двух участников — контроллера и внешней среды. И действительно, в работе [71] было показано, что задача синтеза работы контроллера сводится к задаче нахождения его выигрышной стратегии в этой игре. В этой же работе было установлено, что задача синтеза разрешима для случая, когда цель контроллера — перевести систему за определённое время из начального состояния в заданное состояние, избегая при этом попадания в некоторые выделенные (аварийные) состояния (такие свойства можно задавать формулами логики ATCTL). Позже в работе [41] был предложен алгоритм решения этой задачи "на лету" (on-the-fly). Этот алгоритм был реализован в рамках среды UPPAAL Tiga [103], которая была использована в нескольких промышленных приложениях, в частности, компанией Skov A/S для автоматического синтеза контроллера для поддержания стабильной температуры в современных свино- и птицефермах [51]; и для автоматического синтеза контроллеров, управляющих автономным поведением роботов [84] (в обоих случаях осуществлялся контроль десятками параметров).

Отношения симуляции между временными автоматами не так хорошо исследованы, как отношения симуляции между конечными автоматами. Главной причиной этого является то, что временные автоматы имеют бесконечное число состояний (поскольку каждое состояние включает в себя значения вещественных переменных-часов), и поэтому разработка алгоритмов проверки временных симуляций является сложной задачей. В то же время, помимо перечисленных выше примеров применения симуляций между конечными

автоматами существуют и специфические примеры применения отношений симуляции в задачах анализа временных игровых автоматов. В частности, отношения временных симуляций могут использоваться в случае, когда состояние полной модели S системы автоматического управления включает в себя информацию о состоянии среды и/или объекта управления, которая не должна использоваться в алгоритме работы контроллера. Например, оно может включать в себя информацию о местоположении робота в пространстве, однако контроллеру робота должны быть доступны только показания датчиков, но не его точное местоположение. Чтобы решить задачу синтеза алгоритма работы контроллера в такой системе, можно разработать модель A, в которую не будет входить информация о скрытом состоянии среды и объекта управления. Затем необходимо построить выигрышную стратегию контроллера для модели A и проверить существование симуляции между A и S. Если эта симуляция выполняется, то из подобия моделей A и S будет следовать, что построенная выигрышная стратегия контроллера для A будет выигрышной и для S. Поскольку в данной стратегии используется только доступная контроллеру информация о состоянии среды и объекта управления, то эта стратегия может лежать в основе алгоритма работы контроллера.

В данной диссертационной работе ставится задача разработки определения отношения симуляции между временными игровыми автоматами, которое бы сохраняло выполнимость формул логики ATCTL, и разработки алгоритма проверки выполнимости этого отношения.

Глава 1

Обнаружение подобных фрагментов в исходных кодах программ

1.1. Постановка задачи

Из-за использования программистами операции copy&paste и других факторов в программах появляются большие похожие фрагменты кода, которые принято называть клонами. В литературе достаточно хорошо исследован феномен программных клонов (см., например, обзор [93]) и показано, что наличие большого числа клонов в программе приводит к увеличению стоимости её поддержки и к другим негативным факторам. К сожалению, в отличие от многих других широко используемых в программировании понятий (например, процедуры, алгоритма) в настоящее время не существует строгого математического определения понятия клона. Но все исследователи сходятся в том, что клоном называются фрагменты, имеющие незначительные синтаксические отличия друг от друга.

В данной работе ставится задача обнаружения клонов, от которых можно легко избавиться при помощи существующих методов реорганизации (рефакторинга) кода. Проанализируем методы рефакторинга и дадим определение, которому должны удовлетворять такие клоны.

Как указывается в обзоре [93], клоны могут быть удалены с использованием операций “Выделение метода” (Extract Method) и “Подъём метода” (Pull Up Method), изложенных в монографии Мартина Фаулера “Рефакторинг: улучшение существующего кода” [121]. “Выделение метода” (в русскоязычной литературе называемое процедурной абстракцией) заключается в

```
s1: int fib(int n) {
s2:    int n0 = 0, n1 = 1, naux, i;
s3:    if (n == 0)
s4:       return 0;
s5:    for (i=0; i < n-1; i++) {
s6:       naux = n1;
s7:       n1 = n0 + n1;
s8:       n0 = naux; }
s9:    return n1; }
```

Рисунок 1.1. Фрагмент кода

выделении общего кода в новую функцию (или метод класса, если речь идёт об обьектно-ориентированном программировании). “Подъём метода” заключается в перемещении общего метода двух классов в их базовый класс. Если два класса не являются потомками одного базового класса, то перед операцией “Подъём метода” можно создать новый базовый класс при помощи операции “Выделение родительского класса” (Extract Superclass).

Опишем вид фрагментов кода, входящих в клон, который может быть легко удалён при помощи описанных методов рефакторинга. Понятно, что минимальным фрагментом кода, который может быть безболезненно выделен в отдельную функцию, является последовательность операторов (возможно, состоящая из одного элемента). Из-за существования составных (и вложенных) операторов понятие последовательности операторов может быть истолковано неоднозначно, поэтому мы будем считать, что в клоны могут входить только *линейные* последовательности операторов. Рассмотрим фрагмент кода, изображённый на рисунке 1.1. В соответствии с нашим определением, *линейная* последовательность операторов (s2, s3, s5) может входить в клон, а последо-

вательность (s4, s5) — нет, поскольку оператор s4 относится к блоку `if`, а оператор s5 находится на самом верхнем уровне тела функции.

Рассмотрим теперь, какого рода отличия между фрагментами, составляющими клон, могут допускаться для того, чтобы можно было легко произвести их рефакторинг. Для успешного применения операций “Подъём метода” и “Выделение родительского класса” необходимо, чтобы фрагменты, составляющие клон, были семантически эквивалентны. Проверка семантической эквивалентности является неразрешимой задачей, поэтому на практике можно потребовать, чтобы эти фрагменты были синтаксически эквивалентны, т. е. один фрагмент может быть получен из другого переименованием локальных переменных, добавлением/удалением комментариев и незначащих пробелов.

Для примененимости операции “Выделение метода” мы можем допустить более широкие отличия одного фрагмента клона от другого, например, замену одной константы на другую или замену одного подвыражения на другое подвыражение. И подвыражения, и константы являются частными случаями синтаксических единиц, т. е таких фрагментов кода, к которым может быть применено синтаксическое правило языка программирования. Например, фрагменты “`a+b`” и “`a`” являются синтаксическими единицами, а фрагмент “`a+`” не является таковым. Легко видеть, что наиболее простыми для рефакторинга являются клоны, один фрагмент которых может быть получен из другого фрагмента путём замены одних синтаксических единиц на любые другие синтаксические единицы.

На рисунке 1.2 приведён пример такого клона и результат применения к нему операции “Выделение метода”. Видно, что первый фрагмент может быть получен из второго фрагмента заменой всех выражений вида `b>>8` на `OPC`. Это позволило провести процедурную абстракцию (применить “Выделение метода”) и заменить эти два фрагмента на вызов функции `f`, имеющей один

```
try  {                  try  {                  void f(int arg) {
  p++;                    p++;                    try  {
  s[o++] = OPC;           s[o++] = b >> 8;          p+;
}  catch  (Exc e) {   }  catch  (Exc e) {          s[o++] = arg;
  resizeArray(OPC);       resizeArray(b>>8);    }  catch  (Exc e) {
}                       }                         resizeArray(arg);
                                                  }
                                                }
```

Рисунок 1.2. Клон, удовлетворяющий определению, используемому в данной работе, и его процедурная абстракция

```
x = a + b;
y = f(x,j);
cout << y;
```

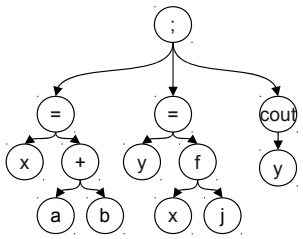

Рисунок 1.3. Пример абстрактного синтаксического дерева (АСД)

аргумент и приведённой на том же рисунке.

Таким образом, мы приходим к следующему неформальному определению клонов, удовлетворяющих указанным выше критериям. Два фрагмента кода называются клонами, если эти фрагменты являются линейными последовательностями операторов и один из фрагментов может быть получен из другого путём замены некоторых синтаксических единиц на любые другие синтаксические единицы.

Чтобы дать формальное определение клонов нам потребуется ввести по-

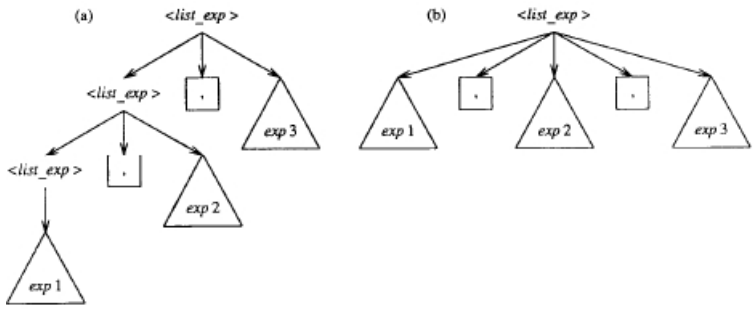

Рисунок 1.4. Обычное и линеаризированное представление АСД

нятие абстрактного синтаксического дерева. Абстрактное синтаксическое дерево (abstract syntax tree, АСД) — это конечное, размеченное, ориентированное дерево, в котором внутренние вершины помечены операторами языка программирования, а листья — соответствующими операндами. АСД подробно исследованы Ахо (Aho), Сети (Sethi), Ульманом (Ulman) в книге “Компиляторы. Принципы, технологии, инструменты” [114]. АСД широко применяются для синтаксического анализа при разработке компиляторов. Не всякая программа может быть представлена в виде АСД, но те не менее, большие фрагмента кода (тела процедур, функции), линейные участки допускают построение их АСД.

Легко видеть, что синтаксические единицы исходного кода программы соответствуют поддеревьям АСД этой программы. На рисунке 1.3 приведён пример абстрактного синтаксического дерева.

Для представления АСД, порождаемых леворекурсивными правилами вывода, такими, как `<list-exp> ::= <list-exp> "," <exp> | <exp>`, мы используем так называемое линеаризированное представление [113]. В этом представлении все вершины, соответствующие правилам `<exp>` являются непосредственными потомками вершины, соответствующей `<list-exp>`. На рисунке 1.4 приведены схематические изображения обычного и линиеаризи-

рованного АСД.

Мы описали, какими могут быть фрагменты кода, составляющие клон, и какие между ними допускаются отличия. Понятно, что имеет смысл производить поиск только клонов, состоящих из достаточно больших и достаточно похожих фрагментов. Чтобы формализовать это, будем считать, что задана функция $|\cdot|$, сопоставляющая произвольному фрагменту E некоторое число $|E|$, называемое его размером; и функция ρ, сопоставляющая паре фрагментов E_1 и E_2 число $\rho(E_1, E_2)$, называемое расстоянием между E_1 и E_2.

В результате мы приходим к следующему определению клона, от которого можно легко избавиться при помощи рефакторинга.

Определение 1.1 (Клон кода)**.** Две последовательности операторов E_1 и E_2 формируют (p_1, p_2)-клон в метрике $(\rho, |\cdot|)$, если абстрактное синтаксическое дерево последовательности E_1 можно получить из абстрактного синтаксического дерева последовательности E_2 путём замены некоторых поддеревьев на другие поддеревья, и справедливы неравенства $\min(|E_1|, |E_2|) \geq p_1$ и $\rho(E_1, E_2) \leq p_2$.

Определение 1.1 является параметризованным и зависит от того, какие выбраны функции ρ и $|\cdot|$. В данной работе ставится задача выбора подходящих функций $|\cdot|$ и ρ.

Кроме того, в данной работе ставится задача разработки и реализации алгоритма, который бы позволял обнаруживать клоны, удовлетворяющие определению 1.1.

1.2. Обзор методов и средств автоматического обнаружения клонов

1.2.1. Цель обзора

В данном разделе проводится обзор существующих методов и средств автоматического обнаружения клонов, основанный, в первую очередь, на работе [93]. Сравнительный анализ определений клонов и алгоритмов из обнаружения позволит нам:

1. выявить место, которое занимает определение 1.1 в существующей таксономии клонов и

2. выяснить, существует ли подходящий алгоритм среди извесных ранее для обнаружения клонов, удовлетворяющих определению 1.1.

1.2.2. Общепризнанная классификация определений клонов

Определения клонов кода классифицируются исходя из представления исходного текста программы, с которым работает алгоритм их обнаружения. В обзоре [93] и в большинстве других работ выделяются следующие типы представления исходного текста:

- последовательность строк (текстовое представление) [10, 38, 60, 72, 110],

- последовательность лексем [10, 31, 62],

- абстрактное синтаксическое дерево [26, 45, 113],

- набор числовых характеристик кода [70, 73, 83],

- граф потока управления [48, 64, 67].

Кроме перечисленных типов представления некоторые исследователи используют их комбинацию.

Поскольку в определении 1.1 используется понятие абстрактного синтаксического дерева, то разумно предположить, что и алгоритм поиска клонов, удовлетворяющих этому определению, будет работать на уровне абстрактных синтаксических деревьев. Однако в обзоре мы вкратце остановимся и на других подходах.

1.2.3. Текстовые клоны

Детекторы клонов, работающие на уровне текстового представления, производят поиск посимвольно совпадающих участков кода [10, 38, 60, 72, 110]. По запросу пользователя может проводится предварительная нормализация исходного текста, в частности удаление комментариев и незначащих пробелов. Такая нормализация позволяет уменьшить число ложных срабатываний, однако для неё необходимо наличие лексического анализатора, что сужает область применимости детектора клонов.

Особняком стоит работа [72], в которой клоны ищутся исходя из анализа текста комментариев и имён переменных. В этом подходе во время нормализации необходимо избавляться от служебных слов.

В случае, когда предварительной нормализации не производится, детекторы текстовых клонов являются универсальными, то есть могут использоваться для нахождения клонов в программе, написанной на произвольном языке программирования (и даже тексте, написанном на естественном языке).

1.2.4. Клоны на уровне последовательности лексем

В этом подходе исходный код программы преобразуется в последовательность лексем, в которой затем ищутся одинаковые подпоследовательности [10, 31, 62].

В средстве CCFinder [62] все лексемы, соответствующие именам переменных, классов и функций заменяются на специальные лексемы-заполнители. Это позволяет находить клоны, отличающиеся в идентификаторах.

Средство Dup[10] может находить так называемые параметризованные клоны, в которых несколько вхождений одной переменной были заменены на другую переменную.

В работе [31] приводится алгоритм, который позволяет находить клоны, в которых один фрагмент получается из другого фрагмента вставкой/удалением небольшого числа лексем (максимально допустимое число лексем является параметром алгоритма). Этот алгоритм реализован в автоматическом детекторе клонов CPMiner.

Использование двух описанных выше видов представления исходных кодов не позволяют находить клоны, удовлетворяющие определению 1.1, поскольку на этих уровнях невозможно выделить операторы и синтаксические единицы.

1.2.5. Клоны на уровне абстрактных синтаксических деревьев

Абстрактное синтаксическое дерево (abstract syntax tree, АСД) — это конечное, размеченное, ориентированное дерево, в котором внутренние вершины помечены операторами языка программирования, а листья — соответствующими операндами. Синтаксические деревья используются в парсерах для промежуточного представления программы между деревом разбора (конкрет-

```
  return result;   return x; }
}                int bar() { int y;
int foo(){
  int a;
```

Рисунок 1.5. Пример бесполезного клона, обнаруженного на лексическом уровне

ным синтаксическим деревом) и структурой данных которая затем используется в качестве внутреннего представления компилятора или интерпретатора компьютерной программы для оптимизации и генерации кода [114].

Представление исходного кода в виде абстрактного синтаксического дерева сохраняет всю информацию о синтаксисе программы. Используя эту информацию, можно проигнорировать клон, изображенный на рисунке 1.5, который занимает конец одной функции и начало следующей функции.

В работе [113] описывается первый алгоритм нахождения отличий на уровне АСД между двумя программами. Утилита cdiff, разработанная авторами данной статьи, работает в соответствии с принципами динамического программирования и подсвечивает отличия между фрагментами кода на уровне АСД.

В средстве CloneDR [26] используется подход к обнаружению клонов, в котором для каждого поддерева АСД программы вычисляется заданная хеш-функция, а затем происходит поиск последовательностей одинаково помеченных поддеревьев. Данная хеш-функция была выбрана так, что при её вычислении игнорируются имена переменных, функций, константы. Похожий подход используется и в разработанном позже средстве ccdiml[90]. К сожалению, сложно придумать хеш-функцию, которая была бы инвариантна от-

носительно замены *произвольных* подвыражений на другие подвыражения, таким образом, метод, описанный в работе [26], нельзя использовать для нахождения клонов, удовлетворяющих определению 1.1.

В работе [45] описывается алгоритм нахождения так называемых структурных клонов, которые могут отличаться в *произвольных* поддеревьях их АСД. Данный алгоритм реализован в средстве Asta. Из описания инструментального средства Asta легко видеть, что с его помощью могут быть найдены клоны, удовлетворяющиеы определению 1.1, однако используемый алгоритм позволяет находить только клоны, в которые входят отдельно стоящие операторы, но не последовательности операторов. Таким образом, средство Asta может найти похожие функции, но не фрагменты функций, состоящие из последовательностей операторов.

1.2.6. Клоны на уровне числовых характеристик кода

В этой группе алгоритмов для каждого фрагмента (функции, класса, файла) программы вычисляется векторы значений заданных числовых характеристик (метрики). Затем происходит поиск фрагментов, имеющих похожие или совпадающие метрики. В качестве метрик могут использоваться количества строк кода функций или количество вызовов других функций. В большинстве случаев для вычисления значений метрик необходимо сначала построить абстрактное синтаксическое дерево программы.

Данный подход рассмотрен в работах [70, 73, 83].

Очевидно, что на уровне числовых характеристик кода нельзя искать клоны, удовлетворяющие определению 1.1.

1.2.7. Клоны на уровне графа зависимостей программы

Граф зависимостей программы (program dependence graph, ГЗП) [46] — это абстрактное представление программы, в котором хранится информация о зависимостях по данным и по управлению. В русскоязычной литературе ГЗП называют также информационно-логическим графом [117].

При поиске клонов на уровне ГЗП появляется возможность учитывать семантическую информацию. В методах обнаружения клонов на уровне ГЗП происходит поиск изоморфных подграфов в ГЗП. Эта задача является NP-полной, и поэтому на практике используются эвристические алгоритмы. Подход, основанный на использовании ГЗП, рассмотрен в работах [48, 64, 67].

1.2.8. Выводы из обзора

Таким образом, наше определение клонов 1.1 в наибольшей степени похоже на определения, данные в работах [45] и [26]. В работе [45] даётся алгоритм обнаружения клонов, которые могут отличаться в произвольных подвыражениях, но сами фрагменты, составляющие клон, являются отдельно стоящими операторами. Алгоритм, описанный в работе [26], позволяет находить клоны, состоящие из последовательностей операторов, однако эти фрагменты могут отличаться только в именах и значениях констант. Поэтому ни в одной из рассмотренных работ не описан подход, позволяющий обнаруживать клоны, удовлетворяющие определению 1.1.

1.3. Основные определения

В данном разделе приводятся определения, которые будут использоваться при описании алгоритма поиска клонов.

Будем задавать дерево (в т. ч. абстрактное синтасическое дерево) четвёркой (V, E, Σ, L), где V — множество вершин, $E \subset V \times V$ — множество дуг, Σ — множество допустимых пометок и $L : V \to \Sigma$ — функция, назначающая вершинам метки. Также для задания деревьев будем использовать линейную скобочную нотацию (польскую запись), например, запись $+(+(a, b), c)$ соответствует абстрактному синтаксическому дереву выражения `(a+b)+c`. Пусть $\mathbb{T}$ - множество всех деревьев.

Деревья являются частным случаем ориентированных графов без циклов (directed acyclic graph, DAG), и для компактной записи синтаксических структур их целесообразно представлять именно в виде DAG. Например, при таком представлении можно соединить воедино вершины, соответствущие вхождениям имён одной и той же переменной. Опираясь на алгоритмы манипуляции с представлениями термов в виде DAG [18], алгоритм поиска клонов, описанный в данной диссертационной работе, переносится и на представление синтаксиса программ в виде ориентированных графов без циклов.

1.3.1. Антиунификация

Зафиксируем множество $\Upsilon = \{?_1, ?_2, \dots\}$ элементы которого мы будем называть метками-заполнителями. Будем называть те деревья, в которых нет меток-заполнителей, абстрактными синтаксическими деревьями, а все прочие деревья — шаблонами. Подстановками будем называть функции из множества Υ в множество деревьев $\mathbb{T}$.

Результатом применения подстановки σ к дереву T будем называть дерево T', полученное из T путём замены каждого листа, помеченого некоторой меткой-заполнителем $?_i$, на дерево $\sigma(?_i)$. Будем использовать запись $\sigma(T)$ для обозначения результата применения подстановки σ к дереву T. Например, если $\sigma(?_1) = +(x, y)$, $\sigma(?_2) = z$ и $T = +(?_1, ?_2)$, то $\sigma(T) = +(+(x, y), z)$.

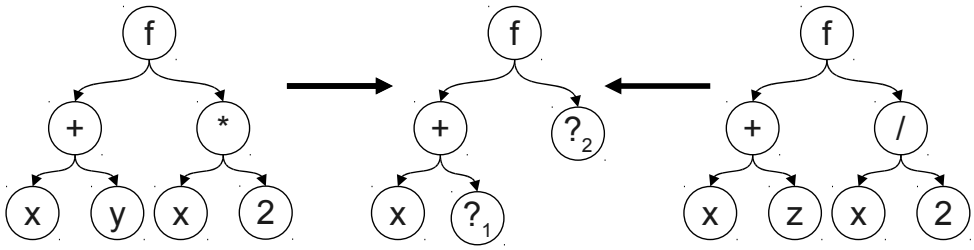

Рисунок 1.6. Пример применения операции антиунификации

Назовём дерево T' уточнением T, если существует подстановка σ такая, что $T' = \sigma(T)$. Назовём T' переименованием T, если T' является уточнением T, и T является уточнением T', будем обозначать это при помощи записи $T \sim T'$. Назовём дерево U общим шаблоном деревьев T_1 и T_2, если существуют подстановки σ_1 и σ_2 такие, что выполняется $\sigma_1(T_1) = \sigma_2(T_2) = U$. Нетрудно заметить, что дерево T' является переименоваием T, если существует подстановка σ, для которой верно $T' = \sigma(T)$ и которая заменяет лишь обозначения меток-заполнителей в листьях дерева T. Например, $+(?_1, a)$ является переименованием $+(?_2, a)$, но не является переименованием $+(?_1, b)$.

Определение 1.2 (Наиболее специальный шаблон (НСШ) [85, 92])**.** Назовём наиболее специальным шаблоном двух деревьев T_1 и T_2 такое дерево $T_A = AU(T_1, T_2)$, что оно является их общим шаблоном и любой другой их общий шаблон T'_A является уточнением T_A.

Антиунификацией будем называть операцию вычисления НСШ двух деревьев.

Легко видеть, что для любых двух деревьев T_1 и T_2 их НСШ существует и определён однозначно с точностью до переименования [85].

На рисунке 1.6 приведён пример НСШ двух деревьев.

В качестве алгоритма построения НСШ может использоваться алгоритм, изложенный в работе [97]. Кроме того, в работе [18] и в диссертации [119]

приводится алгоритм нахождения НСШ, который может быть применён не только к деревьям, но и к ориентированным графам без циклов.

1.3.2. Свойства наиболее специальных шаблонов

НСШ как среднее значение множества деревьев

Легко видеть, что оператор антиунификации является коммутативным и ассоциативным, и, таким образом, его можно применять и к множеству деревьев. НСШ множества деревьев равен наиболее детализированному шаблону, которому удовлетворяют все деревья этого множества. Таким образом, он может восприниматься как "обшая форма", равная общей верхней части всех деревьев множества. Это свойство антиунификации было использовано в работе [78] для нахождения классов часто встречающихся формул в математических текстах.

Расстояние по антиунификации

Пусть c_1 — количество листьев дерева T, помеченных символами-заполнителями и c_2 — количество остальных листьев. Определим размер дерева как $|T| = 0.5c_1 + c_2$. Поясним, почему мы выбрали это определение. Размер абстрактного синтаксического дерева, определённый таким образом, инвариантен относительно способа представления фрагмента в виде АСД, поскольку он равен количеству вхождений всех имён и констант в программу. Например, фрагмент "`a+-b`" может быть представлен как деревом $-(a, b)$ так и деревом $+(a, -(b))$ (в зависимости от использования оптимизаций при построении АСД), но в обоих случаях количество листьев, помеченных метками-заполнителями равно $c_1 = 0$ и количество остальных листьев равно $c_2 = 2$. Поэтому размер обоих этих деревьев будет равен $0.5c_1 + c_2 = 2$.

Обоснование того, почему при вычислении размера дерева вес меток-заполнителей в два раза меньше, чем вес обычных листьев, будет дано после того, как мы определим расстояние по антиунификации между двумя деревьями.

Определение 1.3 (Расстояние по антиунификации)**.** Пусть дерево T_A является НСШ деревьев T_1 и T_2 с подстановками σ_1 и σ_2 соответственно, и пусть Υ_A — множество меток-заполнителей, встречающихся в T_A. Тогда расстоянием по антиунификации между T_1 и T_2 назовём число, равное суммарному размеру всех деревьев, возвращаемых подстановками σ_1 и σ_2 за вычетом числа меток-заполнителей в T_A. Иначе говоря, это расстояние равно $\rho(T_1, T_2) = \sum_{\alpha \in \Upsilon_U} |\sigma_1(\alpha)| + \sum_{\alpha \in \Upsilon_U} |\sigma_2(\alpha)| - |\Upsilon_A|$.

В качестве примера рассмотрим два дерева $T_1 = +(+(i, j), k)$ и $T_2 = +(a, b)$. Их НСШ равен $U = +(?_1, ?_2)$, $\Upsilon_U = \{?_1, ?_2\}$ и подстановки равны $\sigma_1 = \{?_1 \mapsto +(i, j), ?_2 \mapsto k\}$ и $\sigma_2 = \{?_1 \mapsto a, ?_2 \mapsto b\}$ (доопределим σ_1 и σ_2 на $\Upsilon \setminus \Upsilon_U$ произвольным образом). Расстояние по антиунификации между этими двумя деревьями равно $\rho(T_1, T_2) = \sum_{\alpha \in \Upsilon_U} |\sigma_1(\alpha)| + \sum_{\alpha \in \Upsilon_U} |\sigma_2(\alpha)| - 2a * |\Upsilon_U| = (2 + 1) + (1 + 1) - 2 = 3$.

Легко видеть, что расстояние по антиунификации между двумя деревьями равно 0 тогда и только тогда, когда одно дерево является переименованием другого. Следовательно, ρ является семи-метрикой, т. е. для любых деревьев T_1 и T_2 справедливо:

1. $\rho(T_1, T_2) \geq 0$

2. $\rho(T_1, T_2) = 0 \Longleftrightarrow T_1 \sim T_2$ (аксиома тождественности)

3. $\rho(T_1, T_2) = \rho(T_2, T_1)$ (аксиома симметрии)

Однако ρ не является метрикой, поскольку не выполняется условие треугольника. Например, если $T_1 = \cdot(a, b)$, $T_2 = \cdot(a, d)$, $T_3 = \cdot(c, d)$, то $\rho(T_1, T_2) = 1$, $\rho(T_2, T_3) = 1$ и $\rho(T_1, T_3) = 3$, следовательно, $\rho(T_1, T_2) + \rho(T_2, T_3) < \rho(T_1, T_3)$.

Расстояние по антиунификации является частным случаем расстояния редактирования, определённого на множестве деревьев [14], с ограниченным набором операций. Его значение отражает структуру деревьев, и слабо чувствительно к таким операциям, как замена небольшого поддерева на другое небольшое поддерево, но чувствительно к изменению порядка поддеревьев.

1.4. Алгоритм поиска клонов

В данном разделе будет приведён алгоритм поиска клонов, удовлетворяющих определению 1.1.

Алгоритм поиска клонов, основанный на полном переборе всех пар последовательностей операторов, имеет слишком большую сложность. Например, если в программе имеется n блоков, каждый из которых состоит из m операторов, то количество пар последовательностей операторов (т. е. количество потенциальных клонов) равно $\Theta(n^2m^3)$. Таким образом, если в программе имеется 10000 блоков по 10 операторов в каждом, то суммарно необходимо будет провести 10^{12} операций.

Поэтому в данной работе используется эвристический алгоритм поиска клонов. Если прямой алгоритм требует полного перебора всех возможных клонов в независимости от устройства операторов программы, то производительность эвристического алгоритма существенно улучшается, если в программе имеется много похожих (с точки зрения расстояния по антиунификации) операторов. Однако в некоторых случаях эвристический алгоритм может упускать некоторые клоны, удовлетворяющие определению 1.1. В разде-

ле 1.4.4 будет предложен способ, при помощи которого для заданной программы и заданного вида клонов можно получить такие параметры алгоритма, что при их использовании алгоритм будет обладать свойством полноты.

Предлагаемый алгоритм состоит из трёх процедур.

На первом этапе все операторы разбиваются на группы похожих операторов (кластеры). Необходимым условием вхождением двух операторов в один кластер является наличие их общего НСШ. Используется простой однопроходный алгоритм кластеризации. Каждый новый оператор сравнивается с существующими кластерами; в зависимости от результата сравнения он либо добавляется в подходящий кластер, либо формирует новый. В целях эффективности каждый новый оператор сравнивается не со всеми операторами некоторого кластера, а только с их общим НСШ. По окончании первого этапа все операторы будут помечены номерами кластеров, которым они принадлежат.

На следующем этапе проводится поиск одинаково помеченных последовательностей операторов (пометкой оператора является номер кластера, которому он принадлежит). В алгоритме поиска используются суффиксные деревья [53]. По окончании второго этапа имеется множество пар последовательностей; в каждой последовательности на одинаковых позициях находятся операторы, расстояние по антиунификации между которыми не превышает заданного порога. Это, однако, не гарантирует того, что эти пары будут похожи в целом, поскольку число отличий между ними может быть велико. Например, если расстояние между каждой парой операторов $(e_{1,i}, e_{2,i})$, стоящих на одинаковых позициях в последовательностях $(e_{1,1}, e_{1,2}, \ldots, e_{1,n})$ и $(e_{2,1}, e_{2,2}, \ldots, e_{2,n})$, равно 1, то расстояние между последовательностями будет равно n.

На последнем этапе все выявленные одинаково помеченные последова-

тельности операторов сравниваются попарно при помощи антиунификации для окончательного выделения клонов.

Рассмотрим эти три этапа подробнее.

1.4.1. Кластеризация операторов

На первом этапе все операторы разбиваются на группы похожих (с точки зрения расстояния по антиунификации) операторов. Используется простой однопроходный алгоритм кластеризации. Каждый новый оператор сравнивается с существующими кластерами; в зависимости от результата сравнения он либо добавляется в подходящий кластер, либо формирует новый. В целях эффективности каждый новый оператор сравнивается не со всеми операторами некоторого кластера, а только с их НСШ. Кроме того, для того, чтобы ускорить процесс кластериации, применяется хеширование операторов.

Алгоритм кластеризации

```
foreach tree in statement_trees
  bestcluster = argmin(add_cost(cluster, tree))
    if add_cost(bestcluster, tree) < threshold
      bestcluster.append(tree)
    else
      clusters.append(new Cluster(tree))
```

Рисунок 1.7. Алгоритм первого этапа кластеризации

Наша задача — разбить все операторы на группы похожих операторов (кластеры). Аналогичная задача уже решалась в литературе, в частности,

в монографии [116] приводится однопроходный алгоритм кластеризации элементов множества $X = \{x_1, x_2, \dots x_n\}$. В этом алгоритме происходит последовательный перебор всех элементов множества X и расстояние от каждого очередного элемента x до центра каждого построенного к данному моменту кластера сравнивается с пороговой величиной c. Если все эти расстояния превосходят значение порога c, то формируется новый кластер, состоящий из единственного элемента x. Иначе x добавляется в кластер с самым близким к x центром.

Данный алгоритм может быть применён и для кластеризации операторов, если в качестве функции расстояния взять расстояние по антиунификации, а в качестве центров кластеров использовать их НСШ.

Алгоритм кластеризации операторов показан на рисунке 1.7. Рассмотрим, какими свойствами должна обладать функция add_cost(*cluster*, *tree*), используемая в этом алгоритме.

Назовём мерой связанности кластера[1] $C = \{T_1, T_2, \dots, T_n\}$, НСШ которого равен U, величину $\delta(C) = -\sum_{i=1\dots n} \rho(T_i, U)$. Мера связанности кластера отражает, насколько хорошо НСШ кластера приближает его элементы.

Предположим, что мы хотим вычислить стоимость добавления дерева T в кластер C, т. е. вычислить значение функции add_cost(C, T). Предположим также, что C состоит за n элементов, U — НСШ кластера C и U' — НСШ кластера $C \cup \{T\}$ (U' может быть вычислен как $U' = AU(U, T)$). Разумно предположить, что значение $add_cost(C, T)$, должно быть тем больше, чем больше значение $\delta(C) - \delta(C \cup T)$, т. е уменьшение меры связанности кластера C при добавлении в него элемента T. Если для каждого $i = 1 \dots n$ значение $\rho(T_i, U') - \rho(T_i, U)$ приблизить значением $\rho(U, U')$, то значение $add_cost(C, T)$ будет равно $n * \rho(U, U') + \rho(U', T)$. Данное определение

[1] Не путать с понятием меры связности модулей, используемым в программировании

функции $add_cost(C, T)$ удобно тем, что при её вычислении нет необходимости сравнивать новое дерево T *со всеми* элементами кластера C. Кроме того, удобно, что значение функции $add_cost(C, T)$ является большим в следующих случаях:

- в кластере C много элементов и после добавления оператора T в этот кластер их НСШ изменится, или
- НСШ кластера $C \cup \{T\}$ сильно отличается от оператора T.

Во время работы алгоритма 1.7 изменяются кластеры (в них добавляются элементы) и поэтому меняются их НСШ. Это может привести к ситуации, когда один и тот же оператор, встречающийся в начале и в конце кластеризации будет помещен в два разных кластера (даже если это будет редко происходить, такая ситуация очень негативно повлияет на качество поиска).

Поэтому после первого прохода осуществляется второй (корректирующий) проход.

Множество НСШ кластеров (обозначим его A), построенных во время первого прохода, является множеством часто встречающихся шаблонов операторов. Поэтому два оператора, которые похожи на один шаблон, могут считаться похожими и друг на друга. Это свойство используется во втором проходе, в котором каждый оператор помечается тем элементом множества A, на который он больше всего похож (с точки зрения расстояния по антиунификации).

Хеширование операторов

Для того, чтобы ускорить процесс кластериации, в алгоритме может использоваться хеширование операторов.

Определим глубину вершины дерева следующим образом: глубина корня равна нулю, глубина любой другой вершины на единицу больше глубины её предка. Назовём d-срезом дерева T дерево, полученное из T путём удаления всех вершин глубины больше d и удаления всех дуг смежных, таким вершинам [45]. Например, 1-срезом дерева $+(+(a,b),c)$ является дерево $+(+,c)$.

Определение 1.4. Пусть дана некоторая хеш-функция на множестве деревьев. Назовём значением d-хеша дерева T значение этой хеш-функции от d-среза этого дерева.

Таким образом, достаточным условием равенства d-хеша двух деревьев является то, что их верхнии части глубины d совпадают. Например, значения 1-хешей от деревьев $+(+(a,b),c)$ и $+(+(d,-(e,1)),c)$ совпадают.

Будем считать, что между деревьями с различными значениям d-хеша расстояние по антиунификации равно бесконечности. Таким образом, во время кластеризации мы можем сравнивать только деревья, имеющие одинаковое значение d-хеша, и тем самым ускорить процесс кластеризации.

При увеличении глубины хеширования d растёт скорость кластеризации, однако при слишком больших значениях глубины хеширования падает качество кластеризации, поскольку для достаточно больших d даже похожие операторы имеют разные значения d-хеша и поэтому они попадут в разные кластеры. Это приведёт к тому, что не все клоны, удовлетворяющие определению 1.1, будут найдены.

1.4.2. Поиск пар одинаково помеченных последовательностей операторов

После первого этапа рассматриваемого алгоритма каждый оператор имеет пометку, равную номеру кластера, которому он принадлежит.

На следующем этапе проводится поиск всех пар одинаково помеченных последовательностей операторов, т.е последовательностей на одинаковых позициях которых стоят одинаково помеченные операторы. Поиск проводится только среди последовательностей, размер которых не меньше, чем минимальный размер клона (задаётся в качестве параметра алгоритма). Мы применяем стандартный алгоритм поиска таких пар, основанный на использовании суффиксных деревьев [53]. Найденные пары являются кандидатами на клоны и должны быть окончательно проверены на следующем этапе.

1.4.3. Окончательная проверка найденных пар на подобие

После завершения второго этапа мы имеем множество кандидатов на клоны. Эти кандидаты окончательно проверяются во время третьего этапа.

Представим, что имеется кандидат на клон, состоящий из следующих последовательностей операторов: $\{s_1, s_2, \ldots, s_n\}$ и $\{t_1, t_2, \ldots, t_n\}$. Чтобы проверить эту пару на сходство, создадим два новых дерева $\cdot(s_1, s_2, \ldots, s_n)$ и $\cdot(t_1, t_2, \ldots, t_n)$ и вычислим расстояние по антиунификации между ними. Если это расстояние меньше, чем некоторый заранее заданный порог, будем считать этот кандидат клоном, иначе - отбракуем его.

1.4.4. Достаточное условие полноты алгоритма

Приведённый алгоритм является эвристическим, т. е может найти не все клоны, удовлетворяющие определению 1.1. Причиной этого является то, что во время первого этапа (кластеризации) два оператора могут попасть в разные кластеры, хотя на самом деле они похожи и должны находиться на одинаковых позициях двух фрагментов одного клона. Данный клон не будет найден разработанным алгоритмом, поскольку на втором этапе эти два фрагмента будут помечены по-разному.

Однако, мы приведём достаточные условия того, что для заданной программы и для заданного максимального расстояния между фрагментами, составляющими клон, приведённый алгоритм будет обладать свойством полноты. Свойство полноты в данном случае означает то, что будут найдены все клоны, удовлетворяющие заданным в качестве параметров алгоритма ограничениям (минимальный размер и максимальное отличие фрагментов). Для этого при кластеризации операторов мы будем применять только хеширование, т. е. два оператора попадут в один кластер в том и только в том случае, когда у них одинаковое значение d-хеша. Выясним значение параметра d, при котором будут найдены все клоны, расстояние по антиунификации между фрагментами которых не превышает r.

Предположим, что минимальная высота АСД каждого оператора программы равна h; и что каждая вершина АСД программы имеет как минимум k потомков. Легко видеть, что если расстояние между фрагментами f_1 и f_2, составляющими клон, не превышает r, то минимальная глубина вершины АСД операторов f_1 и f_2, в которых эти фрагменты отличаются друг от друга, равна $h - \lceil \log_k r \rceil - 1$. Поэтому для того, чтобы приведённый алгоритм поиска клонов был полным, достаточно потребовать, чтобы глубина хеширования d не превышала $h - \lceil \log_k r \rceil - 1$.

1.4.5. Обобщение на ориентированные графы без циклов

После применения операции “копирования и вставка” (copy&paste) возникает потребность модифицировать вставленный фрагмент так, чтобы он стал согласованным с новым окружением. Часто это заключается в систематическом переименовании: все вхождения одной переменной заменяются на другую переменную. Клоны, в которые входят фрагменты с систематическим переименованием, называются параметризованными клонами (parametrized

clones)[93].

Для того, чтобы уметь обнаруживать параметризованные клоны, необходимо использовать функцию расстояния, которая оценивает все переименования одной переменной как одно переименование.

Приведённый выше алгоритм обнаружения клонов может быть обобщён и для того, чтобы находить параметризованные клоны. Для этого в абстрактных синтаксических деревьях операторов необходимо склеивать воедино все листья, имеющие одинаковые пометки.

1.5. Теоретическое сравнение возможностей разработанного и существующих алгоритмов

В данном разделе мы сравним используемое нами определение клона 1.1 с другими существующими определениями.

1.5.1. Классификация используемого определения понятия клона

В обзоре [93] приведена следующая классификация типов отличий между фрагментами, составляющих один клон:

1. нет отличий,
2. переименования (переменных, функций и классов) и замены одних констант на другие,
3. прочие отличия (вставка/удаление операторов, изменение порядка операторов, замена одних подвыражений на другие),
4. семантическая эквивалентность (при этом не важно, как соотносятся между собой текстовые представления фрагментов).

В определении 1.1 никак не используется семантика операторов языка программирования, следовательно оно не подпадает под тип 4. Поскольку фрагменты, составляющие клон, удовлетворяющий определению 1.1, могут отличаться не только в именах и константах, но и в целых подвыражениях, то определение 1.1 имеет тип 3.

Кроме перечисленных выше четырёх типов клонов, при классификации клонов исследователи обычно используют следующие термины [93]:

- параметризованные клоны (parametrized clones), т. е. клоны, в которых все вхождения одного имени были заменены на другое имя,
- функциональные клоны (function clones), то есть клоны, фрагменты которых состоят из описаний функций,
- клоны с изменением порядка операторов (reorded clones),
- клоны с вставкой/удалением операторов (gapped clones).

В соответствии с замечанием, данным в разделе 1.4.5, приведённый алгоритм позволяет находить параметризованные клоны. Кроме того, поскольку определение функции является частным случаем (составного) оператора, то алгоритм также позволяет находить и функциональные клоны. Для того, чтобы находить клоны с изменением порядка операторов, необходимо при антиунификации считать всех потомков одного родительского составного оператора коммутативными. В этом случае, однако, сложность антиунификации сильно возрастёт (соответствующая ей задача распознавания является NP-полной [9]), поэтому на практике было решено отказаться от поиска клонов с изменением порядка операторов. Кроме того, наш алгоритм не позволяет находить клоны с вставкой и удалением оператов.

1.5.2. Сравнение используемого определения с существующими схожими определениями

Сравним наше определение клонов с наиболее похожими определениями, описанными в литературе.

Как было указано в разделе 1.2.5, наше определение клонов 1.1 в наибольшей степени похоже на определения, данные в работах [45], [26] и [66]. В работах [26] и [66] даются алгоритмы обнаружения клонов, состоящих из последовательностей операторов, в которых допускается переименование переменных. Таким образом, эти алгоритмы позволяют находить клоны второго типа (см. раздел 1.5.1), а не третьего типа, как разработанный нами алгоритм.

Работа [45] является первой работой, в которой для нахождения клонов применялась так называемая структурная, а не лексическая абстракция. Структурная абстракция позволяет установить похожесть операторов `a[x]` и `a[y+1]`, так как они оба являются примером шаблона `a[?]`. Алгоритм Эванса (Evans), описанный в работе [45], производит поиск больших и часто встречающихся шаблонов в абстрактном синтаксическом дереве программы. Он основан на использовании эвристик и строит шаблоны “снизу вверх”, уточняя и увеличивая их шаг за шагом. Например, шаблон `f(?, ?)` может быть уточнён до шаблона `f(1, ?)`, если функция `f` в большинстве случаев вызывается с первым аргументом равным `1`. НСШ, введённые в нашей работе, по сути являются шаблонами. В нашей работе НСШ строятся сверху-вниз, поскольку при увеличении кластеров их НСШ обобщаются. К сожалению, достаточно сложно сравнить полноту и точность поиска метода, изложенного в нашей работе, и метода, изложенного в работе [45]. Но по нашему мнению, описанный в нашей работе подход является более общим, поскольку он основан на таких общих понятиях, как функция расстояния между двумя операторами

и “общая форма” операторов. Это делает возможным использование и других алгоритмов кластеризации помимо описанного.

Главное преимущество нашего подхода перед подходом, изложенным в работе [45], заключается в том, что наш алгоритм позволяет находить клоны, в которые входят *последовательности* операторов, в то время, как алгоритм Эванса позволяет находить только клоны, состоящие из отдельных операторов. Предположим, что фрагмент дублированного кода занимает только часть тела функции. Тогда наш алгоритм позволит найти этот фрагмент целиком, в то время, как алгоритм Эванса сможет найти только отдельные операторы, составляющие этот фрагмент. Если добавить к алгоритму Эванса этап выделения сплошных последовательностей найденных им операторов, то фрагменты, найденные таким алгоритмом не обязательно будут являться клонами. Проблема в том, что эти фрагменты могут состоять из большого числа операторов, и хотя сами операторы, находящиеся на одинаковых позициях, будут похожи (но не полностью совпадать), суммарное отличие между двумя *последовательностями* операторов будет велико.

Стоит отметить, что если рассматривать алгоритм Эванса как алгоритм кластеризации операторов (а не алгоритм поиска клонов), то его можно использовать в качестве этапа кластеризации описанного в данной работе алгоритма. Однако такая связка не будет обладать свойством полноты поиска ни при каких значениях входных параметров.

1.6. Описание практической реализации

Мы реализовали описанный выше алгоритм обнаружения клонов в программно-инструментальном средстве Clone Digger. Разработанное средство написано на языке Python и поэтому является кроссплатформенным. Мы

проверяли его работоспособность на операционных системах Windows и Linux. Средство Clone Digger распространяется под лицензией GNU General Public License (GPL) и может быть загружено с сайта `http://clonedigger.sourceforge.net`.

1.6.1. Поддерживаемые языки программирования

Абстрактные синтаксические деревья для разных языков программирования строятся при помощи специальных модулей. В этих модулях для построения АСД может использоваться информация компилятора/интерпритатора, кроме того модули могут автоматически создаваться при помощи генераторов парсеров на основе грамматик. В настоящее время созданы модули для языка Python версий 2.X (используется стандартная библиотека "compiler") и языков Java 1.5, Javascript, Lua (используется генератор парсеров ANTLR [82]).

1.6.2. Подготовка исходных данных

Как и другие средства обнаружения клонов, Clone Digger предлагается использовать для нахождения клонов не в отдельном файле с исходными текстами, а во всех файлов некоторого проекта. На основании обсуждений средства Clone Digger с его пользователями на конференции EuroPython 2008 были выработаны рекомендации по исключению некоторых файлов из рассмотрения средством.

Во-первых следует избавиться от автоматически сгенерированных исходных текстов. Клоны, найденные в таких файлах, являются, очевидно, бесполезными. Кроме того, автоматически сгенерированные файлы часто содержат длинные последовательности похожих операций (таких, как присваивание значения константе). Это приводит к увеличению длительности работы

Clone # 10971
Distance between two fragments = 1
Clone size = 7

Source file "eclipse-jdtcore/src /internal/compiler/codegen /CodeStream.java" The first line is 66	Source file "eclipse-jdtcore/src/internal /compiler/codegen/CodeStream.java" The first line is 2465
`countLabels = 0;`	`countLabels = 0;`
`stackDepth--;`	`stackDepth--;`
`try { position++; bCodeStream[classFileOffset++] = OPC_aaload; } catch (IndexOutOfBoundsException e) { resizeByteArray(OPC_aaload); } }`	`try { position++; bCodeStream[classFileOffset++] = OPC_invokevirtual; } catch (IndexOutOfBoundsException e) { resizeByteArray(OPC_invokevirtual); }`

Рисунок 1.8. Пример информации о клоне, обнаруженном средством Clone Digger

второго этапа алгоритма (описанного в разделе 1.4.2), поскольку количество непересекающихся подслов одинаковой длины у слова длины n растёт нелинейно в зависимости от n.

Также рекомендуется удалить тесты из дерева пороекта. Часто тесты имеют сходную структуру, однако если применить к ним рефакторинг, то они будут значительно менее понятны человеку.

Кроме того, по очевидным причинам стоит удалить из дерева проекта сторонние библиотеки.

1.6.3. Формат вывода

Clone Digger записывает найденные клоны в HTML файл с подсветкой отличий. На рисунке 1.8 приведён пример одного обнаруженного клона.

1.7. Результаты экспериментальных исследований

Как было указано в разделе 1.2.5, разработанный алгоритм обнаружения клонов больше всего похож на алгоритм, используемый в средстве CloneDR компании Semantic Designs [26], и на алгоритм, используемый в средстве Asta, разработанном в Microsoft Research [45].

К сожалению, средство Asta не находится в открытом доступе, и нам не удалось связаться с его разработчиками.

Средство CloneDR предоставляется на коммерческой основе, однако на сайте компании-разработчика[2] доступна к скачиванию пробная версия. Отчёт пробной версии содержит максимум 9 клонов, судя по всему, имеющих наибольший размер. Понятно, что имея в распоряжении только пробную версию CloneDR, нельзя провести полноценное сравнение CloneDR и Clone Digger, однако в разделе 1.7.1 мы всё же провели некоторое сравнение.

Кроме того, в свободном доступе находятся данные эксперимента Беллона (Bellon) [27]. В этом эксперименте один эксперт вручную сравнивал качество поиска шести различных средств обнаружения клонов, в том числе средства CloneDR. В данной работе в разделе 1.7.2 мы повторили эксперимент Беллона с небольшими отличиями: три эксперта проанализировали клоны, найденные средствами Clone Digger, CloneDR и Dup, и, таким образом, мы смогли сравнить качество поиска этих трёх средств обнаружения клонов. Список клонов, обнаруженных CloneDR и Dup, был взят из общедоступных данных эксперимента Беллона[3].

1.7.1. Сравнение Clone Digger и пробной версии CloneDR

Средство CloneDR поддерживает следующие языки: C, C++, Java, Cobol. Единственным языком, который поддерживает и CloneDR и Clone Digger, является Java, поэтому мы сравнивали эти два средства на проектах, написанных на языке Java.

Мы запустили CloneDR и Clone Digger на проектах netbeans-javadoc[4]

[2] `http://www.semanticdesigns.com`

[3] `http://www.bauhaus-stuttgart.de/clones/index.html`

[4] `http://bits.netbeans.org/dev/javadoc`

Project	Size (loc)	Time	
		CD	CloneDR
netbeans-javadoc	14K	7m57s	2m12s
eclipse-jdtcore	146K	2h43m	1h2m

Рисунок 1.9. Сравнение производительности Clone Digger и CloneDR

и eclipse-jdtcore[5], времена работы приведены в таблице 1.9. Стоит отметить нелинейный рост времени работы в зависимости от размера проекта для обоих средств. Видно, что CloneDR работает быстрее, чем Clone Digger. Одной из причин этого является то, что Clone Digger в отличие от CloneDR написано на интерпретируемом языке программирования (на языке Python).

Как было сказано выше, к загрузке с официального сайта компании-разработчика доступна только пробная версия средства CloneDR, в отчёте которой содержится лишь 9, предположительно, самых больших найденных клонов. Мы убедились в том, что все клоны, которые были выданы пробной версией CloneDR, могут быть обнаружены Clone Digger при некоторых значениях параметров-порогов.

Кроме того, мы изолировали некоторые клоны, найденные Clone Digger, в отдельные файлы, и подали эти файлы на вход CloneDR. При помощи такого трюка можно проверить, способно ли средство CloneDR найти эти клоны, поскольку если оно может это то сделать, то они будут присутствовать и в отчёте пробной версии. Таким образом, мы смогли убедиться, что существуют клоны, которые были найдены Clone Digger, и которые не могут быть найдены CloneDR в принципе (при произвольных значениях порогов). К таким клонам относятся клоны, в которых одни подвыражения (более сложные, чем отдельная константа или идентификатор) были заменены на другие подвы-

[5] `http://www.eclipse.org/jdt/core`

ражения.

1.7.2. Полуавтоматическое сравнение качества поиска средств Clone Digger, CloneDR и Dup

Общее описание эксперимента

В данном разделе мы сравним качество поиска трёх средств: Clone Digger, CloneDR [26] и Dup (средство, работающее на лексемном уровне) [10].

Задача поиска клонов в некотором смысле похожа на задачу информационного поиска (information retrival): имеется некоторый домен X и множество интересующих нас элементов $Y \subseteq X$. Задача заключается в построении по косвенным признакам множества Y', максимально приближенного к Y. Качество информационного поиска можно оценить, зная Y, путём подсчёта количеств ошибок первого и второго рода [79]. К сожалению, данный подход не может быть применён напрямую для оценки качества автоматического поиска клонов, поскольку не существует общепризнанного точного определения клона.

В работе [27] Беллон описывает эксперимент по оценке качества поиска автоматических детекторов клонов, основанный на использовании эталонного множества клонов. Это множество строится вручную: анализируются клоны, найденные одним из нескольких сравнимаемых средств, и те клоны, информация о которых, по мнению эксперта, будет полезна для пользователей, попадают в эталонное множество.

В эксперименте Беллона классификацией клонов занимался только один человек (сам Беллон). В нашем эксперименте принимали участие три аспиранта ЛВК ВМиК МГУ: П. Булычёв, Д. Коваленко и А. Шалимов. Эксперты вручную проанализировали одно и то же множество клонов, и для каж-

Средство	Найдено клонов	Из них просмотрено вручную
Clone Digger	10985	228
CloneDR	2508	60
Dup	18007	349

Рисунок 1.10. Количество всех найденных клонов и клонов, классифицированных вручную

дого клона указали, является ли информация об этом клоне полезной для разработчика, или нет (позже мы укажем критерий, которым они руководствовались). В результатах эксперимента представлены данные, полученные экспертами по отдельности, и данные, полученные в результате голосования (клон попадал в эталонное множество, если 2 из 3 экспертов голосовали за это).

В эксперименте Беллона проводилось сравнение шести средств обнаружения клонов (Dup [10], CloneDR [26], CCFinder [62], Duplix [67], CLAN [73], Duploc [38]) на четырёх проектах на языке C и четырех проектах на языке Java (суммарный размер проектов был 850 тысяч строк кода). В нашей работе мы ограничились сравнением средств Clone Digger, CloneDR и Dup на одном Java-проекте eclipse-jdtcore, состоящим из 146 тысяч строк кода. Мы выбрали для сравнения средство CloneDR, поскольку алгоритм его работы похож на алгоритм работы средства Clone Digger. Средство Dup было выбрано потому, что в эксперименте Беллона оно показало результаты “ортогональные” результатам CloneDR: CloneDR показало высокую точность и низкую полноту поиска, а Dup показало низкую точность и наивысшую полноту поиска. Проект eclipse-jdtcore входил в эксперимент Беллона, и в свободном доступе по адресу [36] находятся отчёты, присланные участниками эксперимента. Мы воспользовались этими данными для нашего эксперимента для средств

CloneDR и Dup.

В эксперименте Беллона эталонное множество формируется на основе классификации не всех найденных клонов (это слишком трудоёмкая задача), а лишь на основе 2% найденных клонов. Ручная классификация 2% найденных клонов заняла у Беллона 77 часов. Мы тоже анализировали только 2% найденных клонов, у каждого из экспертов ручная классификация этого множества заняла примерно 8 часов. В таблице 1.10 приведено количество найденных клонов, и количество клонов, вручную классифицированное экспертами.

В работе [27] не указывается, каким именно критерием руководствовался Беллон при классификации клонов. В нашей работе мы считали, что информация о некотором клоне является полезной, если от него можно избавиться при помощи методов рефакторинга.

Для экономии времени эксперты при классификации анализировали только фрагменты, составляющие клон, но не их контекст. Например, если клон состоял из двух методов, то эксперты не проверяли, принадлежат ли эти методы родственным классам или нет, хотя от этого зависит применимость операции рефакторинга под названием “Подъём метода” (описанной в разделе 1.1). Поэтому эксперты, возможно, совершали ошибки второго рода: ошибочно считали информацию о клоне полезной, хотя по обьективным причинам он не может быть удалён или его удаление не уменьшит стоимость поддержки кода.

В результате эксперты построили три эталонных множества клонов, и четвёртое множество было автоматически построено методом голосования. Для каждого эталонного множества E и каждого рассматриваемого средства t было вычислено две характеристики: полнота (recall) и точность (precision) обнаружения клонов. Полнота — это процент от числа клонов из эталонно-

го множества E, которые были обнаружены средством t. При вычисления данной характеристики допускалось неполное совпадение клонов. Точность — это процент от числа клонов, обнаруженных средством t, которые попали в множество E. Полнота обнаружения выбранного средства тем больше, чем меньше оно допускает ошибок первого рода, а точность обнаружения тем больше, чем меньше количество ошибок второго рода.

В следующем разделе мы формально определим способ вычисления полноты и точности обнаружения клонов.

Описание необходимого математического аппарата

Определение 1.5. Фрагмент кода — это пара $\langle n, l \rangle$, где n — некоторое имя файла с исходными текстами и l — некоторое множество номеров строк в этом файле.

Определение 1.6. Клон — это пара $\langle f_1, f_2 \rangle$, где f_1 и f_2 — фрагменты кода. Будем считать, что фрагменты кода в клоне упорядоченны лексикографически по имени файла, т. е. $n_1 \leq n_2$, где $f_i = \langle n_i, l_i \rangle$, $i = 1, 2$.

Определение 1.7. Пусть даны два клона $c_1 = \langle f_{1,1}, f_{1,2} \rangle$ и $c_2 = \langle f_{2,1}, f_{2,2} \rangle$, где $f_{i,j} = \langle n_{i,j}, l_{i,j} \rangle$, $i, j \in \{1, 2\}$. Определим метрику схожести клонов c_1 и c_2 следующим образом: если $n_{1,1}$ отлично от $n_{2,1}$ или $n_{2,1}$ отлично от $n_{2,2}$, то $similar(c_1, c_2) = \infty$, иначе

$$similar(c_1, c_2) = min\Big(\frac{|l_{1,1} \cap l_{2,1}|}{|l_{1,1} \cup l_{2,1}|}, \frac{|l_{1,2} \cap l_{2,2}|}{|l_{1,2} \cup l_{2,2}|}\Big) \tag{1.1}$$

Пусть у нас имеется множество T средств обнаружения клонов, которые мы хотим сравнить между собой. Будем считать, что эти средства были запущены на одном проекте и $cands(t)$ — множество клонов, обнаруженных средством $t \in T$. Для каждого средства t выберем случайным образом определённый процент клонов из $cands(t)$ и поместим их в множество $oracled_cands(t)$.

В нашем эксперименте мы выбирали 2% клонов (такое же значение использовал Беллон).

После этого эксперт вручную классифицирует клоны из множества $\bigcup_{t \in T} oracled_cands(t)$, и те клоны, от которых по его мнению можно избавиться при помощи рефакторинга, помещаются в множество all_refs. Во время классификации у эксперта нет информации о том, каким средством был обнаружен тот или иной клон.

Пусть $accepted_cands(t)$ — множество клонов из $oracled_cands(t)$, которые были помещены в эталонное множество all_refs. Пусть $matched_cands(t)$ — множество всех $c \in oracled_cands(t)$, для которых существует такой клон $c' \in all_refs$, что $similar(c, c') \geq p$ для некоторого заранее выбранного порога p. В нашем эксперименте, как и в эксперименте Беллона, мы использовали значение p равное 0.7.

Полноту обнаружения средства T определим следующим образом:

$$recall(t) = \frac{|matched_cands(t)|}{|all_refs|} \tag{1.2}$$

Точность обнаружения средства T зададим следующим соотношением:

$$precision(t) = \frac{|accepted_cands(t)|}{|oracled_cands(t)|} \tag{1.3}$$

Результаты эксперимента и выводы

Полученные значения функций $recall$ и $precision$ показаны на диаграммах 1.11 и 1.12.

Показатели средств CloneDR и Dup соотносятся между собой так же, как и в результатах эксперимента Беллона, даже не смотря на то, что Беллон не указал критерий, которым он руководствовался при классификации клонов. CloneDR имеет более высокую точность клонов обнаружения, чем Dup,

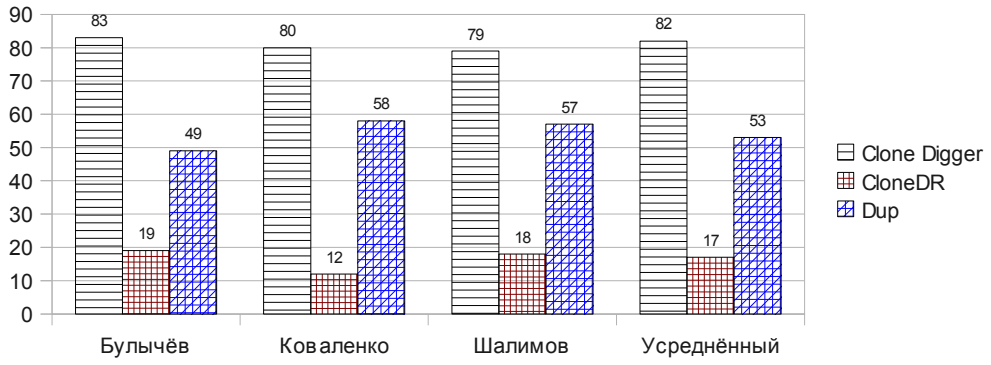

Рисунок 1.11. Сравнение полноты обнаружения клонов

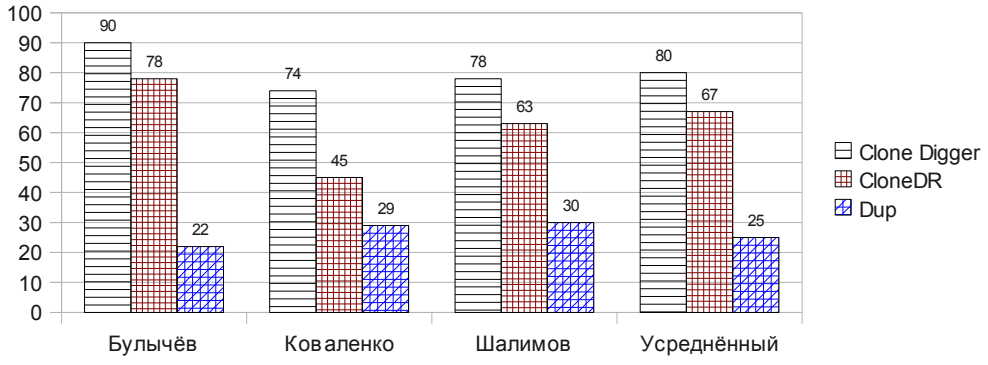

Рисунок 1.12. Сравнение точности обнаружения клонов

но меньшую полноту полноту обнаружения. Это соотносится с результатами теоретического сравнения алгоритмов обнаружения клонов, работающих на лексемном уровне и работающих на уровне абстрактных синтаксических деревьев, приведённого в разделах 1.2.4 и 1.2.5.

Если сравнивать Clone Digger с двумя другими средствами (Clone DR и Dup), то видно, что средство Clone Digger превосходит их как по точности, так и по полноте поиска клонов, от которых можно избавиться при помощи рефакторинга.

1.8. Область применимости

Описанный в данной главе метод позволяет обнаруживать клоны, состоящие из двух фрагментов, один из которых может быть получен из другого заменой некоторых подвыражений на другие подвыражения. Поэтому данный метод позволяет учитывать информацию о синтаксическом устройстве операторов и плохо подходит для обнаружения клонов в программах, написанных на языках низкого уровня (например, байт-коде Java, языках ассемблера), поскольку операторы в таких программах имеют очень простое синтаксическое устройство. Для поиска клонов в таких программах целесообразно применять методы, работащие с представлением программ в виде последовательности лексем [10, 31, 62], поскольку для языков низкого уровня эти методы имеют более высокое быстродействие и находят те же клоны, что и разработанный нами метод.

Кроме того, описанный метод, как и все другие известные методы обнаружения клонов, не в полной мере учитывает семантику программных конструкций. Например, операторы `x=a+b` и `x=b+a`, несомненно, являются клонами с семантической точки зрения (без учёта их размера), но такие клоны

сложно обнаружить, поскольку задачи антиунификации в алгебре, обладающей свойством коммутативности (ассоциативности, идемпотентности) имеют высокую сложность [9].

1.9. Заключение

В данной главе дано новое формальное определение клонов. Главная отличительная особенность этого определения состоит в том, что клоны, удовлетворяющие ему, могут быть легко устранены применением известных алгоритмов, используемых в рефакторинге (реорганизации) программ, основу которых составляет процедурная абстракция. Наряду с этим введённое определение позволяет дать формальную математическую постановку задачи обнаружения клонов.

Описан алгоритм поиска клонов, удовлетворяющих этому определению. Данный алгоритм был реализован в средстве Clone Digger, распространяемом под открытой лицензией GPL. Проведено экспериментальное сравнение результатов работы Clone Digger и двух других средств Asta и CloneDR на основе методики, общепринятой при изучении эффективности средств выявления клонов. Эксперименты показали, что разработанное нами средство превосходит два другие средства как по полноте так и по точности поиска.

Глава 2

Проверка отношений подобия между размеченными системами переходов (структурами Крипке)

2.1. Постановка задачи

Проверка правильности программ — это одна из наиболее трудных задач программирования. Один из подходов к решению этой задачи — верификация моделей программ [118], при которой проверяется правильность не самой программы, а её абстрактного образа (в дальнейшем называемым абстракцией или моделью программы). Модель программы в этом случае должна обладать свойством корректности, т. е. выполнимость спецификации на модели должно вести к её выполнимости и на исходной программе. Верификация моделей программ позволяет получать завершимые алгоритмы, устанавливающие правильность программ.

Модели программы обычно строятся вручную, поскольку их автоматическое построение не всегда возможно. Причиной этого является то, что задача автоматического построения корректной и допускающей автоматическую проверку модели в общем случае является неразрешимой. Если бы это было не так, верификация моделей программ позволяла бы решать задачу проверки завершимости программ на Тьюринг-полных языках (путём сведения проверки останова программы к проверке останова модели). Кроме того, даже если модель уже построена, то в большинстве случаев её корректность нельзя проверить автоматически. Это является следствием того, что в настоящее время не определена формальная семантика многих распространённых

языков (например, C++), компиляторов и библиотек прикладных функций.

По мере верификации модели проверяемые спецификации могут становиться более изощрёнными, и исходная модель может оказаться слишком сложной для автоматического анализа. В этом случае целесообразно построить более абстрактную модель, и впоследствии проводить её автоматический анализ (а не анализ первоначальной модели). Поэтому построение абстрактной модели программы — это многоступенчатый процесс, в котором происходит последовательное упрощение ранее построенных моделей с целью увеличения эффективности верификации.

Корректность каждой очередной модели M_{i+1} может быть доказана по индукции на основе корреткности предыдущей модели M_i и выполнимости между M_{i+1} и M_i отношения подобия специального вида, называемого отношением симуляции [74]. Отношения симуляции интересны тем, что они сохраняют выполнимость спецификаций, заданных формулами темпоральных логик. Это означает, что если некоторая спецификация выполняется на M_{i+1}, то она выполняется и на M_i (следовательно, по индукции она выполняется и на исходной программе). Чем более изощрёнными становятся спецификации, которые мы хотим проверить, тем более простыми должны быть модели, допускающие эффективную верификацию, и тем более гибкими должны быть те абстрактные преобразования, которые приводят к построению этих моделей. Поэтому для доказательства корректности моделей может возникнуть необходимость в использовании различных отношений симуляции, которые бы учитывали эти абстрактные преобразования.

Поскольку используется множество отношений симуляции [15, 35, 43, 65, 105], то на практике существует потребность в создании средства проверки симуляций, которое могло бы быть настроено на проверку произвольного отношения симуляции в зависимости от потребностей пользователя. Допол-

нительным аргументом для создания такого средства является то, что его можно будет настроить на проверку новых (в настоящее время не определённых) отношений. С научной точки зрения разработка данного средства интересно тем, что оно позволит дать ответ на вопрос о том, можно ли сконструировать универсальный алгоритм проверки симуляций, не уступающий по эффективности существующим индивидуальным алгоритмам.

В данной работе ставится задача разработки такого средства проверки симуляций.

2.1.1. Модели программ

В качестве моделей программ мы будем использовать структуры Крипке [118], определение которых даётся ниже:

Определение 2.1. Структура Крипке (размеченная система переходов, LTS) — это пятёрка $\langle S, s_0, R, \Sigma, L\rangle$, где

- S — конечное множество состояний,
- $s_0 \in S$ — начальное состояние,
- $R \subseteq S \times S$ — отношение переходов,
- Σ — множество атомарных переменных,
- $L : S \to 2^{\Sigma}$ — функция разметки состояний.

Структуры Крипке были созданы Солом Крипке в конце 1950х — начале 1960х годов в качестве интерпретаций модальных логик. Впоследствии было обнаружено, что они подходят и в качестве интерпретации темпоральных логик. Когда темпоральные логики стали использоваться для задания спецификаций программ, то обнаружилось, что структуры Крипке — хорошая математическая конструкция для описания моделей программ.

В данной работе структуры Крипке были выбраны в качестве формализма задания моделей, поскольку они используются во многих популярных средствах проверки моделей программ, в частности, в средствах NuSMV [77] и Spin [57].

Множество S, входящее в структуру Крипке — множество состояний программы, среди множества S выделено начальное состояние $s_0 \in S$. Состояние структуры Крипке соответствует "моментальному снимку" или глобальному состоянию вычисления программы, которую моделирует эта структура Крипке. R — множество переходов, в результате выполнения которых изменяется состояние вычисления. Атомарные переменные — предикаты на множестве состояний, с помощью которых описываются конечные данные программы. Поскольку в общем случае число таких атомарных переменных очень велико, то применяется абстракция: выделяется некоторое множество свойств Σ (атомарных переменных), и каждое состояние s характеризуется булевым вектором $L(s)$, показывающим, какие свойства верны в данном состоянии, а какие — нет.

Любая пара состояний $(s_1, s_2) \in R$ называется переходом. Для простоты будем писать $s_1 \to s_2$ вместо $(s_1, s_2) \in R$. Обычно предполагается что отношение переходов является тотальным, т. е. для любого состояния $s_1 \in S$ существует такое состояние s_2, для которого выполняется $s_1 \to s_2$. Для улучшения наглядности рассматриваемых примеров мы не будем придерживаться этого требования.

Вычислением системы переходов M, начинающимся в s_1, является такая бесконечная последовательность состояний $\pi = (s_1, s_2, \dots)$, что для любого неотрицательного i выполняется $s_i \to s_{i+1}$. Обозначим π^i — суффикс π, начинающийся с позиции i.

Способы построения структур Крипке для систем взаимодействующих

процессов описаны в работе [118].

2.1.2. Темпоральные спецификации

Для задания спецификаций поведения программ часто используют темпоральную логику CTL^*, а также её подможества [118]. В этой логике есть два типа формул: формулы состояния и формулы пути. Пусть Σ — множество атомарных переменных.

Синтаксис формул CTL^* определяется следующими правилами:

- если p — атомарная переменная, то p — формула состояния,
- если φ и ψ — формулы состояния, то $\varphi \vee \psi$, $\varphi \wedge \psi$, $\varphi \rightarrow \psi$ и $\neg\varphi$ — формулы состояния,
- если φ — формула пути, то $E\varphi$ и $A\varphi$ — формулы состояния.

Два дополнительных правила необходимы для определения синтаксиса формул пути:

- если φ — формула состояния, то φ — формула пути,
- если φ и ψ — формулы пути, то $\varphi \vee \psi$, $\varphi \wedge \psi$, $\varphi \rightarrow \psi$, $\neg\varphi$, $X\varphi$ и $\varphi U \psi$ — тоже формулы пути.

Тот факт, что формула φ выполняется в состоянии s (или на вычислении π) модели M будем обозначать как $M, s \models \varphi$ ($M, \pi \models \varphi$, соответственно).

Формулы логики CTL^* имеют следующую семантику:

- $M, s \models p$ (где p является атомарной перменной), если $p \in L(S)$,
- $M, s \models \varphi \wedge \psi$, если одновременно выполняется $M, s \models \varphi$ и $M, s \models \psi$ (выполнимость для булевых связок $\vee$, $\neg$, $\rightarrow$ определяется аналогично в соответствии с классической логикой)

- $M, s \models A\varphi$, если для *любого* вычисления π, начинающегося в s, выполняется $M, \pi \models \varphi$,
- $M, s \models E\varphi$, если для *некоторого* вычисления π, начинающегося в s, выполняется $M, \pi \models \varphi$.
- $M, \pi \models X\varphi$, если $M, \pi^1 \models \varphi$,
- $M, \pi \models \varphi U \psi$, если существует такое $j \geq 0$, что $M, \pi^j \models \varphi$ и для всех $0 \leq i < j$ выполняется $M, \pi^i \models \varphi$,
- $M, \pi \models F\varphi$, если существует такое $i \geq 0$, что $M, \pi^i \models \varphi$,
- $M, \pi \models G\varphi$, если для всех $i \geq 0$ справедливо $M, \pi^i \models \varphi$.

Будем говорить, что формула φ выполняется на структуре Крипке M (обозначается $M \models \varphi$), если справедливо $M, s_0 \models \varphi$, где s_0 — начальное состояние M.

Логика ветвящегося времени CTL — это фрагмент CTL^*, в котором каждый темпоральный оператор X, F, G и U должен следовать непосредственно за квантором пути A или E [25]. Будем говорить, что формула находится в позитивной нормальной форме, если все отрицания в ней применяются только к атомарным переменным. Логика ACTL (соответственно, $ACTL$) — это множество формул CTL^* (соответственно CTL) в позитивной нормальной форме, не содержащих квантор E. Логика линейного времени LTL — это множество формул CTL^* вида $A\varphi$, где подформула φ не содержит кванторов E и A [86].

Оператор X используется для описания свойств, которые должны выполняться в *следующем* состоянии вычисления. Поэтому данный оператор не применяется для описания свойств параллельных асинхронных систем,

поскольку порядок выполнения процессов в таких системах недетерминирован [69]. Будем обозначать Λ_X (например LTL_X, CTL_X) фрагмент логики Λ, в котором не используется оператор X.

В качестве примера можно привести темпоральную формулу $AG(p \to Fq)$, которая утверждает, что для любого вычисления, всякий раз, когда в этом вычислении встретится состояние, удовлетворяющее атомарному свойству p, впоследствии в этом вычислении обязательно встретится состояние, удовлетворяющее свойству q. Эта формула принадлежит логике LTL_X, однако существует и эквивалентная ей формула $AG(p \to AFq)$, которая принадлежит $ACTL_X$.

Логика линейного времени LTL поддерживается средством верификации моделей Spin [57], а логика ветвящегося времени CTL — средством NuSMV [77].

2.1.3. Различные виды симуляций

Будем говорить, что бинарное отношение $\preceq$ на множестве LTS сохраняет выполнимость формул логики Λ, если для любой формулы $\varphi \in \Lambda$ и любых двух LTS M_1 и M_2, для которых выполняется отношение $M_1 \preceq M_2$, из $M_2 \models \varphi$ следует $M_1 \models \varphi$. Иными словами, любая формула, выполнимая для M_2, выполняется и для M_1.

Будем рассматривать отношения между размеченными системами переходов (LTS), называемые отношениями симуляции [74]. Неформально отношения симуляции могут быть описаны так. Предположим, что имеется внешний по отношению к моделям программ наблюдатель, который способен обозревать некоторые (необязательно все) события, происходящие по ходу выполнения моделей программы. Две модели находятся в отношении симуляции в том случае, если каждому действию в одной модели соответствует некото-

рое действие (или последовательность действий) в другой модели, и эти действия в двух моделях неотличимы для наблюдателя, т. е. приводят к одному и тому же набору наблюдаемых событий. Налагая различные ограничения на возможности наблюдателя, мы получаем большое разнообразие отношений симуляции. Главное достоинство симуляций состоит в том, что многие отношения симуляции сохраняют выполнимость различных темпоральных и модальных логик [15, 17, 35, 43, 65, 105].

Понятие отношения симуляции между моделями программ было впервые введено Милнером (Milner) в 1971 году в работе [74]. Миллнер рассматривал симуляции только между детерменированными моделями программ и предлагал использовать эти отношения как формализм, позволящий проверять эквивалентность программ на некотором уровне абстракции. Позднее, в 1980 году Миллнер обобщил это понятие симуляционной эквивалентности на случай недетерменированных программ и ввёл понятия строгой и слабой бисимуляций [75]. В 1981 году Парк (Park) обобщил понятие симуляции на случай недетерменированных моделей программ, задаваемыми автоматами Бюхи [81]. Важным свойством отношения симуляции, по словам Парка, является то, что с одной стороны отношение симуляции является более сильным, чем широко используемое отношение трассового (языкового) включения [104], а с другой стороны, проверка выполнимости отношения симуляции является алгоритмически более простой задачей, чем задача проверки трассового включения. В дальнейшем в литературе было приведено множество примеров использования отношений симуляций. В частности, отношения симуляции использовались в процессе инкрементального построения моделей, при котором каждая новая версия модели является уточнением предыдущей [21]. Кроме того, отношения симуляции могут использоваться для вычисления инвариантов бесконечных параметризованных семейств моделей [65], для нахождения

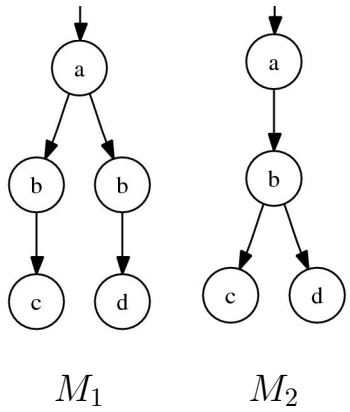

Рисунок 2.1. $M_1 \preceq_{strong} M_2$

симметрии в моделях [106], для автоматического уменьшения размеров моделей [20, 111], для проверки свойств конфиденциальности многопоточных программ [94].

Ниже мы дадим определения строгой и прореженной симуляций (которые в дальнейшем будут использоваться в качестве сквозного примера) и вкратце опишем несколько других симуляций.

Строгая симуляция

Определение 2.2 (Строгая симуляция)**.** Пусть даны две LTS M_1 и M_2, где $M_i = \langle S_i, s_{i0}, R_i, \Sigma_i, L_i \rangle$, $i = 1, 2$, и $\Sigma_1 = \Sigma_2$. Отношение $H \subseteq S_1 \times S_2$ называется отношением строгой симуляции между M_1 и M_2, если для любой пары $(s_1, s_2) \in H$ выполняется:

1. $L_1(s_1) = L_2(s_2)$ и
2. если $(s_1, s_1') \in R_1$, то существует такое состояние $s_2' \in S_2$, что $(s_2, s_2') \in R_2$ и $(s_1', s_2') \in H$.

Отношение $H \subseteq S_1 \times S_2$ называется отношением строгой бисимуляции, если отношение H является отношением строгой симуляции между M_1 и M_2

и отношение H^{-1} является отношением строгой симуляции между M_2 и M_1 (где $H^{-1} = \{(s_2, s_1)|(s_1, s_2) \in H\}$).

Будем говорить что LTS M_1 и M_2 находятся в отношении строгой симуляции (обозначается $M_1 \preceq_{strong} M_2$), если существует отношение строгой симуляции $H \subseteq S_1 \times S_2$ такое, что начальные состояния M_1 м M_2 находятся в этом отношении (т. е. $(s_{10}, s_{20}) \in H$). Аналогично определим отношение строгой бисимуляции (обозначается $M_1 \sim_{strong} M_2$).

Строгая бисимуляция (симуляция) сохраняет выполнимость формул логики CTL^*(ACTL, соответственно) [118].

На рисунке 2.1 показан пример двух систем переходов, между которыми выполняется отношение строгой симуляции. Похожий пример был рассмотрен Миллнером в работе [75] для демонстрации того, что отношение бисимуляции является более слабым, чем отношение изоморфизма. Поскольку строгая симуляция сохраняет логику ACTL, то все формулы этой логики, выполнимые на M_2, выполняются и на M_1. Однако в обратную сторону это утверждение не верно. Например, формула $AG(b \rightarrow ((AG\neg c) \vee (AG\neg d)))$ выполнется на M_1, но не выполняется на M_2.

Прореженная симуляция

Поскольку модели являются абстракциями анализируемых программ, то некоторые свойства программ могут быть никак не отражены в их моделях. В частности, некоторые переменные программы могут не влиять на выполнимость ни одной атомарной переменной (свойства) модели. Это приводит к появлению так называемых невидимых переходов. Будем называть невидимым переходом в LTS $M = \langle S, s_0, R, \Sigma, L\rangle$ такой переход $(s_1, s_2) \in R$, что $L(s_1) = L(s_2)$, остальные переходы будет называть видимыми. Такое определение видимых переходов совпадает с общепринятым в литературе, в част-

ности, в работе Миллнера [75] похожим образом определяются невидимые переходы для LTS с размеченными переходами.

В формулах темпоральных логик, описывающих свойства моделей с невидимыми переходами, нет смысла использовать оператор X, поскольку изменению атомарной переменной могут предшествовать несколько невидимых переходов. Поэтому в качестве отношений симуляции между такими моделями обычно рассматриваются т. н. ослабленные отношения симуляции.

Ослабленные отношения симуляции — это такие отношения, при определении которых, в отличие от строгой симуляции, видимые и невидимые переходы учитываются по-разному. В данной работе мы подробно рассмотрим одного представителя класса ослабленных симуляций: прореженную симуляцию, которая сохраняет выполнимость формул логики $ACTL^*_X$.

Определение 2.3 (Прореженная симуляция[15, 35])**.** Пусть даны две LTS M_1 и M_2, где $M_i = \langle S_i, s_{i0}, R_i, \Sigma_i, L_i \rangle$, $i = 1, 2$, и $\Sigma_1 = \Sigma_2$. Отношение $H \subseteq S_1 \times S_2$ называется отношением прореженной симуляции между M_1 и M_2, если для любой пары $(s_1, s_2) \in H$ выполняется:

1. $L_1(s_1) = L_2(s_2)$ и

2. если $(s_1, s_1') \in R_1$, то в S_2 существует такая последовательность состояний $t_0, t_1, \ldots t_n$ $(n \geq 0)$, что $t_0 = s_2$, $(s_1', t_n) \in H$ и для всех $i < n$ выполняется $(t_i, t_{i+1}) \in R_2$ и $(s_1, t_i) \in H$.

Отношение $H \subseteq S_1 \times S_2$ называется отношением прореженной бисимуляции, если H является отношением прореженной симуляции между M_1 и M_2, а $H^{-1} \subseteq S_2 \times S_1$ является отношением прореженной симуляции между M_2 и M_1.

Будем говорить что LTS M_1 и M_2 находятся в отношении прореженной симуляции (обозначается $M_1 \preceq_{stutt} M_2$), если существует отношение прере-

женной симуляции $H \subseteq S_1 \times S_2$ такое, что $(s_{10}, s_{20}) \in H$ (т. е. начальные состояния M_1 и M_2 находятся в отношении симуляции). Аналогично определим отношение прореженной бисимуляции (обозначается $M_1 \sim_{stutt} M_2$).

Легко показать, что если между двумя моделями выполняется строгая симуляция (бисимуляция), то между ними выполняется и прореженная симуляция (бисимуляция, соответственно).

Прореженная бисимуляция (симуляция) сохраняет выполнимость формул логики CTL^*_X($ACTL^*_X$, соответственно) [35].

Другие ослабленные отношения симуляции

В литературе рассмотрены и другие ослабленные отношения симуляции:

- в работе [43] описана иерархия k_n симуляций $\{H_{k_n}\}$, каждая из которых сохраняет выполнимость формул логики LTL и $H_{k_i} \subseteq H_{k_{i+1}}$ для всех i,
- в работе [105] рассмотрена слабая симуляция, сохраняющая выполнимость формул логики LTL_X,
- в работе [65] использовалась блочная симуляция для поиска инвариантов при верификации бесконечных параметризованных систем.

2.1.4. Задача разработки универсального средства проверки симуляции

В данной работе ставится задача разработки системы проверки симуляций, которая могла бы быть использована для проверки целого ряда отношений.

Такая система должна состоять из двух компонент:

- формального языка, позволяющего описывать различные симуляции,

- программно-инструментального средства, которое по двум размеченным системам переходов и описанию симуляции проверяет, выполняется ли заданная симуляция между этими системами переходов.

2.2. Обзор литературы

В данном разделе мы сравним различные алгоритмы нахождения симуляций, на примере следующих отношений: строгой симуляции, строгой бисимуляции, прореженной симуляции, прореженной бисимуляции. Целью обзора является выбор подхода, который был бы перспективным к нахождению всех отношений, но выбирать его мы будем на примере четырёх вышеуказанных отношений.

2.2.1. Алгоритмы проверки симуляций

Алгоритмы проверки симуляций можно разделить на три категории:

- теоретико-множественные алгоритмы, основанные на нахождении неподвижной точки [54, 89],
- алгоритмы, основанные на поиске наибольшего устойчивого разбиения [47, 50, 89, 95],
- теоретико-игровые алгоритмы [17, 44, 56].

Предположим, что мы проверяем симуляцию между LTS M_1 и M_2, где $M_i = \langle S_i, s_{i0}, R_i, \Sigma_i, L_i \rangle$, $i = 0, 1$. Пусть $n = |S_0| + |S_1|$, $m = |R_0| + |R_1|$.

Для того, чтобы проверить существование некоторой симуляции между M_1 и M_2, необходимо построить отношение симуляции H и проверить, что $(s_{10}, s_{20}) \in H$. Поскольку задача проверки принадлежности элемента множеству является тривиальной, то будем считать что мы решаем только задачу

построения отношения H. Для всех известных определений симуляций, если H_1 и H_2 — отношения симуляции, то $H_1 \cup H_2$ тоже является отношением симуляции. Следовательно, для любой пары LTS M_1 и M_2 существует *максимальное* по включению отношение симуляции, будем считать, что необходимо найти именно максимальное отношение.

Метод неподвижной точки

Опишем схему алгоритма вычисления строгой симуляции, основанного на нахождении неподвижной точки [54]. В данном алгоритме последовательно строятся *приближения сверху* $H_0, H_1, \ldots H_k$ отношения строгой симуляции H до тех пор, пока не будет достигнуто равенство $H_k = H_{k-1} = H$. Первое приближение полагается равным $H_0 = \{(s_1, s_2) | s_1 \in S_1, s_2 \in S_2, L_1(s_1) = L_2(s_2)\}$. На каждом следующем шаге из текущего приближения удаляются все те пары состояний, которые *наверняка* не могут находиться в отношении строгой симуляции. Значение приближения на шаге $i + 1$ вычисляется так: $H_{i+1} = H_i \setminus C_i$, где C_i — это множество всех пар $(s_1, s_2) \in H_i$, для которых включение $(s_1, s_1') \in R_1$ выполняется для некоторого s_1', и не существует s_2' такого, что $(s_2, s_2') \in R_2$ и $(s_1', s_2') \in H_i$.

При использовании некоторых оптимизаций такой алгоритм проверки строгой симуляции имеет сложность $O(mn)$ [54].

Метод наибольшего устойчивого разбиения

Опишем идею алгоритмов, основанных на поиске наибольшего устойчивого разбиения. Разбиением множества Q называется такая система попарно непересекающихся множеств $\rho = \{B_1, \ldots, B_k\}$, что $Q = \bigcup_{B \in \rho} B$. Будем говорить, что разбиение ρ является уточнением разбиения ρ', если для любого $B \in \rho$ существует такое $B' \in \rho'$, что $B \subseteq B'$. Разбиение ρ_i является

наибольшим среди множества разбиений $\{\rho_1, \ldots, \rho_i, \ldots \rho_k\}$, если все разбиения этого множества являются уточнениями ρ_i. Будем говорить, что разбиение ρ устойчиво относительно заданного отношения $T \subseteq Q \times Q$, если для любой пары множеств $B, B' \in \rho$ и для любых $p, q \in B$ выполняется: $T[p] \cap B' = \varnothing \Leftrightarrow T[q] \cap B' = \varnothing$ (где $T[p] = \{(q|(p,q) \in T)\}$). В работе [80] приводится алгоритм нахождения *наибольшего* уточнения заданного разбиения ρ, устойчивого относительно заданного отношения T, который работает за время $O(|T| log(|Q|))$. Обратимся, например, к строгой бисимуляции. Отношение строгой бисимуляции H является симметричным, поэтому H порождает разбиение $\rho_0 = \{B_1, \ldots, B_l\}$ множества $S_1 \cup S_2$, где $s_1, s_2 \in B_i$, тогда и только тогда, когда $(s_1, s_2) \in H$.

В работе [47] показано, что задача поиска строгой бисимуляции эквивалентна задаче поиска наибольшего уточнения ρ_0, устойчивого относительно отношения $R_1 \cup R_2$ (R_i — отношение перехода i-ой модели). Это позволяет вычислять отношение строгой бисимуляции за время $O(m \log n)$. В работе [50] приводится модификация данного алгоритма для нахождения *прореженной* бисимуляции за время $O(mn)$. В работе [95] для ряда симуляций приводятся алгоритмы их проверки, в которых наибольшее устойчивое разбиение вычисляется при помощи символьных методов. В работе [89] для нахождения отношения прореженной симуляции используется комбинация двух подходов: состояния группируются в множества эквивалентных состояний, и одновременно используется теоретико-множественный метод вычисления неподвижной точки. Этот алгоритм имеет сложность по времени $O(n^3 m)$ и по объёму памяти $O(n^2 \log n)$.

2.2.2. Теоретико-игровой подход к проверке симуляций

В теоретико-игровом подходе задача проверки симуляции сводится к задаче нахождения выигрышной стратегии в игре с конечной суммой двух игроков, один из которых (Duplicator) пытается доказать, что выбранное отношение существует, а другой (Spoiler) пытается найти контрпример. Граф игры конечен, и игроки ходят попеременно. Если один из игроков не может сделать очередной ход, то его соперник выигрывает. Если игра продолжается бесконечно долго (как в тесте Тьюринга [102]), то выигрывает Duplicator.

Например, в игре проверки строгой симуляции между M_1 и M_2 у каждого игрока есть фишка, игрок Spoiler передвигает свою фишку по состояниям M_1, а игрок Duplicator — по состояниям M_2. Если ход принадлежит игроку Spoiler и фишка находится на состоянии s_1, то он выбирает некоторый переход $(s_1, s_1') \in R_1$ в M_1 и передвигает свою фишку на состояние s_1'. После этого игрок Duplicator должен передвинуть свою фишку на такое состояние s_2', что существует переход из s_2 в s_2' и состояния s_1' и s_2' одинаково помечены, т. е. $(s_2, s_2') \in R_2$ и $L_1(s_1') = L_2(s_2')$.

Удобство теоретико-игрового подхода заключается в том, что способ нахождения симуляции непосредственно следует из определения, достаточно лишь найти выигрышную стратегию в конечной антагонистической игре двух игроков. Для решения последней задачи существует алгоритм, имеющий линейную по времени сложность [7].

Теоретико-игровой подход может использоваться для нахождения множества отношений, причём в некоторых случаях правила игры непосредственно вытекают из определения симуляции, как в случае с k_n симуляциями [43].

Для всех четырёх рассматриваемых отношений существуют теоретико-игровые алгоритмы их проверки. Так, алгоритмы проверки строгой симуляции и бисимуляции имеет сложность $O(mn)$ [44], а алгоритмы проверки

	симуляция	бисимуляция
строгая	$O(mn)$ время [44]*, [54]***	$O(m \log n)$ время [47]**
прореженная	$O(m^2)$ время и память [17]* $O(n^3 m)$ время $O(n^2 \log n)$ память [89]**	$O(mn)$ время $O(m)$ память [50]***

(*) - теоретико-игровой алгоритм

(**) - алгоритм, основанный на поиске устойчивого разбиения

(***) - алгоритм, основанный на поиске неподвижной точки

Рисунок 2.2. Верхние оценки сложности вычисления строгой и прореженной симуляции и бисимуляции

прореженных симуляций и бисимуляций — $O(m^2)$ [17].

2.2.3. Вывод из обзора

В таблице 2.2 приведены теоретические оценки сложности наиболее эффективных алгоритмов нахождения прореженной и строгой симуляций и бисимуляций.

На основании обзора можно сделать вывод о том, что в основу языка описания симуляций целесообразно положить теоретико-игровой подход, поскольку теоретико-игровой подход годится для описания всех четырёх рассматриваемых отношений: прореженной симуляции и бисимуляции и строгой симуляции и бисимуляции. Кроме того, в некоторых случаях использование теоретико-игрового подхода даёт оптимальные по сложности алгоритмы проверки симуляций.

2.3. Универсальный язык задания симуляций

В данном разделе мы опишем разработанный язык описания симуляций. Вначале мы опишем теоретико-игровой подход к проверке симуляций, лежащий в основе этого языка.

2.3.1. Теоретико-игровой подход к проверке симуляций

Общее описание теоретико-игрового подхода было дано в разделе 2.2.2, ниже будет приведено формальное описание этого подхода

Определение 2.4. Игра — это пятёрка $\langle V_S, V_D, E_S, E_D, v_0 \rangle$, где

- V_S и V_D — множества состояний игроков Spoiler и Duplicator соответственно,
- $E_S \subseteq V_S \times V_D$ и $E_D \subseteq V_D \times V_S$ — множества допустимых ходов,
- $v_0 \in V_S$ — начальное состояние.

Если множества V_S и V_D конечны, то игру будем называть конечной. В дальнейшем будем подрузамевать только конечные игры, поскольку все игры проверки симуляций между конечными структурами Крипке являются конечными.

Пусть $V = V_S \cup V_D$, $E = E_S \cup E_D$ и $Succ(v)$ — функция, возвращающая всех прямых потомков игрового состояния v, т. е. $Succ(v) = \{v' | (v, v') \in E\}$.

История игры — это конечная или бесконечная последовательность игровых состояний $(v_1, v_2, v_3, \dots)$ такая, что $(v_i, v_{i+1}) \in E$ для любого $i \geq 0$. Будем называть стратегией игрока Duplicator функцию $W : V_D \to V_S \cup \{halt\}$ и говорить, что история игры $(v_1, v_2, v_3, \dots)$ удовлетворяет стратегии W игрока P, если для любого состояния $v_i \in V_P$ из истории игры, за исключением,

возможно, последнего, выполняется $v_{i+1} = W(v_i)$, а для последнего состояния (если история игры конечна) выполняется $W(v_i) = halt$. Стратегия W игрока Duplicator называется выигрышной для него в состоянии v (понятие выигрышного состояния было дано в разделе 2.3.1), если любая история игры, начинающаяся в состоянии v и удовлетворяющая стратегии W, либо является бесконечной, либо завершается состоянием игрока Spoiler. Стратегия W игрока Spoiler считается выигрышной для него в v, если любая история игры, начинающаяся в состоянии v и удовлетворяющая стратегии W, завершается состоянием игрока Duplicator. Будем говорить, что состояние v является выигрышным для игрока Duplicator, если у него существует выигрышная стратегия в этом состоянии, иначе состояние v является выигрышным для игрока Spoiler. Назовём игру выигрышной для игрока P, если у игрока P существует выигрышная стратегия в начальном состоянии v_0.

Множество достижимых игровых состояний V_{reach} в игре $G = \langle V_S, V_D, E_S, E_D, v_0 \rangle$ — это множество всех таких $v \in V$, для которых существует конечная история этой игры $(v_0, v_1, v_2, \ldots, v_n)$, удовлетворяющая условию $v_n = v$. Назовём редуцированной версией игры G игру $G' = \langle V'_S, V'_D, E'_S, E'_D, v_0 \rangle$, в которой:

- $V'_S = V_S \cap V_{reach}$,
- $V'_D = V_D \cap V_{reach}$,
- $E'_S = \{(v_S, v_D) | (v_S, v_D) \in E_S \wedge v_S \in V_{reach} \wedge v_D \in V_{reach}\}$,
- $E'_D = \{(v_D, v_S) | (v_D, v_S) \in E_D \wedge v_S \in V_{reach} \wedge v_D \in V_{reach}\}$.

Легко видеть, что в редуцированной игре G' выигрывает тот же игрок, что и в исходной игре G.

Явный алгоритм решения игры проверки симуляции

Для того, чтобы привести и обосновать алгоритм проверки существования выигрышной стратегии игрока Duplicator, нужно ввести понятие разметки. Разметка игры $G = \langle V_S, V_D, E_S, E_D, v_0 \rangle$ — это функция $m : V \to \{0, 1\}$. Разметка называется корректной, если всякая вершина, помеченная 1, является выигрышной для игрока Spoiler. Наша задача — получить *полную* корректную разметку, т. е. разметку, которая сопоставляет 0 только тем вершинам, в которых у игрока Duplicator есть выигрышная стратегия. Определим оператор g, действующий на множестве функций разметки, следующим образом [7]:

$$g(m)(v) = \begin{cases} 1 & \text{если } v \in V_S \text{ и } \exists v' \in Succ(v) : m(v') = 1 \\ & \quad \text{или } v \in V_D \text{ и } |Succ(v)| = 0 \\ & \quad \text{или } v \in V_D \text{ и } \forall v' \in Succ(v) : m(v') = 1 \\ 0 & \text{иначе} \end{cases}$$

Определим отношение частичного порядка на множестве разметок естественным образом: $m_1 \leq m_2$ если для любого игрового состояния v выполняется $m_1(v) \leq m_2(v)$. Множество всех разметок образует конечную решётку. В самом деле, можно рассматривать стратегии как булевы вектора значений во всех игровых состояниях, а, как известно, булевы вектора образуют конечную решётку. Легко видеть, что оператор g является монотонным, т. е. из $m_1 \leq m_2$ следует $g(m_1) \leq g(m_2)$. По теореме Кнастера-Тарского [100] о существовании наименьшей неподвижной точки у монотонного ограниченного оператора у g имеется наименьшая неподвижная точка, т. е. такое m, что $g(m) = m$ и для любого другого m', удовлетворяющего этому свойству, выполняется $m \leq m'$.

Пусть m_0 — разметка, назначающая 0 каждому игровому состоянию.

Наименьшую неподвижную точку m оператора g можно найти, построив последовательность разметок $(m_0, m_1, m_2, \dots)$, где $m_{i+1} = g(m_i) = g^{i+1}(m_0)$; известно, что для конечных решёток эта последовательность сходится к наименьшей неподвижной точке. Это означает, что для всех i больше некоторого k выполняется $m_i = m$. Данная последовательность называется итерационной последовательностью Клини для вычисления наименьшей неподвижной точки монотонных операторов.

Теорема 2.1. *Если m — наименьшая неподвижная точка оператора g, то множество $\{v | v \in V \wedge m(v) = 0\}$ является множеством всех выигрышных игровых состояний игрока Duplicator.*

Доказательство. Будем говорить, что игровое состояние $v \in V$ является выигрышным ранга k для Spoiler (и проигрышным ранга k для Duplicator), если существует стратегия игрока Spoiler, позволяющая перейти в выигрышное для него состояние не более чем за k игровых ходов. Поскольку мы рассматриваем конечную игру, то любое выигрышное состояния игрока Spoiler имеет некоторый конечный ранг. Докажем по индукции, что при $k > 0$ множество $\{v | v \in V \wedge m_k(v) = 1\}$ содержит все выигрышные состояния игрока Spoiler ранга $k - 1$. В самом деле, множество $\{v | v \in V \wedge m_1(v) = 1\}$ является множеством выигрышных состояний игрока Spoiler ранга 0, поскольку оно содержит все те игровые состояния игрока Duplicator, из которых он не может сделать переход. Справедливость индуктивного перехода является следствием того, что игровое состояние игрока Spoiler (соответственно Duplicator) v является выигрышным ранга k, где $k > 1$, тогда и только тогда, когда хотя бы один его потомок (соответственно, все его потомки) являются выигрышными ранга $k - 1$. □

Таким образом, для того, чтобы убедиться, что игрок Duplicator выиг-

```
function Solve
  m := m_0
  while g(m) ≠ m do m := g(m)
  return (m(v_0) ≡ 0)
```

Рисунок 2.3. Алгоритм решения игры

рывает в игре G, достаточно найти такое k, что $m^{k+1}(m_0) \equiv m^k(m_0)$, и проверить, что $m^k(m_0)(v_0) = 0$. Этот алгоритм приведён на рисунке 2.3. Легко видеть, что последовательность Клини сходится спустя не более чем $|V| + 1$ шагов, и на каждом шаге для построения очередной функции разметки требуется $|V||E|$ операций. Таким образом, сложность алгоритма Solve равна $O(|V|^2|E|)$.

В описанном алгоритме на каждой итерации функция разметки строится заново. Если *обновлять* её значения лишь в тех вершинах, разметка одного из последователей которых изменилась на предыдущих итерациях, то можно добиться сложности $O(|E|)$ [7]. Однако для реализации такого алгоритма необходимо иметь явное перечисление всех игровых состояний и переходов, и это трудно совместимо с символьным способом представления множеств, широко используемым при верификации.

Правила игр для нахождения некоторых отношений

В некоторых случаях правила игры для нахождения симуляции непосредственно вытекают из определения симуляции. Для более сложных отношений необходимо вручную создавать правила игры и доказывать их корректность. В качестве примера здесь приводятся правила игры для проверки

строгой бисимуляции, описанные в работах [44, 76, 98], и правила игры для нахождения прореженной симуляции, описанные в работе [17].

Строгая бисимуляция

Рассмотрим две размеченные системы переходов M_1 и M_2, где $M_i = \langle S_i, s_{i0}, R_i, \Sigma_i, L_i \rangle$, $i = 1, 2$. Зададим игру $G_{\sim_{strong}, M_1, M_2} = \langle V_S, V_D, E_S, E_D, v_0 \rangle$ для нахождения строгой бисимуляции между M_1 и M_2:

- $V_S = \{(s_1, s_2)_S | s_1 \in S_1 \wedge s_2 \in S_2\}$,
- $V_D = \{(s_1, s_2, b)_D | s_1 \in S_1 \wedge s_2 \in S_2 \wedge b \in \{1, 2\}\}$,
- $v_0 = (s_{10}, s_{20})$,
- $E_S = E_{S1} \cup E_{S2}$, где:
  - $E_{S1} = \{((s_1, s_2)_S, (s_1', s_2, 1)_D) \mid (s_1, s_1') \in R_1\}$,
  - $E_{S2} = \{((s_1, s_2)_S, (s_1, s_2', 2)_D) \mid (s_2, s_2') \in R_2\}$,
- $E_D = E_{D1} \cup E_{D2}$, где:
  - $E_{D1} = \{((s_1, s_2, 1)_D, (s_1, s_2')_S) \mid (s_2, s_2') \in R_2 \wedge L_1(s_1) = L_2(s_2')\}$,
  - $E_{D2} = \{((s_1, s_2, 2)_D, (s_1', s_2)_S) \mid (s_1, s_1') \in R_1 \wedge L_1(s_1') = L_2(s_2)\}$,

Теорема 2.2. [76, 98] *Для любых двух размеченных систем переходов M_1 и M_2 выполняется: $M_1 \sim_{strong} M_2$ тогда и только тогда, когда игрок Duplicator выигрывает в игре $G_{\sim_{strong}, M_1, M_2}$.*

Прореженная симуляция

Рассмотрим две размеченные системы переходов M_1 и M_2, где $M_i = \langle S_i, s_{i0}, R_i, \Sigma_i, L_i \rangle$, $i = 1, 2$. Зададим игру $G_{\preceq_{stutt}, M_1, M_2} = \langle V_S, V_D, E_S, E_D, v_0 \rangle$

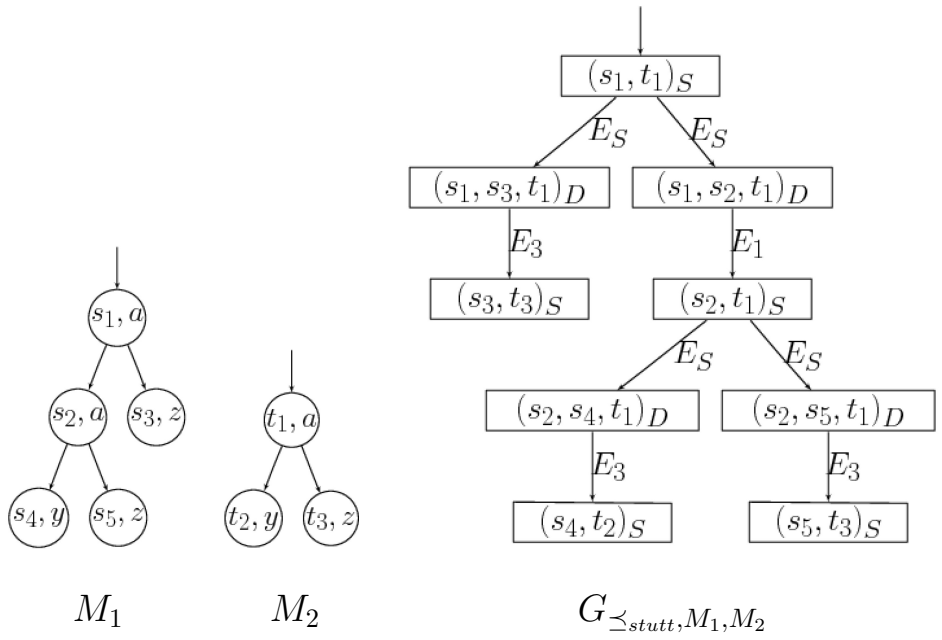

Рисунок 2.4. Пример игры для проверки прореженной симуляции

для нахождения прореженной симуляции между M_1 и M_2:

- $V_S = \{(s_1, s_2)_S | s_1 \in S_1 \wedge s_2 \in S_2\}$,
- $V_D = \{(s_1, s_1', s_2)_D | s_1 \in S_1 \wedge s_1' \in S_1 \wedge s_2 \in S_2\}$,
- $v_0 = (s_{10}, s_{20})$,
- $E_S = \{((s_1, s_2)_S, (s_1, s_1', s_2)_D) \mid (s_1, s_1') \in R_1\}$,
- $E_D = E_1 \cup E_2 \cup E_3$, где
  - $E_1 = \{((s_1, s_1', s_2)_D, (s_1', s_2)_S) \mid L_2(s_2) = L_1(s_1')\}$,
  - $E_2 = \{((s_1, s_1', s_2)_D, (s_1, s_2')_S) \mid (s_2, s_2') \in R_2 \wedge L_2(s_2') = L_1(s_1)\}$,
  - $E_3 = \{((s_1, s_1', s_2)_D, (s_1', s_2')_S) \mid (s_2, s_2') \in R_2 \wedge L_2(s_2') = L_1(s_1')\}$.

Будем говорить, что в модели M есть цикл по одинаково помеченным вершинам, если в M есть вычисление $\pi = (s_0, s_1, \dots)$, в котором для всех m, начиная с некоторого n, выполняется $L(s_m) = L(s_n)$.

Теорема 2.3. [17] *Для любых двух размеченных систем переходов M_1 и M_2 без циклов по одинаково помеченным вершинам выполняется: $M_1 \preceq_{stutt} M_2$ тогда и только тогда, когда игрок Duplicator выигрывает в игре $G_{\preceq_{stutt},M_1,M_2}$.*

В работе [17] также приводится способ проверки прореженной симуляции между моделями с циклами одинаково помеченных вершин, который заключается в предварительном избавления от таких циклов.

На рисунке 2.4 показаны две модели M_1 и M_2, для которых выполняется $M_1 \sim_{stutt} M_2$, и, следовательно, выполняется $M_1 \preceq_{stutt} M_2$ и $M_2 \preceq_{stutt} M_1$; и приведён граф игры $G_{\preceq_{stutt},M_1,M_2}$.

2.3.2. Теоретико-игровой язык описания симуляций

Нами было замечено, что для всех известных отношений симуляции $\preceq$ задача проверки выполнимости отношения $M_1 \preceq M_2$ между двумя LTS сводится к задаче поиска выигрышной стратегии в антагонистической игре двух игроков $G_{\preceq,M_1,M_2}$. В этой игре множество позиций игроков определяется множествами состояний LTS M_1 и M_2, но форма правил самой игры зависит только от отношения симуляции $\preceq$. Таким образом, отношение симуляции $\preceq$ можно рассматривать как семейство игр, имеющих некоторую общую форму правил игры.

Подобно тому, как целый класс отношений на различных предметных областях можно описать одной формулой логики предикатов, бесконечные семейства игр с различными множествами позиций игроков, но общей формой правил игры, можно описать одним формальным выражением, которое будем называть *шаблоном игры*.

Каждая из игр $G_{\preceq,M_1,M_2} = \langle V_S, V_D, E_S, E_D, v_0 \rangle$, соответствующая известным отношениям симуляции, определяется следующими компонентами:

1. Множествами $V_S = S_1^{n_1} \times S_2^{n_2} \times \{0,1\}^{n_3}$ и $V_D = S_1^{m_1} \times S_2^{m_2} \times \{0,1\}^{m_3}$ допустимых позиций игроков *Spoiler* и *Duplicator*. Таким образом, тип позиций игроков определяется парой троек $\langle n_1, n_2, n_3 \rangle$ и $\langle m_1, m_2, m_3 \rangle$.

2. Отношениями игровых переходов (правилами игры) E_S и E_D; каждое из указанных отношений представляет собой некоторое подмножество декартового произведения $S_1^{n_1+m_1} \times S_2^{n_2+m_2} \times \{0,1\}^{n_3+m_3}$. Эти отношения могут быть заданы посредством типизированных формул логики предикатов φ_S и φ_D в некоторой фиксированной сигнатуре (одна из таких сигнатур будет приведена ниже).

3. Начальной позицией v_0, зависящей только от начальных состояний LTS M_1 и M_2.

На основании проведённого анализа устройств игр $G_{\preceq,M_1,M_2}$ для проверки строгой и прореженной симуляции и бисимуляции можно ввести определение шаблона игры $GT_{\preceq}$, порождающего множество игр $G_{\preceq}[M_1, M_2]$ для всевозможных пар LTS M_1 и M_2.

В формулах, описывающих правила игры, могут использоваться переменные 6 типов:

- $\mathcal{X}_S^1$, $\mathcal{X}_D^1$ — два множества переменных, значениями которых могут являться состояния LTS M_1,
- $\mathcal{X}_S^2$, $\mathcal{X}_D^2$ — два множества переменных, значениями которых могут являться состояния LTS M_2,
- $\mathcal{X}_S^3$, $\mathcal{X}_D^3$ — два множества булевых переменных.

Переменные из множеств $\mathcal{X}_S^1$, $\mathcal{X}_S^2$ и $\mathcal{X}_S^3$ используются для обозначения компонентов позиции игрока Spoiler, а переменные из множеств $\mathcal{X}_D^1$, $\mathcal{X}_D^2$ и $\mathcal{X}_D^3$ — для обозначения компонентов позиции игрока Duplicator.

Правила игры описываются формулами логики предикатов сигнатуры $\sigma = \langle\{R_1, R_2\}, \{L_1, L_2\}, \{0, 1\}\rangle$. Эта сигнатура состоит из двух двухместных предикатных символов R_1 и R_2, используемых для обозначения отношений переходов LTS, двух одноместных функциональных символов L_1 и L_2, используемых для обозначения функций разметки состояний LTS, а также из двух булевых констант. В сигнатуре σ правильно построенными термами считаются только переменные и термы $L_1(x)$ и $L_2(y)$, где $x \in \mathcal{X}_S^1 \cup \mathcal{X}_D^1$, $y \in \mathcal{X}_S^2 \cup \mathcal{X}_D^2$. В качестве атомарных формул разрешается использовать булевы константы, переменные из множеств $\mathcal{X}_S^3$, $\mathcal{X}_D^3$, а также формулы вида $R_1(X, Y)$, $R_2(X, Y)$, $t_1 = t_2$, где X, Y — переменные одного из четырех типов $\mathcal{X}_p^i$, $i \in \{1, 2\}$, $p \in \{S, D\}$, а t_1, t_2 — термы.

Определение 2.5. Шаблоном игры называется система $\langle n_1, n_2, n_3, m_1, m_2, m_3, \varphi_S, \varphi_D\rangle$, состоящая из

- шести натуральных чисел $n_1, n_2, n_3, m_1, m_2, m_3$, определяющих тип игры,
- $\varphi_S(\bar{x}_S^1, \bar{x}_S^2, \bar{x}_S^3, \bar{x}_D^1, \bar{x}_D^2, \bar{x}_D^3)$, $\varphi_D(\bar{x}_S^1, \bar{x}_S^2, \bar{x}_S^3, \bar{x}_D^1, \bar{x}_D^2, \bar{x}_D^3)$ — формулы логики предикатов с равенством сигнатуры σ, зависящих от $n_1, n_2, n_3, m_1, m_2, m_3$ свободных переменных типов $\mathcal{X}_S^1$, $\mathcal{X}_S^2$, $\mathcal{X}_S^3$, $\mathcal{X}_D^1$, $\mathcal{X}_D^2$ и $\mathcal{X}_D^3$ соответственно.

Любая пара LTS $M_i = \langle S_i, s_{i0}, R_i, \Sigma_i, L_i\rangle$, $i = 1, 2$, задает интерпретацию I_{M_1,M_2} для формул сигнатуры σ. В этой интерпретации значениями переменных из множеств $\mathcal{X}_S^i$ и $\mathcal{X}_D^i$ служат состояния LTS M_i, оценкой предикатных символов R_i служит отношение переходов LTS M_i, а оценкой функциональных символов L_i служит функция разметки LTS M_i.

Определение 2.6. Примером шаблона игры $GT = \langle n_1, n_2, n_3, m_1, m_2, m_3, \varphi_S, \varphi_D\rangle$ для LTS $M_1 = \langle S_1, s_{20}, R_1, \Sigma_1, L_1\rangle$ и

$M_2 = \langle S_2, s_{20}, R_2, \Sigma_2, L_2 \rangle$ называется игра $GT[M_1, M_2] = \langle V_S, V_D, E_S, E_D, v_0 \rangle$, в которой:

- $V_S = S_1^{n_1} \times S_2^{n_2} \times \{0,1\}^{n_3}$,
- $V_D = S_1^{m_1} \times S_2^{m_2} \times \{0,1\}^{m_3}$,
- отношение игровых переходов E_S (соответственно E_D) состоит из всех таких пар (v_S, v_D) (соответственно (v_D, v_S)), представляющих игровые позиции $v_S \in V_S$ и $v_D \in V_D$, для которых формула φ_S (соответственно φ_D) выполняется в интерпретации I_{M_1,M_2} на наборе (v_S, v_D) значений свободных переменных, т. е.

$$\begin{aligned} (v_S, v_D) \in E_S \quad &\Longleftrightarrow \quad I_{M_1,M_2} \models \varphi_S(\bar{x}_S^1, \bar{x}_S^2, \bar{x}_S^3, \bar{x}_D^1, \bar{x}_D^2, \bar{x}_D^3)[v_S, v_D], \\ (v_D, v_S) \in E_D \quad &\Longleftrightarrow \quad I_{M_1,M_2} \models \varphi_D(\bar{x}_S^1, \bar{x}_S^2, \bar{x}_S^3, \bar{x}_D^1, \bar{x}_D^2, \bar{x}_D^3)[v_S, v_D], \end{aligned}$$

- начальную игровую позицию v_0 представляет набор $(\{s_{10}\}^{n_1}, \{s_{20}\}^{n_2}, \{0\}^{n_3})$.

Описание строгой бисимуляции в рамках разработанного языка

Зададим шаблон игры $GT_{\sim_{strong}} = \langle n_1, n_2, n_3, m_1, m_2, m_3, \varphi_S, \varphi_D \rangle$ для проверки строгой бисимуляции. Для этого необходимо переформулировать определение игры $G_{\sim_{strong},M_1,M_2}$ из раздела 2.3.1. Положим $n_1 = n_2 = m_1 = m_2 = m_3 = 1$, $n_3 = 0$ и $\bar{x}_S^1 = (s_1)$, $\bar{x}_S^2 = (s_2)$, $\bar{x}_S^3 = \varnothing$, $\bar{x}_D^1 = (d_1)$, $\bar{x}_D^2 = (d_2)$, $\bar{x}_D^3 = (d_3)$. Формула $\varphi_S(\bar{x}_S^1, \bar{x}_S^2, \bar{x}_S^3, \bar{x}_D^1, \bar{x}_D^2, \bar{x}_D^3)$ имеет следующий вид:

$$\begin{aligned} &\big((s_2 = d_2) \wedge d_3 \wedge R_1(s_1, d_1)\big) \vee \\ &\big((s_1 = d_1) \wedge \neg d_3 \wedge R_2(s_2, d_2)\big) \end{aligned} \tag{2.1}$$

Формула $\varphi_D(\bar{x}_S^1, \bar{x}_S^2, \bar{x}_S^3, \bar{x}_D^1, \bar{x}_D^2, \bar{x}_D^3)$ имеет следующий вид:

$$\begin{gathered}\big(L_1(s_1) = L_2(s_2)\big)\wedge \\ \big(((s_1 = d_1) \wedge d_3 \wedge R_2(s_2, d_2))\vee \\ ((s_2 = d_2) \wedge \neg d_3 \wedge R_1(s_1, d_1))\big)\end{gathered} \tag{2.2}$$

Описание прореженной симуляции в рамках разработанного языка

Зададим шаблон игры $GT_{\preceq_{stutt}} = \langle n_1, n_2, n_3, m_1, m_2, m_3, \varphi_S, \varphi_D\rangle$ для проверки прореженной симуляции Для этого необходимо переформулировать определение игры $G_{\preceq_{stutt}, M_1, M_2}$ из раздела 2.3.1. Положим $n_1 = n_2 = m_2 = 1$, $n_3 = m_3 = 0$, $m_1 = 2$ и $\bar{x}_S^1 = (s_1)$, $\bar{x}_S^2 = (s_2)$, $\bar{x}_S^3 = \bar{x}_D^3 = \varnothing$, $\bar{x}_D^1 = (d_1, d_2)$, $\bar{x}_D^2 = (d_3)$. Формула $\varphi_S(\bar{x}_S^1, \bar{x}_S^2, \bar{x}_S^3, \bar{x}_D^1, \bar{x}_D^2, \bar{x}_D^3)$ имеет следующий вид:

$$(s_1 = d_1) \wedge (s_2 = d_3) \wedge R_1(s_1, d_2) \tag{2.3}$$

Формула $\varphi_D(\bar{x}_S^1, \bar{x}_S^2, \bar{x}_S^3, \bar{x}_D^1, \bar{x}_D^2, \bar{x}_D^3)$ имеет следующий вид:

$$\begin{gathered}\big(L_1(s_1) = L_2(s_2)\big)\wedge \\ \big(((s_1 = d_2) \wedge (s_2 = d_3))\vee \\ ((s_1 = d_1) \wedge R_2(d_3, s_2))\vee \\ ((s_1 = d_2) \wedge R_2(d_3, s_2))\big)\end{gathered} \tag{2.4}$$

Прочие отношения, которые можно описать в рамках разработанного языка

Кроме описанных выше шаблонов игр проверки строгой бисимуляции и прореженной симуляции, в рамках разработанного языка мы описали следующие отношения:

- строгую симуляцию и прореженную бисимуляцию, в соответствии с правилами, описанными в работах [44] и [17],
- k_n симуляцию для $n = 2$ [43], устройство игры непосредственно вытекало из определения симуляции,
- слабую симуляцию [105], при определении которой в множество допустимых предикатов пришлось добавить предикаты $R_1^*(s_1, s_2)$ и $R_2^*(s_1, s_2)$, предикат $R_i^*(s_1, s_2)$ истинен, если существует *невидимое* вычисление из s_1 в s_1' и $(s_1', s_2) \in R_i$.

2.4. Универсальный символьный алгоритм проверки симуляций

Символьные алгоритмы верификации работают со сложными структурами данных (формулами, уравнениями, логическим схемами и др.), позволяющими компактно описывать конечные множества. В данной главе для удобства записи будет использоваться формульное представление множеств.

Приведём символьный алгоритм, который по шаблону игры $GT_{\preceq}$ и по двум LTS M_1 и M_2 проверяет, является ли игра $G = GT_{\preceq}[M_1, M_2]$ выигрышной для игрока Spoiler. Для того, чтобы сократить пространство поиска выигрышной стратегии, наш алгоритм строит редукцию игры G, т. е. удаляет из неё все недостижимые игровые состояния и переходы. Алгоритм является модификацией алгоритма Эделькампа (Edelkamp), описанного в работе [39], и состоит из следующих трёх процедур:

1. Построение отношения игрового перехода.
2. Построение множества достижимых игровых состояний и удаление недостижимых переходов.

3. Решение игры:

   а. разбиение достижимых игровых состояний на выигрышные и проигрышные состояния игрока Spoiler,

   б. проверка принадлежности начального игрового состояния множеству выигрышных состояний игрока Spoiler.

Главным отличием нашего алгоритма от алгоритма Эделькампа заключается в том, что алгоритму Эделькампа на вход подаётся символьное представление начального множества проигрышных состояний каждого из игроков. В нашем алгоритме это множество необходимо строить во время работы алгоритма (в него входят те состояния, из которых не существует ходов).

При описании алгоритма мы не будем детализировать, каким образом реализованы операции над формулами. Нам лишь достаточно, чтобы библиотека работы с формулами поддерживала все логические операции ($\neg$, $\wedge$, $\vee$), квантификацию ($\forall$, $\exists$) и проверку на эквивалентность ($\equiv$). После описания алгоритма мы остановимся на особенностях его реализации при помощи упорядоченных двоичных разрешающих диаграмм (OBDD).

2.4.1. Построение отношения игрового перехода

Пусть даны две LTS M_1 и M_2, где $M_i = \langle S_i, s_{i0}, R_i, \Sigma_i, L_i \rangle$, $i = 1, 2$ и $\Sigma_1 = \Sigma_2 = \Sigma$. Будем считать, что множества состояний S_1 и S_2 конечны. Тогда и множества переходов R_1 и R_2 конечны и, следовательно, они представимы формулами φ_{R_1} и φ_{R_2} логики предикатов с равенством над сигнатурами $\langle \varnothing, \varnothing, S_1 \rangle$ и $\langle \varnothing, \varnothing, S_2 \rangle$ соответственно. Подставив эти формулы вместо предикатов R_1 и R_2 в формулах φ_S и φ_D, мы получаем формулы логики предикатов с равенством φ'_S и φ'_D над сигнатурой $\langle \varnothing, \{L_1, L_2\}, \{0, 1\} \cup S_1 \cup S_2 \rangle$.

Как было сказано выше в разделе 2.3.2, в формулах φ_S и φ_D функциональные символы не могут быть вложены в другие фунциональные символы и предикаты. Поэтому функции L_i в этих формулах могут встречаться только в подформулах вида $L_i(x) = L_j(y)$ (где $i, j \in \{1, 2\}$). Очевидно, что такие подформулы можно заменить на формулы логики предикатов с равенством над сигнатурой $\langle \varnothing, \varnothing, S_1 \cup S_2 \rangle$. После этой замены из формул φ'_S и φ'_D мы получаем формулы ψ_{E_S} и ψ_{E_D} над сигнатурой $\langle \varnothing, \varnothing, S_1 \cup S_2 \rangle$.

Формулы ψ_{E_S} и ψ_{E_D} зависят от переменных $\mathcal{X}_S$ и $\mathcal{X}_D$ и характеризуют множества допустимых ходов в игре $GT_{\preceq}[M_1, M_2]$.

Зададим также формулу ψ_{v_0}, характеризующую начальное состояние игры:

$$\psi_{v_0}(\mathcal{X}_S) \equiv \bigwedge_{x \in \mathcal{X}_S^1} (x = s_{10}) \wedge \bigwedge_{x \in \mathcal{X}_S^2} (x = s_{20}) \wedge \bigwedge_{x \in \mathcal{X}_S^3} (x = 0) \tag{2.5}$$

2.4.2. Построение множества достижимых игровых состояний и удаление недостижимых переходов

Формулы $\psi_{E_S}(\mathcal{X}_S, \mathcal{X}_D)$ и $\psi_{E_D}(\mathcal{X}_S, \mathcal{X}_D)$ описывают отношения переходов *нередуцированной* игры, в которой могут присутствовать недостижимые переходы. Наша задача — построить символьное представление *редуцированной* игры. Алгоритм, изображённый на рисунке 2.5, последовательно строит множества достижимых состояний, а затем оставляет только достижимые игровые переходы.

Легко видеть, что процедура BuildReachable завершается, поскольку всевозможные значения формулы $\psi_{V_S}(\mathcal{X}_S)$ образуют конечную решётку по включению, и в этой решётке не существует бесконечной строго возрастающей последовательности.

Корректность алгоритма BuildReachable доказывает следующая теоре-

```
procedure BuildRechable
  ψ_{V_S}(X_S) := ψ_{v_0}(X_S)
  do
    ψ'_{V_S}(X_S) := ψ_{V_S}(X_S)
    ψ_{V_D}(X_D) := ψ_{V_D}(X_D) ∨ ∃X_S(ψ_{V_S}(X_S) ∧ ψ_{E_S}(X_S, X_D))
    ψ_{V_S}(X_S) := ψ_{V_S}(X_S) ∨ ∃X_D(ψ_{V_D}(X_D) ∧ ψ_{E_D}(X_S, X_D))
  while ψ_{V_S}(X_S) ≢ ψ'_{V_S}(X_S)
  ψ_{E_S}(X_S, X_D) := ψ_{E_S}(X_S, X_D) ∧ ψ_{V_S}(X_S)
  ψ_{E_D}(X_S, X_D) := ψ_{E_D}(X_S, X_D) ∧ ψ_{V_D}(X_D)
```

Рисунок 2.5. Алгоритм построения редуцированной игры

ма.

Теорема 2.4. *После завершения работы процедуры BuildReachable формулы $\psi_{V_S}(\mathcal{X}_S)$ и $\psi_{V_D}(\mathcal{X}_D)$ характеризуют множества достижимых состояний игроков S и D соответственно.*

Доказательство. Докажем справедливость утверждения только для состояний игрока S, для состояний игрока D оно доказывается аналогично. Пусть $\psi^i_{V_S}(\mathcal{X}_S)$ — значение формулы $\psi_{V_S}(\mathcal{X}_S)$ на i-ой итерации цикла. Легко видеть, что формула $\psi^i_{V_S}(\mathcal{X}_S)$ характеризует все те состояния, которые достижимы из начального состояния максимум за $2i$ шагов. Поскольку формула $\psi_{V_S}(\mathcal{X}_S)$ эквивалентна $\vee_{i=0}^{k} \psi^i_{V_S}(\mathcal{X}_S)$ (где k — номер последней итерации цикла), то $\psi_{V_S}(\mathcal{X}_S)$ характеризует множество игровых состояний, достижимых из начального за любое конечное число шагов, что и требовалось доказать. □

```
function SymbSolve
  SW(X_S) := false
  DL(X_D) := false
  do
    SW'(X_S) := SW(X_S)
    DL'(X_D) := DL(X_D)
    SW(X_S) := ∃X_D(ψ_{E_S}(X_S, X_D) ∧ DL'(X_D))
    DL(X_D) := ∀X_S(ψ_{E_D}(X_S, X_D) → SW'(X_S)) ∨ (ψ_{V_D}(X_D) ∧ ¬∃X_S(ψ_{E_D}(X_S, X_D))
  while (SW(X_S) ∨ DL(X_D)) ≢ (SW'(X_S) ∨ DL'(X_D))
  return (SW(X_S) ∧ ψ_{v_0}(X_S)) ≡ false
```

Рисунок 2.6. Алгоритм нахождения выигрышных состояний игрока Spoiler

2.4.3. Решение игры

После завершения процедуры BuildReachable мы имеем символьное представлением каждого компонента редуцированной игры. Теперь наша задача — разбить все игровые состояния на выигрышные и проигрышные для игрока Spoiler.

На рисунке 2.6 приведена модификация описанного ранее алгоритма Solve, в которой используются формульные представления множеств. Процедура SymbSolve итеративно строит формулы $SW(\mathcal{X}_S)$ и $DL(\mathcal{X}_D)$. На каждой итерации формула $SW(\mathcal{X}_S) \vee DL(\mathcal{X}_D)$ характеризует *некоторое* подмножество множества выигрышных для Spoiler состояний, а после выхода из цикла эта формула характеризует множество *всех* такие состояний. Завершимость и корректность алгоритма SymbSolve следует из доказанных выше завершимости и корректности алгоритма Solve.

В конце процедуры SymbSolve происходит проверка принадлежности

начального состояния множеству выигрышных состояний Spoiler. Если $SW(\mathcal{X}_S) \wedge \psi_{v_0}(\mathcal{X}_S) \equiv false$, то выигрывает Duplicator. Это свидетельствует о том, что симуляция выполняется. В противном случае симуляция не выполняется.

2.4.4. Особенности реализации алгоритма при помощи двоичных разрешающих диаграмм

При описании алгоритма мы не детализировали, каким образом проводятся операции над формулами. Одним из широко распространённых способов представления формул являются упорядоченные двоичные разрешающие диаграммы (OBDD) [16]. OBDD — это каноническая форма представления булевых функций $f : \{0,1\}^n \to \{0,1\}$. Часто представление функций в виде OBDD оказывается значительно более компактным, чем традиционные нормальные формы, такие как дизъюнктивная нормальная форма и коньюнктивная нормальная форма. Упорядоченные двоичные разрешающие диаграммы нашли широкое применение во многих задачах верификации моделей программ.

Как показал Бриант (Bryant) в 1986 году в своей основополагающей работе [16], операции над функциями, заданных в виде OBDD, имеют следующую сложность:

- OBDD функции, заданной булевой формулой от n переменных, в худшем случае имеет размер (занимает память) $O(2^n/n)$;
- построение конъюнкции и дизъюнкции функций f_1 и f_2 отнимает время $O(n_1 n_2)$, где n_1 и n_2 — размеры OBDD функций f_1 и f_2;
- проверка двух функций на эквивалентность и построение дополнения функции может быть произведено за линейное время.

В формулах, используемых в нашем алгоритме, все переменные принимают значение из конечных множеств S_1, S_2 или $\{0, 1\}$. Элементы каждого конечного множества X можно закодировать при помощи булева вектора длины $\lceil log_2|X|\rceil$. Это позволяет перейти от каждой используемой формулы φ, не содержащей функциональные символы, к булевой формуле $\varphi_{\mathcal{B}}$ такой, что формула φ истинна на наборе v тогда и только тогда, когда формула $\varphi_{\mathcal{B}}$ истинна на наборе $v_{\mathcal{B}}$, где $v_{\mathcal{B}}$ — булево представление набора v.

Таким образом, все операции над формулами, используемые в приведённом алгоритме проверки симуляций, можно реализовать при помощи операций над OBDD.

В равной мере для символьного представления моделей и игры могут быть использованы и другие математические конструкции: регулярные выражения, прямое формульное представление и др. OBDD были выбраны ввиду широкой распространённости библиотек для манипуляции с ними.

2.5. Описание практической реализации

В рамках данной диссертации было разработано программно-инструментальное средство, которое по описаниям двух моделей и описанию отношения (би)симуляции в виде шаблона игры определяет, выполняется ли заданное отношение между этими моделями.

В качестве языка описания моделей мы использовали язык верификатора NuSMV [77] потому, что исходные коды NuSMV предоставляются под лицензией GPL, и его разработчики заявляют что NuSMV может использоваться в качестве платформы для разработки сторонних средств, используемых в верификации моделей. Кроме того, средство NuSMV может использоваться для проверки свойств ветвящегося времени, выполнимость которых

сохраняют отношения симуляции.

Работа средства проверки симуляций состоит из следующих этапов. Вначале проводится разбор правил игры проверки симуляции, заданных в виде шаблона игры. Далее при помощи изменённой версии NuSMV конструируются OBDD представлений двух моделей. После этого запускается алгоритм проверки симуляций, и в случае, если симуляция не выполняется, пользователю выдаётся контрпример. Этот контрпример фактически является выигрышным подграфом (стратегией) игрока Spoiler. Для представления контрпримеров используется формат Graphviz [49].

Разработанное средство написано на языке Python и работает в операционной среде Linux. В качестве библиотеки работы с OBDD используется библиотека Cudd [32]. Сама библиотека Cudd реализована на языке C++, работа с ней из Python осуществлялась с помощью враппера PyCudd [87].

2.6. Применение средства проверки симуляции для инкрементального построения модели

В этом разделе рассматривается пример применения разработанной среды проверки симуляции.

Зачастую при построении модели допускаются ошибки, которые не могут быть выявлены на этапе автоматической верификации [24]. Существует метод инкрементального построения моделей программ, который в некоторых случаях поможет избежать таких ошибок [21]. В этом методе последовательно строятся модели программ $M_1, \ldots, M_n$ так, что каждая следующая модель является уточнением предыдущей модели и содержит больше подробностей о конструируемой программе. Разумно предположить, что между моделями M_{i+1} и M_i (где $1 \leq i < n$) выполняется некоторое отношение симуляции

(которое выбирается исходя из характера отличий между M_{i+1} и M_i). Если же это отношение симуляции не выполняется, то контрпример, выданный средством проверки симуляций, поможет найти ошибку в модели M_{i+1}.

Рассмотрим пример построения модели, описывающей одно из решений задачи обедающих философов [115]. На примере этой задачи часто рассматривают более общую задачу распределения ресурсов, возникающую при проектировании распределённых программ. Напомним, что в задаче обедающих философов имеется N философов, сидящих за круглым столом, перед каждым философом находится тарелка, и кроме того между каждыми двумя философами лежит одна вилка (т. е. всего имеется N вилок). Каждый философ может начать есть только когда он взял вилки слева и справа от себя, а положить вилки он может только после того, как закончил есть. Задача заключается в организации взаимодействия между философами так, чтобы исключить возможность тупика (deadlock), т. е. ситуации, когда каждый философ держит вилку в левой руке, и ни один философ не может взять вилку справа от себя и приступить к еде.

Вначале мы создали модель A, в которой не накладывалось никаких ограничений на действия философов, и поэтому был возможен тупик.

В модели B появляется маркер; каждый философ может получить маркер только когда вилки слева и справа от него лежат на столе. После того, как философ получил маркер, он может взять вилки и начать есть; после того как он поел, он освобождает маркер. В 1971 году Дейсктрой было доказано, что в этой модели тупик не достижим [37], кроме того это свойство можно можно проверить автоматически при помощи NuSMV.

В модели C маркер движется строго по кругу, и каждый философ обязан поесть прежде, чем он передаст маркер своему соседу. Будем считать, что модель C и является окончательной моделью анализируемого способа

взаимодействия философов.

Рассмотрим, какое отношение симуляции должно выполняться между моделями A и B и моделями B и C. Переменная, задающая положение маркера (назовём её m), должна быть невидимой при вычислении симуляции, поскольку в модели A она отсутствует, а в модели C, в отличие от модели B, маркер не должен освобождаться при переходе от одного философа к другому. Изменение переменной m происходит во время невидимых переходов, и поэтому необходимо использовать ослабленное отношение симуляции. Можно предположить, что между моделями B и A и между моделями C и B должно выполнятся отношение прореженной симуляции, где все переменные являются видимыми за исключением переменной m.

При исследовании моделей оказалось, что между моделями B и A выполняется не только отношение прореженной симуляции, но и отношение строгой симуляции. Причина этого заключается в том, что из-за особенностей языка NuSMV в модели A в каждом состоянии есть переход в это же состояние, и эти переходы соответствуют невидимым переходам в модели B.

В таблице 2.7 приведены результаты проверки $B \preceq_{strong} A$ и $C \preceq_{stutt} B$ для различных количеств обедающих философов. Среда проверки симуляций запускалась на компьютере с процессором AMD Athlon 5200+. Стоит отметить, что большое число состояний в моделях A и B обусловлено так называемым экспоненциальным ростом числа состояний (в зависимости от количества параллельных процессов, составляющих модель), а описание самих моделей занимает меньше 100 строчек кода на языке NuSMV. Число состояний модели C и размер игры для нахождения симуляции между B и C растёт линейно в зависимости от числа процессов, однако размер модели B растёт экспоненциально, и для её построения требуется больше времени, чем для решения игры.

Строгая симуляция: $B \preceq_{strong} A$

N_{proc}	$N(A) \times N(B)$	$T(A) \times T(B)$	N_{game}	N_{obdd}	$Time$
4	$1.2 \cdot 10^3 \times 1.6 \cdot 10^3$	$5.7 \cdot 10^3 \times 8.0 \cdot 10^3$	6676	$1.9 \cdot 10^4$	0m3s
5	$6.5 \cdot 10^3 \times 1.0 \cdot 10^4$	$3.6 \cdot 10^4 \times 6.1 \cdot 10^4$	$4.1 \cdot 10^4$	$1.0 \cdot 10^5$	0m17s
6	$3.4 \cdot 10^4 \times 6.3 \cdot 10^4$	$2.2 \cdot 10^5 \times 4.5 \cdot 10^5$	$2.4 \cdot 10^5$	$5.7 \cdot 10^5$	1m38s
7	$1.7 \cdot 10^5 \times 4.0 \cdot 10^5$	$1.3 \cdot 10^6 \times 3.2 \cdot 10^6$	$1.4 \cdot 10^6$	$2.9 \cdot 10^6$	10m58s
8	$9.2 \cdot 10^5 \times 2.5 \cdot 10^6$	$7.4 \cdot 10^6 \times 2.2 \cdot 10^7$	$8.3 \cdot 10^6$	$1.5 \cdot 10^7$	71m7s

Прореженная симуляция: $C \preceq_{stutt} B$

N_{proc}	$N(B) \times N(C)$	$T(A) \times T(B)$	N_{game}	N_{obdd}	$Time$
7	$63 \times 1.7 \cdot 10^5$	$154 \times 1.3 \cdot 10^6$	1044	6222	7s
8	$72 \times 9.2 \cdot 10^5$	$176 \cdot \times 7.6 \cdot 10^6$	1369	9068	22s
9	$81 \times 4.6 \cdot 10^6$	$198 \times 4.3 \cdot 10^7$	1738	$1.1 \cdot 10^4$	1m23s
10	$90 \times 2.3 \cdot 10^7$	$220 \times 2.3 \cdot 10^8$	2151	$1.4 \cdot 10^4$	4m41s
11	$99 \times 1.1 \cdot 10^8$	$244 \times 1.3 \cdot 10^9$	2608	$1.7 \cdot 10^4$	16m19s

N_{proc} — количество процессов (обедающих философов) в модели,

$N(M)$ — количество состояний в модели M,

$T(M)$ — количество переходов в модели M,

N_{game} — количество достижимых состояний в игре для проверки симуляции,

N_{OBDD} — суммарный размер OBDD, описывающих множества достижимых игровых состояний первого и второго игроков,

$Time$ — занятое время при выполнении на процессоре AMD Athlon 5200+.

Рисунок 2.7. Результаты исследований на моделях универсального средства проверки симуляций, реализованного в рамках данной работы

Стоит отметить, что время работы разработанного средства для данного примера оказалось существенно меньшим теоретических оценок сложности задач проверки симуляций, приведённых в таблице 2.2. Так, теоретико-игровой алгоритм проверки симуляции имеет временную сложность $O(mn)$, где m — суммарное количество переходов, а n — суммарное количество состояний в моделях. Из таблицы 2.7 видно, что при увеличении количества философов на 1 количество переходов в моделях возрастает в среднем 6.8 раз, а суммарное количество состояний — в 5.9 раз. Поэтому теоретическая верхняя оценка сложности проверки строгой симуляции между A и B растёт экспоненциально в зависимости от числа философов, и основание степенной функции примерно равно 40. Однако, время работы алгоритма растёт экспоненциально с основанием степени, примерно равным 6. Такая разница между реальным временем работы алгоритма и его теоретической наихудшей оценкой является следствием того, что в приведённом алгоритме проводится решение *редуцированной* игры, т. е. игры, содержащей только достижимые состояния (это описано в разделе 2.4.2). В то же время, при выводе теоретических оценок сложности теореко-игрового алгоритма считалось, что все состояния игры являются достижимыми.

2.7. Область применимости

В данной главе решалась задача создания средства проверки симуляций, которое могло бы быть настроено на проверку произвольного отношения симуляции между моделями, заданными структурами Крипке. В основе разработанного средства лежит *формальный язык* для задания отношения симуляции и *алгоритм*, который по двум моделями и определению симуляции на данном языке проверяет, выполняется ли это отношение симуляции

между этими моделями.

Показано, что разработанный формальный язык достаточно богат для описания строгой симуляции, строгой бисимуляции, прореженной симуляции, прореженной бисимуляции, k_2 симуляции, слабой и блочной симуляций. В то же время этот язык не подходит для описания т. н. справедливых симуляций [44], т. е. симуляций, определённых на множестве структур Крипке с ограничениями справедливости [118]. В справедливых структурах Крипке считаются допустимыми только те вычисления, которые бесконечно часто проходят через так называемые принимающие состояния. Эта особенность справедливых структур Крипке позволяет их использовать, например, для задания модели системы параллельных процессов, в которой каждый процесс рано или поздно будет поставлен на выполнение. К сожалению сложность нахождения справедливых симуляций слишком высока [44], поэтому в наше средство мы не стали добавлять возможность нахождения справедливых отношений симуляции.

Разработанный алгоритм может быть применён только к конечным моделям, т. е. структурам Крипке с конечным числом состояний. Поэтому он не может быть применён к проверке симуляции между структурами Крипке, порождёнными высокоуровневыми описаниями, в которых используется рекурсия неограниченной вложенности.

2.8. Заключение

В данной главе был описан формальный язык, позволяющий задавать различные отношения подобия (симуляции) между моделями программ посредством правил антогонистической игры двух игроков. В терминах предложенного языка мы привели полное описание ряда отношений (строгой си-

муляции, строгой бисимуляции, прореженной симуляции, прореженной бисимуляции). Кроме того, были выявлены общие приёмы, при помощи которых на разработанном нами языке можно описать и другие отношения симуляции (k_2 симуляция, слабая и блочная симуляции). Это показывает, что предложенный язык описания симуляций имеет хорошие выразительные возможности.

Также разработано программно-инструментальное средство, которое позволяет по описанию симуляции на теоретико-игровом языке и описанию двух моделей на языке NuSMV, проверять выполняется ли данная симуляция между двумя данными моделями. Для эффективности работы данного средства были использованы символьные представления моделей и игры проверки симуляции.

Проведено экспериментальное исследование разработанного средства при инкрементальной разработке моделей программ на примере различных вариантов задачи обедающих философов.

Глава 3

Проверка отношений подобия между временными автоматами

Во второй главе были рассмотрены отношения подобия (симуляции) между моделями программ, заданными структурами Крипке.

Структуры Крипке хорошо подходят для описания моделей программ, работающих в дискретном времени. Однако, часто возникает необходимость в анализе программ, работающих в реальном времени. Такие программы могут содержать переменные-таймеры, значениями которых могут быть вещественные числа. При написании таких программ приходится учитывать и проверять течение реального времени. Для описания моделей программ с таймерами хорошо подходят временные автоматы, введённые Алюром (Alur) и Диллом (Dill) в 1994 году [5]. По существу, временные автоматы — это конечные автоматы, снабжённые конечным множеством таймеров и ограничениями на значения таймеров, при которых допустимы те или иные переходы. Пока автомат находится в одном состоянии, значения всех таймеров растут с одинаковой скоростью, при переходе из состояния в состояние значения некоторых таймеров могут сбрасываться в 0. В работах [4, 5] было показано, что для временных автоматов являются разрешимыми многие задачи анализа, включая задачу проверки достижимости выбранного состояния и задачу проверки выполнимости формул темпоральной логики Timed CTL(TCTL). Благодаря этим результатам временные автоматы можно использовать в качестве моделей программ при решении задач верификации информационных систем, работающих в реальном времени.

Временные автоматы также используются в качестве удобной математи-

ческой модели при решении задачи синтеза систем автоматического управления. Эта задача синтеза состоит в построении такого алгоритма работы устройства управления (контроллера), что объект управления будет корректно функционировать в независимости от действий среды (такие системы называются системами автоматического управления). При решении этой задачи система автоматического управления может быть представлена автоматом, множество переходов которого состоит из переходов двух типов, контролируемых и неконтролируемых устройством управления. Такой автомат фактически представляет собой описание игры двух участников — контроллера и внешней среды. И действительно, в работе [71] было показано, что задача синтеза работы контроллера сводится к задаче нахождения его выигрышной стратегии в этой игре. В этой же работе было установлено, что задача синтеза разрешима для случая, когда цель контроллера — перевести систему за определённое время из начального состояния в заданное состояние, избегая при этом попадания в некоторые выделенные (аварийные) состояния (такие свойства можно задавать формулами логики ATCTL).

Хотя задача синтеза систем автоматического управления и разрешима, большой размер моделей делает затруднительным применение этих алгоритмов на практике. Поэтому, как и в случае с структурами Крипке, использование подходящего отношения симуляции позволило бы повысить эффективность алгоритмов анализа временных автоматов.

К сожалению, за счёт наличия таймеров временные автоматы как правило имеют бесконечное число состояний. Это обстоятельство составляет главную трудность при разработке алгоритмов проверки симуляций на временных автоматах. В частности, в литературе не было определено отношение симуляци, заданное на важном классе игровых временных автоматов.

В данной работе ставится задача разработки формального определения

временной симуляции между игровыми временными автоматами и построение алгоритма проверки этой симуляции.

3.1. Постановка задачи

3.1.1. Условные обозначения

Будем обозначать $\mathbb{R}_{\geq 0}$ множество всех неотрицательных вещественных чисел, а $\mathbb{Z}_{\geq 0}$ — множество всех неотрицательных целых чисел. Пусть X и Y — некоторые множества. Будем использовать запись X^Y для обозначения множества всех отображений из X в Y. Это расходится с привычным в теории множеств обозначением Y^X, но такая запись представляется нам в дальнейшем более удобной.

3.1.2. Временной автомат

Пусть X — конечное множество вещественных переменных. Будем использовать запись $\mathcal{B}(X)$ для обозначения множества формул, удовлетворяющих грамматике:

$$\varphi ::= x \sim k \mid \varphi \wedge \varphi, \tag{3.1}$$

где $k \in \mathbb{Z}_{\geq 0}$, $x \in X$ и $\sim \in \{<, \leq, =, >, \geq\}$.

Ниже будет введён ряд определений, необходимых для описания временного автомата и его семантики (эти определения в том или ином варианте присутствуют в большинстве работ по временным автоматам).

Оценка (valuation) переменных из X — это отображение из X в $\mathbb{R}_{\geq 0}$, т. е. элемент $\mathbb{R}_{\geq 0}^X$. В терминологии, используемой в теории схем программ, оценка переменных из X является интерпретацией множества переменных X в области интерпретации $\mathbb{R}_{\geq 0}$ [120].

Будем обозначать $\vec{0}$ оценку, сопоставляющую 0 каждой переменной.

Пусть Y — некоторое подмножество X; тогда будем использовать запись $v[Y]$ для обозначения оценки переменных из X, сопоставляющей 0 всем $x \in Y$, и $v(x)$ всем $x \in X \setminus Y$. Запись $v[Y]$ будет использоваться для обозначения того, что таймеры из Y были сброшены в 0. В качестве примера рассмотрим множество переменных $Y = \{x_2\}$ и оценку $v = \{x_1 \mapsto 1, x_2 \mapsto 2\}$, для которых верно $v[Y] = \{x_1 \mapsto 1, x_2 \mapsto 0\}$.

Будем использовать запись $v|_Y$ для обозначения оценки переменных из Y, для которой верно $v(x) = v|_Y(x)$ для всех $x \in Y$. Эта запись будет использоваться для выделения значений некоторых таймеров. Например, если верно $v = \{x_1 \mapsto 1, x_2 \mapsto 2\}$ и $Y = \{x_1\}$, то $v|_Y = \{x_1 \mapsto 1\}$.

Пусть даны два непересекающихся множества вещественных переменных X_1 и X_2, и две оценки $v_1 \in \mathbb{R}_{\geq 0}^{X_1}$ и $v_2 \in \mathbb{R}_{\geq 0}^{X_2}$. Будем использовать запись $v_1 \otimes v_2$ для обозначения такой оценки переменных из $X_1 \cup X_2$, что $(v_1 \otimes v_2)(x)$ равно $v_1(x)$, если $x \in X_1$, и $v_2(x)$ — иначе. Иными словами, эта запись обозначает то, что две оценки непересекающихся множеств таймеров объединяются в одну. В качестве примера можно рассмотреть оценки $v_1 = \{x_1 \mapsto 1\}$ и $v_2 = \{x_2 \mapsto 2\}$, для которых верно $v_1 \otimes v_2 = \{x_1 \mapsto 1, x_2 \mapsto 2\}$.

Пусть $\delta \in \mathbb{R}_{\geq 0}$, тогда будем обозначать $v + \delta$ такую оценку X, что для всех $x \in X$ выполняется $(v+\delta)(x) = v(x)+\delta$. Эта запись будет использоваться в том случае, когда все переменные оценки увеличиываются на одно и то же число. Например, для оценки $v = \{x_1 \mapsto 1, x_2 \mapsto 2\}$ верно $(v + 1) = \{x_1 \mapsto 2, x_2 \mapsto 3\}$.

Если $g \in \mathcal{B}(X)$ и $v \in \mathbb{R}_{\geq 0}^{X}$, то будем считать, что выполняется $v \models g$, если v удовлетворяет формуле g. Запись $v \models g$ обозначает то, что формула g выполняется на оценке v. В качестве примера рассмотрим оценку $v = \{x_1 \mapsto 1, x_2 \mapsto 2\}$, для которой выполняется $v \models (x_1 \leq 1)$ и не выполняется $v \models (x_2 \leq 1)$.

Будем обозначать $[\![g, X]\!]$ множество $\{v \in \mathbb{R}^X_{\geq 0} \mid v \models g\}$. Если X ясно из контекста, то для простоты будем использовать запись $[\![g]\!]$.

Кроме приведённых определений мы будем использовать следующее свойство, являющееся аналогом свойства выпуклости для линейных пространств. Для каждой формулы g и для любой оценки v и любого $\delta \in \mathbb{R}_{\geq 0}$, если $v \models g$ и $v + \delta \models g$, то и $v + \delta' \models g$, где $0 \leq \delta' \leq \delta$.

Определение 3.1 ([5]). Временной автомат (timed automata, TA) — это шестёрка $\langle S, s_0, X, \Sigma, R, Inv \rangle$, где

- S — конечное множество, элементы которого называются дискретными состояниями,
- $s_0 \in S$ — начальное дискретное состояние,
- X — конечное множество вещественных переменных (таймеров),
- Σ — множество пометок переходов (действий), включающее в себя так называемое невидимое действие τ,
- $R \subseteq S \times \mathcal{B}(X) \times \Sigma \times \wp(X) \times S$ — конечное множество переходов ($\wp(X)$ — множество всех подмножеств X),
- $Inv : S \to \mathcal{B}(X)$ — функция, сопоставляющая дискретным состояниям формулы (такие формулы будем называть инвариантами).

Определим размеченную систему переходов $(Q, q_0, \to)$, описывающую операционную семантику временного автомата. Множество состояний $Q = S \times \mathbb{R}^X_{\geq 0}$ — это множество пар, включающих в себя некоторое дискретное состояние и некоторую оценку переменных из X. Начальным состоянием автомата является пара $q_0 = (s_0, \vec{0})$.

Определим теперь отношение $\to$, определённое на множестве состояний автомата. Будем называть предохранителем перехода (s, g, a, Y, s') формулу g. Если множество R содержит переход (s, g, a, Y, s'), то это означает, что временной автомат может совершить переход по действию a из дискретного состояния s в дискретное состояние s' при условии, что значения таймеров удовлетворяют предохранителю g; после совершения этого перехода таймеры из множества Y сбросятся в ноль. Это можно формализовать следующим образом: будем считать, что выполняется $(s, v) \xrightarrow{a} (s', v')$, если существует такой переход $e = (s, g, a, Y, s') \in R$, что $v \models g$, $v' = v[Y]$ и $v' \models Inv(s')$. Если нам важно, что был задействован именно переход e, то будем использовать запись $(s, v) \xrightarrow{a,e} (s', v')$. Будем считать, что функция $l(e)$ возвращает пометку перехода e, т. е. $l(e) = a$, если $e = (s, g, a, Y, s')$.

Автомат может оставаться в дискретном состоянии в течении некоторого времени при условии, что инвариант этого дискретного состояния не будет нарушен. Это можно формализовать следующим образом: для некоторого числа $\delta \in \mathbb{R}_{\geq 0}$ справедливо $(s, v) \xrightarrow{\delta} (s, v')$, если $v' = v + \delta$ и $v' \models Inv(s)$.

Вычислением временного автомата будем называть такую конечную или бесконечную последовательность $(q_1, \delta_1, q_2, a_1, q_3, \delta_2, q_4, a_2, q_5, \dots)$, что для всех $i \geq 1$ выполняется $q_i \in Q$, $a_i \in \Sigma$, $\delta_i \in \mathbb{R}_{\geq 0}$, $q_{2i-1} \xrightarrow{\delta_i} q_{2i}$ и $q_{2i} \xrightarrow{a_i} q_{2i+1}$. Будем также использовать запись $q_1 \xrightarrow{\delta_1} q_2 \xrightarrow{a_1} q_3 \to \dots$ для обозначения вычисления.

В введённом понятии вычисления дискретные переходы чередуются с задержками по времени. Можно предложить и другое определение, в котором не будет накладываться ограничений на относительный порядок этих двух типов переходов. Однако, легко видеть, что, если в произвольном вычислении заменить последовательные задержки по времени $q_1 \xrightarrow{\delta_1} q_2$ и $q_2 \xrightarrow{\delta_2} q_3$ на одну $q_1 \xrightarrow{\delta_1+\delta_2} q_3$, и вставить между последовательными дискретными переходами

$q_1 \xrightarrow{a_1} q_2$ и $q_2 \xrightarrow{a_2} q_3$ нулевую задержку по времени $q_2 \xrightarrow{0} q_2$, то мы придём к вычислению, удовлетворяющему используемому нами определению.

Длительностью конечного вычисления $(q_1, \delta_1, q_2, a_1, q_3, \ldots, q_{2n+1})$ назовём значение суммы $\sum_{i=1}^{n} \delta_i$, множеством видимых действий этого вычисления назовём $\{a_1, \ldots, a_n\} \setminus \{\tau\}$, и последним действием — a_n.

3.1.3. Временной игровой автомат

Временные автоматы могут применяться для решения задачи синтеза контроллера (устройства управления), т. е. построения такого алгоритма его работы, что поведение объекта управления будет корректным независимо от действий среды. Для описания моделей таких систем необходимо выделить среди множества переходов автомата те переходы, которые не контролируются устройством управления.

Определение 3.2 ([71]). Временной игровой автомат (timed game automata, TGA) — это временной автомат, в котором множество переходов R разбито на множество контролируемых (R^c) и множество неконтролируемых (R^u) переходов.

Будем использовать нижние индексы c и u для обозначения контролируемых и неконтролируемых переходов, т. е. переход $q \xrightarrow{a}_c q'$ контролируем, а переход $q \xrightarrow{a}_u q'$ неконтролируем.

Будем считать что неконтролируемые и контролируемые переходы никогда не помечены одной и той же видимой пометкой, т. е. не существует таких переходов $(s_1, g_1, a_1, Y_1, s_1') \in R^c$ и $(s_2, g_2, a_2, Y_2, s_2') \in R^u$, что $a_1 = a_2$ и $a_1 \neq \tau$.

Стратегия

Определение 3.3 ([71]). Стратегия контроллера (соответственно, среды) в TGA $\langle S, s_0, \Sigma, X, R, Inv\rangle$ — это функция f, определённая на множестве конечных вычислений, и возвращающая некоторый элемент множества $(\mathbb{R}_{\geq 0} \times R^c)$ (соответственно, $(\mathbb{R}_{\geq 0} \times R^u)$).

Определение 3.4. Будем называть стратегию f стратегией с нулевой памятью, если для любых двух вычислений π_1 и π_2, у которых одинаковое последнее состояние, справедливо $f(\pi_1) = f(\pi_2)$.

Поскольку значение стратегии с нулевой памятью зависит только от последнего состояния вычисления, то её можно считать определённой на множестве состояний Q.

В дальнейшем, если не оговорено иначе, под стратегией будем понимать стратегию с нулевой памятью.

Значение функции $f(q)$ определяет действия игрока в состоянии q. Если $f(q) = (\delta, e)$, то это означает, что игрок планирует оставаться в текущем дискретном состоянии на протяжении периода времени δ, а затем совершить переход e.

Любую стратегию f можно задать парой функций δ и e таких, что для любого q выполняется $f(q) = (\delta(q), e(q))$.

Разумно потребовать, что если согласно стратегии необходимо оставаться в текущем дискретном состоянии δ единиц времени, а затем совершить переход e, то во-первых, при ожидании не нарушится инвариант текущего дискретного состояния, а во-вторых, стратегия “не поменяет своего решения” во время задержки. Таким образом, будем рассматривать только такие стратегии (δ, e), что для любого состояния $q \in Q$, для любого $\delta' \leq \delta(q)$ существует переход $q \xrightarrow{\delta'} q'$ в такое состояние q', что $f(q') = (\delta(q) - \delta', e(\pi))$.

Определение 3.5. Пусть дан TGA $\langle S, l_0, \Sigma, X, R, Inv\rangle$, и пусть (δ^c, e^c) и (δ^u, e^u) — некоторые стратегии контроллера и среды соответственно. Вычисление $(q_1, \delta_1, q_2, a_1, q_3, \dots)$ будем называть порождённым стратегиями (δ^c, e^c) и (δ^u, e^u), если для любого $i \in \mathbb{N}$, из $e_i^c = e^c(q_{2i-1})$, $e_i^u = e^u(q_{2i-1})$ и $\delta_i^u = \delta^u(q_{2i-1})$, $\delta_i^c = \delta^c(q_{2i-1})$ следует:

- если $\delta_i^u \leq \delta_i^c$, то $q_{2i-1} \xrightarrow{\delta_i^u} q_{2i} \xrightarrow{e_i^u} q_{2i+1}$ и $a_i = l(e_i^u)$
- иначе $q_{2i-1} \xrightarrow{\delta_i^c} q_{2i} \xrightarrow{e_i^c} q_{2i+1}$ и $a_i = l(e_i^c)$

Аналогично определяется вычисление, порождённое стратегиями с ненулевой памятью.

Отметим, что по определению 3.5, если стратегии контроллера и среды требуют совершить некоторый переход одновременно (т. е. $\delta_c(q) = \delta_u(q)$), то переход совершает среда.

Пусть f^c — некоторая стратегия контроллера в TGA M. Обозначим $M[f^c, q]$ множество всех вычислений в M, начинающихся в состоянии q и порождённых f^c и некоторой стратегией среды f^u.

Задача верификации состоит в том, чтобы проверить, верно ли, что существует некоторая стратегия f^c, для которой все вычисления из $M[f^c, q]$ удовлетворяют заданной спецификаций. Для этого необходимо определить формальный язык задания спецификаций, что будет сделано в следующем разделе.

3.1.4. Логика ATCTL

Для задания спецификаций будем использоваться логику ATCTL, которая является одним из универсальных фрагментов логики Timed CTL [4].

Определение 3.6. Логика ATCTL состоит из всех формул вида $A\sigma_1 U_t \sigma_2$ и $A\sigma_1 W_t \sigma_2$, где $t \in \mathbb{Z}_{\geq 0} \cup \{+\infty\}$, а σ_1 и σ_2 — некоторые множества видимых действий.

Для того, чтобы определить выполнимость формул логики ATCTL, определим сначала выполнимость их подформул вида $\sigma_1 U_t \sigma_2$ и $\sigma_1 W_t \sigma_2$.

Будем говорить, что вычисление π временного автомата M удовлетворяет формуле $\sigma_1 U_t \sigma_2$, если существует такой префикс этого вычисления π', что выполняются следующие условия:

1. все видимые действия π' лежат в множестве σ_1, и
2. последнее действие π' лежит в σ_2, и
3. длительность π' не превышает t.

Вычисление π временного автомата M удовлетворяет формуле $\sigma_1 W_t \sigma_2$, если это вычисление удовлетворяет формуле $\sigma_1 U_t \sigma_2$, или все видимые действия π принадлежат множеству σ_1.

Будем говорить, что формула $A\varphi$ логики ATCTL выполняется на TGA M в состоянии q (обозначается $M, q \models A\phi$), если существует такая стратегия контроллера f_c, что все вычисления из множества $M[f^c, q]$ удовлетворяют формуле φ. Если формула $A\phi$ выполняется в *начальном* состоянии M, то будем обозначать это записью $M \models A\varphi$. Стратегию f^c, использование которой контроллером приводит к выполнимости формулы $A\varphi$ на M, будем называть выигрышной в игре $(M, A\varphi)$.

В работе [71] показано, что классы стратегий с нулевой и с ненулевой памятью имеют одинаковую выразительную мощность в случае, когда спецификации задаются формулами ATCTL. Это означает, что для любого TGA и для любой формулы ATCTL справедливо: в игре $(M, A\varphi)$ у контроллера

существует выигрышная стратегия с нулевой памятью тогда и только тогда, когда в этой игре у него существует стратегия с ненулевой памятью. В общем случае это не так, т. е. можно привести такое свойство φ (невыразимое формулой ATCTL) и такой TGA M, что в игре $(M, A\varphi)$ у контроллера существует выигрышная стратегия с ненулевой памятью, но не существует выигрышной стратегии с нулевой памятью.

Для проверки выполнимости формул ATCTL на временных игровых автоматов может использоваться средство UPPAAL Tiga[103], в котором реализован алгоритм, описанный в работе [41].

3.1.5. Постановка задачи

Будем говорить, что бинарное отношение $\preceq$ на множестве TGA сохраняет выполнимость формул логики L, если для любой формулы $\varphi \in L$ и любых двух TGA M_1 и M_2 таких, что $M_1 \preceq M_2$, из $M_2 \models \varphi$ следует $M_1 \models \varphi$.

В данной работе исследуется задача нахождения отношения $\preceq$, сохраняющего выполнимость формул логики ATCTL.

Данным свойством, несомненно, обладает отношение изоморфизма, т. е. отношение, в котором находятся те и только те пары TGA M_1 и M_2, что M_1 изоморфен M_2 с учётом переименования переменных-таймеров. Однако нам хотелось бы, чтобы искомое отношение $\preceq$ позволяло сравнивать существенно различные по устройству и размеру модели.

Среди отношений, сохраняющих выполнимость формул логики ATCTL, наибольший интерес представляют отношения, для которых существуют эффективные алгоритмы их проверки. Будем искать именно такое отношение.

3.2. Обзор существующих подходов

3.2.1. Языковое включение

Поскольку в формулах логики ATCTL не допускается использовать вложенные кванторы по вычислениям, то эта логика описывает свойства линейного времени. Для неигровых автоматов самым общим отношением, сохраняющим свойства линейного времени, является отношение языкового включения. Это отношение определяется следующим образом: два автомата M_1 и M_2 находятся в отношении языкового включения, если каждое слово, принимаемое автоматом M_2, принимается и автоматом M_1.

Обобщим отношение языкового включения на случай игровых временных автоматов, как это было сделано в работе [3]. Назовём видимой проекцией вычисления $q_0 \xrightarrow{\delta_1} q_1 \xrightarrow{a_1} q_2 \xrightarrow{\delta_2} q_3 \xrightarrow{a_2} q_4 \dots$ последовательность $(\delta_1, a_1, \delta_2, a_2, \dots)$. Будем говорить, что два игровых автомата M_1 и M_2 находятся в отношении языкового включения, если для любой стратегии контроллера f_2^c в M_2 существует такая стратегия контроллера f_1^c в M_1, что для любого вычисления $\pi_2 \in M_2[f_2^c, q_0]$ существует такое вычисление $\pi_1 \in M_1[f_1^c, q_0]$, что видимые проекции вычислений π_1 и π_2 совпадают. Легко видеть, что отношение языкового включения сохраняет выполнимость формул логики ATCTL. К сожалению, задача проверки языкового включения временных неигровых автоматов является алгоритмически неразрешимой [5]. Поскольку временные игровые автоматы являются частным случаем временных неигровых автоматов, то и задача проверки языкового включения игровых временных автоматов является алгоритмически неразрешимой.

3.2.2. Отношения симуляции

Отношения симуляции, являются более строгими, чем отношение языкового включения, поскольку сохраняют не только линейные свойства, но и структуру дерева вычислений в недетерменированных вычислительных системах. Для всех известных нам формализмов задания моделей задача проверки симуляции является алгоритмически более простой, чем задача проверки языкового включения. Существует большое разнообразие отношений симуляций.

Рассмотрим, какими качествами должно обладать отношение симуляции, сохраняющее выполнимость формул логики ATCTL на временных игровых автоматах. Во-первых, оно должно принадлежать к классу *временных* симуляций, поскольку оно задано над множеством временных автоматов. С другой стороны оно должно принадлежать к классу так называемых *игровых* (альтернирующих) симуляций. Поскольку в указанной логике ATCTL нет возможности задавать свойства, выполнимые в *следующем* состоянии вычисления (в частности нет оператора X), требуемое определение симуляции следует искать среди класса *ослабленных* симуляций, в определении которых видимые и невидимые действия трактуются по-разному.

Рассмотрим существующие определения временных, альтернирующих и ослабленных симуляций.

Временные симуляции

Разрешимость (би)симуляции между временными автоматами была доказана в работе [22] при помощи построения декартова произведения временных зон. Этот алгоритм был реализован в средстве Epsilon[23]. В работе [109] был предложен более эффективный подход для нахождения (слабой)

временной бисимуляции, который был основан на операциях над временными зонами (а не декартовыми произведениями), но он не был реализован на практике.

В работе [58] для проверки временной симуляции между полностью наблюдаемыми и детерменированными временными автоматами M_1 и M_2 предлагается построить автомат M', каждое выполнение которого соответствует параллельному выполнению M_1 и M_2. Тогда задача проверки симуляции сводится к поиску такого достижимого состояния M', из которого нельзя продолжить параллельное выполнение M_1 и M_2.

Временная симуляция сохраняет выполнимость формул логики TCTL [4]. Поскольку логика ATCTL является фрагментом TCTL, то временная симуляция сохраняет и выполнимость формул логики ATCTL на временных неигровых автоматах.

Альтернирующие симуляции

Альтернирующие автоматы Бюхи (их также называют игровыми автоматами) используются как вспомогательный инструмент во многих алгоритмах верификации моделей программ [107].

Кроме того, игровые автоматы могут использоваться и как формализм задания моделей мультиагентных систем [68, 88]. В этом случае для описания спецификаций удобно использовать логику альтернирующего времени ATL [6].

В работе [3] введено понятие игровой (или альтернирующей) симуляции, сохраняющей выполнимость формул универсального фрагмента логики ATL. Также в работе [3] приводится теоретико-игровой алгоритм проверки такой симуляции.

Если не рассматривать временные ограничения, то логика ATCTL явля-

ется фрагментом логики ATL, поэтому альтернирующая симуляция сохраняет выполнимость формул логики ATCTL на (невременных) игровых автоматах.

Ослабленные симуляции

Как было указано в разделе 2.1.3, ослабленными отношеними симуляции называются отношения, в которых, в отличие от строгой симуляции, видимые и невидимые действия учитываются по-разному. Обзор различных видов ослабленных отношений дан в работе [105]. Все приведённые в данном обзоре симуляции имеют эффективные алгоритмы проверки.

3.2.3. Вывод

На основании приведённого выше обзора можно сделать вывод о том, что для определения искомого отношения, сохраняющего выполнимость формул логики ATCTL, можно использовать "комбинацию" определений временной, игровой и ослабленной отношений симуляции. Поскольку задачи проверки этих отношений являются разрешимыми, то можно предположить, что разрешимой будет и задача проверки выполнимости искомого отношения. Это предположение будет обосновано ниже в разделе 3.4.

3.3. Определение слабой альтернирующей временной симуляции

Пусть M_1 и M_2 — два TGA, где $M_i = \langle S_i, s_{i0}, X_i, \Sigma_i, R_i, Inv_i \rangle$, и $Q_i = S_i \times \mathbb{R}_{\geq 0}^{X_i}$ — множества состояний этих автоматов ($i = 1, 2$). Наша задача — дать такое определение отношения симуляции $\preceq$, что если справедливо

$M_1 \preceq M_2$, то выполнимость формул логики ATCTL на M_2 неизбежно влечёт их выполнимость и на M_1.

Определим такое отношение $H \subseteq Q_1 \times Q_2$ между парами *состояний* M_1 и M_2, что если $(q_1, q_2) \in H$, то все формулы ATCTL, выполнимые в состоянии q_2, выполнимы и в состоянии q_1. Тогда выполнимость отношения $M_1 \preceq M_2$ будет эквивалентна тому, что начальные состояния TGA M_1 и M_2 находятся в отношении H.

Дадим определения требуемого отношения симуляции при помощи нескольких рекурсивных правил, которым должны удовлетворять пары состояний (q_1, q_2), находящихся в отношении H.

3.3.1. Видимые дискретные переходы

Представим, что в TGA M_1 и M_2 недопустимы задержки по времени, т. е. недопустимы переходы вида $q_1 \xrightarrow{\delta} q_2$, где $\delta > 0$. Такому ограничению будут удовлетворять, в частности, такие автоматы, в которых все инварианты состояний имеют вид $(x = 0)$, где x — некоторый таймер из $X_1 \cap X_2$.

Тогда в M_1 и M_2 разрешены только дискретные переходы, и поведение этих автоматов полностью описывается поведением схожих по структуре игровых (невременных) автоматов. Отношение симуляции между игровыми автоматами называется альтернирующей симуляцией и рассматривалось в работе [3].

В определении альтернирующей симуляции учитывается тот факт, что добавление контролируемых переходов может увеличить множество свойств, выполнимых (контролируемых) на модели, а добавление неконтролируемых переходов — уменьшить это множество. Поэтому контролируемым переходам в M_2 должны соответствовать контролируемые переходы в M_1, а неконтролируемым переходам в M_1 — неконтролируемые переходы в M_2.

Пусть $(q_1, q_2) \in H$. Потребуем, что каждому контролируемому видимому переходу $q_2 \xrightarrow{a}_c q_2'$ в TGA M_2 должен соответствовать такой контролируемый переход $q_1 \xrightarrow{a}_c q_1'$ в TGA M_1, что выполняется $(q_1', q_2') \in H$. Кроме того, каждому неконтролируемому переходу $q_1 \xrightarrow{a}_u q_1'$ в TGA M_1 должен соответствовать такой неконтролируемый переход $q_2 \xrightarrow{a}_u q_2'$ в M_2, что выполняется $(q_1', q_2') \in H$.

3.3.2. Невидимые дискретные переходы

Пусть $(q_1, q_2) \in H$. Как и в определении слабой симуляции [105], мы не будем требовать, что каждому невидимому переходу в одной из моделей должен соответствовать невидимый переход в другой модели. Однако потребуем, что если существует невидимый переход $q_2 \xrightarrow{\tau}_c q_2'$ в TGA M_2, то пара (q_1, q_2') должна находится в отношении симуляции H. Аналогично, если в M_1 есть невидимый переход $q_1 \xrightarrow{\tau}_u q_1'$, то должно выполняться $(q_1', q_2) \in H$.

Кроме того, если в моделях присутствуют невидимые переходы, то можно ослабить требования, приведённые в разделе 3.3.1. Так, каждому неконтролируемому видимому переходу $q_1 \xrightarrow{a}_u q_1'$ в TGA M_1 может соответствовать некоторая *последовательность* переходов $q_2 \xrightarrow{\tau}_u q_{2,1}' \xrightarrow{\tau}_u \ldots \xrightarrow{\tau}_u q_{2,n}' \xrightarrow{a}_u q_2'$, такая, что $(q_1', q_2') \in H$.

Аналогично, каждому контролируемому переходу $q_2 \xrightarrow{a}_c q_2'$ в TGA M_2 может соответствовать некоторая последовательность переходов $q_1 \xrightarrow{\tau}_c q_{1,1}' \xrightarrow{\tau}_c \ldots \xrightarrow{\tau}_c q_{1,n}' \xrightarrow{a}_u q_1'$, такая, что $(q_1', q_2') \in H$. Поскольку среда имеет приоритет, если стратегии и среды и контроллер требуют совершить некоторый переход одновременно, то в промежуточном состоянии $q_{1,i}'$ среда может помешать контроллеру сделать переход в состояние $q_{1,i+1}'$, поэтому дополнительно потребуем, что должно выполнятся $(q_{1,i}', q_2) \in H$ для каждого $i = 1..n$.

3.3.3. Временные задержки

Представим, что в TGA M_2 существует задержка $q_2 \xrightarrow{\delta} q_2'$, и что для любого промежуточного состояния q_2'' (т. е. $q_2 \xrightarrow{\delta'} q_2''$, где $0 \leq \delta' < \delta$), из q_2'' не разрешён ни один дискретный переход. Тогда, если формула $\sigma_1 U_t \sigma_2$ (или $\sigma_1 W_t \sigma_2$) выполнима в состоянии q_2', то в состоянии q_2 выполнима формула $\sigma_1 U_{t+\delta} \sigma_2$ (соответственно, $\sigma_1 W_{t+\delta} \sigma_2$). Для того, чтобы та же формула выполнялась и в TGA M_1 в состоянии q_1, достаточно потребовать, чтобы в M_1 существовал переход $q_1 \xrightarrow{\delta} q_1'$, и выполнялось $(q_1', q_2') \in H$.

3.3.4. Определение искомого отношения

Определение 3.7. Пусть даны два TGA M_1 и M_2 с множествами состояний Q_1 и Q_2, где $M_i = \langle S_i, s_{i0}, X_i, \Sigma_i, R_i, Inv_i \rangle$. Отношение $H \subseteq Q_1 \times Q_2$ называется отношением слабой альтернирующей временной (timed weak alternating, twa) симуляции между M_1 и M_2, если для любой пары $(q_1, q_2) \in H$ и для любых $q_1' \in Q_1$, $q_2' \in Q_2$, $\delta \in \mathbb{R}_{\geq 0}$, $a \in \Sigma \setminus \{\tau\}$ выполняется:

1. если $(q_2 \xrightarrow{a}_c q_2')$, то существует такое $n \geq 1$ и такая последовательность состояний $q_{1,1}', q_{1,2}', \ldots, q_{1,n}'$ автомата M_1, что $q_{1,1}' = q_1$, $q_{1,n-1}' \xrightarrow{a}_c q_{1,n}'$, $(q_{1,n}', q_2') \in H$ и для всех $i = 2..n-1$ выполняется $(q_{1,i}', q_2) \in H$ и $q_{1,i-1}' \xrightarrow{\tau}_c q_{1,i}'$

2. если $(q_1 \xrightarrow{a}_u q_1')$, то существует такое $n \geq 1$ и такая последовательность состояний $q_{2,1}', q_{2,2}', \ldots, q_{2,n}'$ автомата M_2, что $q_{2,1}' = q_2$, $q_{1,n-1}' \xrightarrow{a}_u q_{1,n}'$, $(q_1', q_{2,n}') \in H$ и для всех $i = 2..n-1$ выполняется $q_{1,i-1}' \xrightarrow{\tau}_u q_{1,i}'$

3. если $q_2 \xrightarrow{\tau}_c q_2'$, то $(q_1, q_2') \in H$

4. если $q_1 \xrightarrow{\tau}_u q_1'$, то $(q_1', q_2) \in H$

5. если $q_2 \xrightarrow{\delta} q_2'$, то существует такое $q_1'' \in Q_1$, что $q_1 \xrightarrow{\delta} q_1''$ и $(q_1'', q_2') \in H$

Отношение twa-симуляции замкнуто по объединению, т. е. если H_1 и H_2 — отношения twa-симуляции, то $H_1 \cup H_2$ — тоже отношение twa-симуляции. Поэтому существует максимальное по включению отношение twa-симуляции, то есть такое отношение H_{max}, что для любого другого отношения twa-симуляции выполняется $H \subseteq H_{max}$.

Будем говорить, что TGA M_1 и M_2 находятся в отношении twa-симуляции (обозначается $M_1 \preceq_{twa} M_2$), если существует такое отношение twa-симуляции $H \subseteq Q_1 \times Q_2$ между M_1 и M_2, что $(q_{10}, q_{20}) \in H$, где q_{10} и q_{20} — начальные состояния M_1 и M_2 соответственно.

Будем говорить, что в M существует цикл по невидимым переходам, если в M существует такое достижимое состояние q и такая последовательность состояний $q_1, q_2, \ldots q_n$, что $q = q_1 = q_n$, и для любого $i = 1..n-1$ выполняется $q_i \xrightarrow{\tau} q_{i+1}$. На практике отсутствие циклов по невидимым переходам означает отсутствие так называемых “живых блокировок” (livelocks), т. е. бесконечного поведения, в котором с точки зрения наблюдателя ничего не происходит.

Теорема 3.1. *Пусть даны два TGA M_1 и M_2, в которых нет циклов по невидимым переходам, и такая формула φ логики ATCTL, что справедливо $M_1 \preceq_{twa} M_2$ и $M_2 \models \varphi$. Тогда справедливо и $M_1 \models \varphi$.*

Доказательство. Пусть $M_i = \langle S_i, s_{i0}, X_i, \Sigma_i, R_i, Inv_i \rangle$, Q_i — множество состояний M_i, q_{i0} — его начальное состояние, где $i = 1, 2$. Предположим, что f_2^c — выигрышная стратегия контроллера в игре (M_2, φ). Покажем, как построить выигрышную стратегию *с ненулевой памятью* контроллера f_1^c в игре (M_1, φ), используя стратегию f_2^c и отношение H.

Для начала рассмотрим случай, когда M_1 и M_2 не содержат невидимых переходов. Тогда для любой пары $(q_1, q_2) \in H$ и для любых $a \in \Sigma$ и $\delta \in \mathbb{R}_{\geq 0}$ выполняется:

- если $q_2 \xrightarrow{a}_c q_2'$, то существует такое состояние $q_1' \in Q_1$, что справедливо $q_1 \xrightarrow{a}_c q_1'$ и $(q_1', q_2') \in H$,

- если $q_1 \xrightarrow{a}_u q_1'$, то существует такое состояние $q_2' \in Q_2$, что справедливо $q_2 \xrightarrow{a}_u q_2'$ и $(q_1', q_2') \in H$,

- если $q_2 \xrightarrow{\delta} q_2'$, то существует такое состояние $q_1' \in Q_1$, что справедливо $q_1 \xrightarrow{\delta} q_1'$ и $(q_1', q_2') \in H$.

Напомним, что стратегия контроллера с ненулевой памятью — это функция, заданная на множестве вычислений. Будем определять стратегию контроллера f_1^c индукцией по длине вычисления π_1 автомата M_1. Допустим, уже построен префикс вычисления π_1. Зафиксируем значение стратегии контроллера $f_1^c(\pi_1)$, а затем выберем произвольное допустимое значение стратегии среды $f_1^u(\pi_1)$ и продолжим префикс π_1, используя значения $f_1^c(\pi_1)$ и $f_1^u(\pi_1)$. Одновременно будем строить некоторое вычисление $\pi_2 \in M_2[f_2^c, q_{20}]$ так, что если q_1 и q_2 — последние состояния уже построенных префиксов вычислений π_1 и π_2, то $(q_1, q_2) \in H$. По определению симуляции $(q_{10}, q_{20}) \in H$, таким образом, это условие будет выполняться и в самом начале, когда вычисления π_1 и π_2 не содержат ни одного перехода.

Пусть π_1 и π_2 — уже построенные вычисления, $q_1 = (s_1, v_1)$ и $q_2 = (s_2, v_2)$ — последние состояния этих вычислений, и пусть $f_2^c(q_2) = (\delta, e_2^c)$, где $l(e_2^c) = a$. Поскольку существует переход $q_2 \xrightarrow{\delta} q_2'$, то по определению симуляции существует переход $q_1 \xrightarrow{\delta} q_1'$, и $(q_1', q_2') \in H$. Поскольку существует переход $q_2' \xrightarrow{e_2^c}_c q_2''$, то по определению симуляции существует и так же помеченный переход $q_1' \xrightarrow{e_1^c}_c q_1''$ (где $l(e_1^c) = a$). Поэтому значение стратегии $f_1^c(\pi_1)$ можно положить равным (δ, e_1^c).

Выберем теперь произвольное допустимое значение стратегии среды $f_1^u(\pi_1) = (\delta', e_1^u)$, где $l(e_1^u) = a'$.

Если $\delta' > \delta$, то вычисление π_1 продолжится задержкой $q_1 \xrightarrow{\delta} q_1'$ и переходом контроллера $q_1' \xrightarrow{e_1^c} q_1''$. Вычисление π_2 в этом случае продолжим переходами $q_2 \xrightarrow{\delta} q_2'$ и $q_2' \xrightarrow{e_2^c} q_2''$.

Если $\delta' \leq \delta$, то вычисление π_1 продолжится задержкой $q_1 \xrightarrow{\delta'}_u q_1'$ и переходом среды $q_1' \xrightarrow{e_1^u}_u q_1''$. Как было показано выше в разделе 3.1.2, инварианты состояний задаются выпуклыми функциями. Поэтому в M_2 допустима задержка $q_2 \xrightarrow{\delta'} q_2'$, и по определению симуляции выполняется $(q_1', q_2') \in H$. Следовательно по определению симуляции существует такие $e_2^u \in R_2^u$ и $q_2'' \in Q_2$, что справедливо $l(e_2^u) = a'$, $q_2' \xrightarrow{e_2^u}_u q_2''$ и $(q_1'', q_2'') \in H$. Продолжим вычисление π_2 переходами $q_2 \xrightarrow{e_2^u} q_2'$ и $q_2' \xrightarrow{\delta} q_2''$.

В обоих описанных выше случаях выполняется $(q_1'', q_2'') \in H$, где q_1'' и q_2'' — последние состояния построенных префиксов вычислений π_1 и π_2.

Назовём проекцией вычисления $\pi = (q_0, \delta_1, q_1, a_1, q_2, \delta_2, \dots)$ последовательность $(\delta_1, a_1, \delta_2, \dots)$. По стратегии контроллера f_2^c в TGA M_2 и отношению H мы построили такую стратегию *с ненулевой памятью* f_1^c контроллера в M_1, что для любого вычисления $\pi_1 \in M_1[f_1^c, q_{10}]$ существует некоторое такое вычисление $\pi_2 \in M_2[f_2^c, q_{20}]$, что проекции вычислений π_1 и π_2 совпадают. Пусть $\varphi = A\psi$, где ψ — это либо $\sigma_1 U_t \sigma_2$, либо $\sigma_1 W_t \sigma_2$. Поскольку стратегия f_2^c является выигрышной в игре (M_2, φ), то на вычислении π_2 выполняется формула ψ. Тогда формула ψ выполняется и на π_1 (поскольку проекции вычислений π_1 и π_2 совпадают). Следовательно, в силу произвольности выбора стратегии среды f_1^u, формула ψ выполняется на любом вычислении из $M_1[f_1^c, q_{10}]$, и поэтому стратегия (с ненулевой памятью) f_1^c является выигрышной в игре (M_1, φ). Поскольку для случая, когда цель игры задаётся формулой ATCTL, стратегии с нулевой памятью и с ненулевой памятью имеют одинаковую выразительную мощность [71], то в игре (M_1, φ) имеется и выигрышная стратегия с нулевой памятью. Следовательно, справедливо $M_1 \models \varphi$.

Мы рассмотрели упрощённый случай, в котором невидимые переходы отсутствовали в моделях. Перейдём теперь к рассмотрению общего случая моделей M_1 и M_2.

Опишем теперь, как обобщить приведённое выше доказательство на случай, когда в M_1 и M_2 есть невидимые переходы. Допустим, что мы построили вычисления π_1 и π_2, q_1 и q_2 — последние состояния этих вычислений и $(q_1, q_2) \in H$.

Рассмотрим сначала случай, когда в M_2 есть контролируемые невидимые переходы. Предположим, что $f_2^c(q_2) = (\delta_1, e_{2,1}^c)$, где $l(e_{2,1}^c) = \tau$. Пусть $q_2 \xrightarrow{\delta_1} q_{2,1} \xrightarrow{e_{2,1}^c}_c q_{2,2}$ и $f_2^c(q_{2,2}) = (\delta_2, e_{2,2}^c)$, где $l(e_{2,2}^c) = a$. Предположим, что $a \neq \tau$. Пусть $q_{2,2} \xrightarrow{\delta_2} q_{2,3} \xrightarrow{e_{2,2}^c} q_{2,4}$. Из пунктов 3 и 5 определения симуляции видно, что в M_1 существует задержка $q_1 \xrightarrow{\delta_1+\delta_2} q_{1,2}$, и существует такой переход $q_{1,2} \xrightarrow{e_1^c} q_{1,3}$, что справедливо $l(e_1^c) = a$ и $(q_{1,3}, q_{2,4}) \in H$. Выберем в качестве значения стратегии $f_1^c(\pi_1) = (\delta_1 + \delta_2, e_1^c)$.

Если $a = \tau$, то надо применять стратегию контроллера в M_2 до тех пор, пока не встретится видимый переход. Рано или поздно такой видимый переход встретится, поскольку по условию теоремы в M_2 нет циклов по невидимым переходам.

Аналогично рассматривается случай, когда в M_1 присутствуют неконтролируемые невидимые переходы.

Рассмотрим случай, когда в M_1 есть контролируемые невидимые переходы. Пусть $f_2^c(q_2) = (\delta, e_2^c)$, где $l(e_2^c) = a$. Тогда последовательности переходов $q_2 \xrightarrow{\delta} q_2' \xrightarrow{e_2^c}_c q_2''$ может соответствовать последовательность переходов $q_1 \xrightarrow{\delta} q_{1,1} \xrightarrow{\tau}_c q_{1,2} \xrightarrow{\tau}_c \ldots \xrightarrow{\tau}_c q_{1,n} \xrightarrow{e_2^c}_c q_1''$, где $l(e_2^c) = a$, $n \geq 1$, $(q_{1,i}, q_2') \in H$ (при $i = 1..n$) и $(q_1'', q_2'') \in H$. Среда имеет приоритет над контроллером при выборе следующего состояния вычисления, поэтому данная последовательность переходов может прерваться переходом $q_{1,i} \xrightarrow{a}_u q_1'''$. По определению симуля-

ции указанному переходу будет соответствовать такой переход $q_2' \xrightarrow{a}_u q_2'''$, что $(q_1''', q_2''') \in H$.

Аналогично рассматривается случай, когда в M_2 присутствуют неконтролируемые невидимые переходы.

Назовём *видимой* проекцией вычисления $\pi = (q_0, \delta_1, q_1, a_1, q_2, \delta_2, \dots)$ последовательность, получаемую из $(\delta_1, a_1, \delta_2, \dots)$ удалением τ-пометок и заменой нескольких подряд идущих длительностей задержек их суммой. Легко видеть, что построенная описанным выше образом стратегия f_1^c обладает следующим свойством: для любого вычисления $\pi_1 \in M_1[f_1^c, q_{10}]$ существует такое вычисление $\pi_2 \in M_2[f_2^c, q_{20}]$, что *видимые* проекции π_1 и π_2 совпадают. Повторив рассуждения из конца первой части доказательства, мы получаем, что $M_1 \models \varphi$.

Таким образом, теорема 3.1 доказана для произвольных моделей M_1 и M_2.

□

По ходу доказательства теоремы 3.1 было установлено также одно важное свойство, который мы используем позже в разделе 3.6 при описании экспериментальных исследований. Предположим, что автоматы M_1 и M_2 детерминированы по действиям, т. е. для любого состояния q одного из автоматов и любого действия a существует не более одного состояния q' такого, что $q \xrightarrow{a} q'$. Тогда для любой последовательности $(\delta_1, a_1, \delta_2, a_2, \dots)$ существует не более одного вычисления π_i автомата M_i такого, что проекция π_i равна $(\delta_1, a_1, \delta_2, a_2, \dots)$ (т. е. $\pi_i = q_{i0} \xrightarrow{\delta_1} q_{i1} \xrightarrow{a_2} q_{i2} \xrightarrow{\delta_2} \dots$ для некоторых состояний $q_{i1}, q_{i2}, \dots$). Следовательно, можно задавать стратегии в детерменированных временных автоматах M_1 и M_2 как функции, определённые на множестве проекций вычислений, и возвращающие пары (δ, a), где $\delta \in \mathbb{R}_{\geq 0}$, a — некоторое действие. Будем называть стратегии, заданные таким образом, ob-страте-

гиями (observation based strategies) [101]. Если значение ob-стратегии равно (δ, a), то это означает, что игрок планирует оставаться в текущем дискретном состоянии на протяжении периода времени δ, а затем совершить такой переход e, что $l(e) = a$. Тогда если в TGA M_1 и M_2 нет невидимых переходов, и стратегия f_2^c является выигрышной в игре (M_2, φ), то построенная в доказательстве теоремы 3.1 выигрышная в игре (M_1, φ) стратегия f_1^c в точности совпадает с стратегией f_2^c (если считать, что стратегии f_1^c и f_2^c определены как ob-стратегии). Поэтому справедливо следующее следствие теоремы 3.1:

Следствие 3.2. *Если TGA M_1 и M_2 детерменированы по действиям и справедливо $M_1 \preceq_{twa} M_2$, то для любой формулы φ логики ATCTL и любой ob-стратегии контроллера f^c, если f^c является выигрышной в игре (M_2, φ), то она является выигрышной и в игре (M_1, φ).*

3.4. Алгоритм проверки слабой альтернирующей временной симуляции

Мы применим теоретико-игровой подход к задаче проверки twa-симуляции. Для этого по двум TGA M_1 и M_2 мы построим такую игру $G_{\preceq_{twa}, M_1, M_2}$, что существование выигрышной стратегии в этой игре эквивалентно существованию twa-симуляции между M_1 и M_2. Игра $G_{\preceq_{twa}, M_1, M_2}$ имеет бесконечно много состояний, поэтому для её решения будет предложен символьный алгоритм.

3.4.1. Игра проверки симуляции (базовый вариант)

Пусть даны два TGA M_1 и M_2, где $M_i = \langle S_i, s_{i0}, X_i, \Sigma, R_i, Inv_i \rangle$, $i = 1, 2$, и $Q_i = S_i \times \mathbb{R}_{\geq 0}^{X_i}$ — множества состояний этих автоматов. Зададим игру

$G_{\preceq_{twa},M_1,M_2} = \langle V_S, V_D, E_S, E_D, v_0 \rangle$ для проверки существования twa-симуляции между M_1 и M_2. Как и в игре проверки невременной симуляции, в игре $G_{\preceq_{twa},M_1,M_2}$ принимают участие два игрока, Spoiler и Duplicator. Компоненты игры суть таковы:

- $V_S = Q_1 \times Q_2$ — состояния игрока Spoiler, множество допустимых ходов игрока Spoiler будет приведено при описании компонента E_S;
- $V_D = (Q_1 \times Q_2 \times \Sigma) \bigcup (Q_1 \times Q_2 \times Q_2 \times \Sigma) \bigcup (Q_1 \times Q_2 \times \mathbb{R}_{\geq 0})$ — состояния игрока Duplicator, где:
  - если игра находится в состоянии $(q_1, q_2, a)_D$, то игрок Duplicator должен найти вычисление вида $q_2 \xrightarrow{\tau}_u q_{2,i} \xrightarrow{\tau}_u \ldots \xrightarrow{\tau}_u q_{2,n} \xrightarrow{a}_u q_2'$ или $q_2 \xrightarrow{a}_u q_2'$, и совершить все переходы этого вычисления, после этого право хода возвращается игроку Spoiler,
  - если игра находится в состоянии $(q_1, q_2, q_2', a)_D$, то игрок Duplicator должен найти вычисление $q_1 \xrightarrow{\tau}_c q_{1,i} \xrightarrow{\tau}_c \ldots \xrightarrow{\tau}_c q_{1,n} \xrightarrow{a}_c q_1'$, и совершить первый переход этого вычисления; либо он может найти переход $q_1 \xrightarrow{a}_c q_1'$ и совершить переходы $q_1 \xrightarrow{a}_u q_1'$ и $q_2 \xrightarrow{a}_u q_2'$, после этого право хода возвращается игроку Spoiler,
  - если игра находится в состоянии (q_1, q_2, δ), то игрок Duplicator должен совершить $q_1 \xrightarrow{\delta} q_1'$;
- $E_S = E_S^1 \cup E_S^2 \cup E_S^3 \cup E_S^4 \cup E_S^5$ — множество переходов игрока Spoiler, где:
  - $E_S^1 = \{((q_1, q_2)_S, (q_1, q_2, q_2', a)_D) | q_2 \xrightarrow{a}_c q_2' \wedge a \in \Sigma \setminus \{\tau\}\}$, т. е. игрок Spoiler выбирает некоторый видимый контролируемый переход в M_2, и передаёт право хода игроку Duplicator,

- $E_S^2 = \{((q_1, q_2)_S, (q_1', q_2, a)_D) | q_1 \xrightarrow{a}_u q_1' \wedge a \in \Sigma \setminus \{\tau\}\}$, т. е. игрок Spoiler совершает некоторый видимый неконтролируемый переход в M_1, и передаёт право хода игроку Duplicator,
- $E_S^3 = \{((q_1, q_2)_S, (q_1, q_2')_S) | q_2 \xrightarrow{\tau}_c q_2'\}$, т. е. игрок Spoiler совершает некоторый невидимый контролируемый переход в M_2, и право хода остаётся у игрока Spoiler,
- $E_S^4 = \{((q_1, q_2)_S, (q_1', q_2)_S) | q_1 \xrightarrow{\tau}_u q_1'\}$, т. е. игрок Spoiler выбирает некоторый невидимый неконтролируемый переход в M_1, и право хода остаётся у игрока Spoiler,
- $E_S^5 = \{((q_1, q_2)_S, (q_1, q_2', \delta)_D) | q_2 \xrightarrow{\delta} q_2' \wedge \delta \in \mathbb{R}_{\geq 0}\}$, т. е. игрок Spoiler совершает δ-переход в M_2, и передаёт право хода игроку Duplicator;

• $E_D = E_D^1 \cup E_D^2 \cup E_D^3 \cup E_D^4 \cup E_D^5$ — множество переходов игрока Duplicator, где:

- $E_D^1 = \{((q_1, q_2, q_2', a)_D, (q_1', q_2')_S) | q_1 \xrightarrow{a}_c q_1' \wedge a \in \Sigma \setminus \{\tau\}\}$, т. е. игрок Duplicator нашёл контролируемый переход в M_1, имеющий пометку a, и передаёт право хода игроку Spoiler,
- $E_D^2 = \{((q_1, q_2, q_2', a)_D, (q_1', q_2)_S) | q_1 \xrightarrow{\tau}_c q_1'\}$, т. е. игрок Duplicator выбрал невидимый контролируемый переход в M_1, и передаёт право хода игроку Spoiler,
- $E_D^3 = \{((q_1, q_2, a)_D, (q_1, q_2')_S) | q_2 \xrightarrow{a}_u q_2' \wedge a \in \Sigma \setminus \{\tau\}\}$, т. е. игрок Duplicator нашёл контролируемый переход в M_1, имеющий пометку a, и передаёт право хода игроку Spoiler,
- $E_D^4 = \{((q_1, q_2, a)_D, (q_1, q_2', a)_D) | q_2 \xrightarrow{\tau}_u q_2'\}$, т. е. игрок Duplicator выбрал невидимый неконтролируемый переход в M_2, но он пока

ещё не нашёл переход, помеченный a, поэтому следующим будет ходить снова игрок Duplicator,

– $E_D^5 = \{((q_1, q_2, \delta)_D, (q_1', q_2)_S) | q_1 \xrightarrow{\delta} q_1' \wedge \delta \in \mathbb{R}_{\geq 0}\}$, т. е. игрок Duplicator совершил δ-переход в M_1;

- $v_0 = (q_{10}, q_{20})$, где $q_{i0} = (s_{i0}, \vec{0})$ — начальное состояние M_i.

Определим понятие выигрышного состояния в игре $G_{\preceq_{twa}, M_1, M_2}$ так же, как мы это сделали в разделе 2.3.1 для игр проверки симуляции между структурами Крипке.

Теорема 3.3. *Пусть в TGA M_1 и M_2 нет циклов по невидимым переходам. Пусть $H \subseteq Q_1 \times Q_2$ — максимальное отношение twa-симуляции между M_1 и M_2. Тогда для любых двух состояний $q_1 \in Q_1$ и $q_2 \in Q_2$ выполняется: $(q_1, q_2) \in H$ тогда и только тогда, когда состояние $(q_1, q_2)_S$ игры $G_{\preceq_{twa}, M_1, M_2}$ является выигрышным для игрока Duplicator.*

Доказательство. Эта теорема доказывается аналогично теоремам 2.2, 2.3. Например, выполнимость требования 1 определения 3.7 соответствует теореме 2.3, выполнимость требований 2,3,4,5 — теореме 2.2. □

Из теоремы 3.3 следует, что между M_1 и M_2 существует отношение twa-симуляциии тогда и только тогда, когда начальное состояние $G_{\preceq_{twa}, M_1, M_2}$ является выигрышным для игрока Spoiler. Однако прямое применение данного результата не позволяет построить алгоритм проверки twa-симуляции, поскольку в игре $G_{\preceq_{twa}, M_1, M_2}$ бесконечно много состояний. В разделе 3.4.3 мы опишем способ компактного символьного представления игры и алгоритм её решения.

3.4.2. Игра проверки симуляции (модифицированный вариант)

В данном разделе мы приведём игру $G^{\delta}_{\preceq_{twa},M_1,M_2}$, которая является промежуточным звеном между игрой $G_{\preceq_{twa},M_1,M_2}$, строение которой непосредственно вытекает из определения twa-симуляции, и конечной символьной игрой $G^{s}_{\preceq_{twa},M_1,M_2}$, которая будет описана в разделе 3.4.3 и которую можно использовать на практике для проверки twa-симуляции. При построении игры $G^{\delta}_{\preceq_{twa},M_1,M_2}$ мы будем предполагать для простоты, что в M_1 нет контролируемых невидимых переходов. Кроме того, мы учтём тот факт, что временные автоматы детерменированы по времени.

Пусть в M_1 нет контролируемых невидимых переходов. В этом случае требование 1 определения twa-симуляции 3.7 упрощается: если $(q_1, q_2) \in H$, то каждому контролируемому видимому переходу $q_2 \xrightarrow{a}_c q_2'$ в M_2 должен соответствовать некоторый такой переход $q_1 \xrightarrow{a}_c q_1'$ в M_1, что $(q_1', q_2') \in H$. Поэтому состояния игрока Duplicator вида $(q_1, q_2, q_2', a)_D$ можно заменить на $(q_1, q_2', a)_D$, поскольку от значения второго компонента состояния (т. е. q_2) не будет зависеть множество игровых переходов. Приведённый в разделе 3.4.3 алгоритм может быть расширен и на случай, когда в M_1 есть контролируемые невидимые переходы.

Временные автоматы детерменированы по времени, т. е. для любого состояния q и любого $\delta \in \mathbb{R}_{\geq 0}$ существует не более одного такого состояния q', что $q \xrightarrow{\delta} q'$. Поэтому игрок Spoiler может сам проверить справедливость требования 5 определения симуляции 3.7, поскольку каждому переходу $q_2 \xrightarrow{\delta} q_2'$ в M_2 соответствует не более одного перехода $q_1 \xrightarrow{\delta} q_1'$ в M_1.

Используем эти особенности и определим игру $G^{\delta}_{\preceq_{twa},M_1,M_2} = \langle V_S, V_D, E_S, E_D, v_0 \rangle$:

- $V_S = Q_1 \times Q_2$,

- $V_D = Q_1 \times Q_2 \times \Sigma$,
- $E_S = E_S^1 \cup E_S^2 \cup E_S^3 \cup E_S^4$, где:
  - $E_S^1 = \{((q_1, q_2)_S, (q_1, q_2', a)_D) | q_2 \xrightarrow{a}_c q_2' \wedge a \in \Sigma \setminus \{\tau\}\}$
  - $E_S^2 = \{((q_1, q_2)_S, (q_1', q_2, a)_D) | q_1 \xrightarrow{a}_u q_1' \wedge a \in \Sigma \setminus \{\tau\}\}$
  - $E_S^3 = \{((q_1, q_2)_S, (q_1, q_2')_S) | q_2 \xrightarrow{\tau}_c q_2'\}$
  - $E_S^4 = \{((q_1, q_2)_S, (q_1', q_2)_S) | q_1 \xrightarrow{\tau}_u q_1'\}$
- $E_D = E_D^1 \cup E_D^2 \cup E_D^3$, где:
  - $E_D^1 = \{((q_1, q_2, a)_D, (q_1', q_2)_S) | q_1 \xrightarrow{a}_c q_1' \wedge a \in \Sigma \setminus \{\tau\}\}$
  - $E_D^2 = \{((q_1, q_2, a)_D, (q_1, q_2')_S) | q_2 \xrightarrow{a}_u q_2' \wedge a \in \Sigma \setminus \{\tau\}\}$
  - $E_D^3 = \{((q_1, q_2, a)_D, (q_1, q_2', a)_D) | q_2 \xrightarrow{\tau}_u q_2'\}$
- $v_0 = (q_{10}, q_{20})$, где $q_{i0} = (s_{i0}, \vec{0})$ — начальное состояние M_i

В игре $G^{\delta}_{\preceq_{twa}, M_1, M_2}$ учитывается соответствие только дискретных переходов TGA M_1 и M_2. Поэтому, если определять выигрышные состояния как наименьшую неподвижную точку оператора g, как это было сделано в разделе 2.3.1, то окажется, что в игре $G^{\delta}_{\preceq_{twa}, M_1, M_2}$ игрок Spoiler никак не сможет проверить выполнимость пункта 5 определения 3.7 отношения twa-симуляции. Для того, чтобы учесть и течение времени, зададим оператор g^{δ}, действующий на множестве оценок (напомним, что оценка — булева функция на множестве игровых состояний). Неподвижная точка оператора g^{δ} будет характеризовать множество выигрышных состояний игрока Spoiler (опреде-

ление функции $Succ$ дано в разделе 2.3.1):

$$g^\delta(m)(v) = \begin{cases} 1 & \text{если } v \in V_S \text{ и } \exists v' \in Succ(v) : m(v') = 1 \\ & \quad \text{или } v \in V_D \text{ и } |Succ(v)| = 0 \\ & \quad \text{или } v \in V_D \text{ и } \forall v' \in Succ(v) : m(v') = 1 \\ & \quad \text{или } v \in V_S,\ v = (q_1, q_2)_S, \\ & \qquad \exists \delta \in \mathbb{R}_{\geq 0},\ \exists q_2' : q_2 \xrightarrow{\delta} q_2' \text{ и } \nexists q_1' : q_1 \xrightarrow{\delta} q_1' \\ & \quad \text{или } v =\in V_S,\ v = (q_1, q_2)_S, \\ & \qquad \exists \delta \in \mathbb{R}_{\geq 0},\ \exists q_2' : q_2 \xrightarrow{\delta} q_2',\ \exists q_1' : q_1 \xrightarrow{\delta} q_1' \text{ и } m((q_1', q_2')_S) = 1 \\ 0 & \text{иначе} \end{cases}$$

Как видно из определения оператора g^δ, состояние $v = (q_1, q_2)_S$ является выигрышным для игрока Spoiler в следующих случаях:

- оно выигрышно для игрока Spoiler в соответствии с определением, данным в разделе 2.3.1, или
- в состоянии q_2 допустима задержка $q_2 \xrightarrow{\delta} q_2'$ (для некоторого $\delta \in \mathbb{R}_{\geq 0}$), а в M_1 такая задержка недопустима, или
- существуют переходы $q_1 \xrightarrow{\delta} q_1'$ и $q_2 \xrightarrow{\delta} q_2'$, и состояние $(q_1', q_2')_S$ является выигрышным для игрока Spoiler.

Заметим, что множества состояний игрока Spoiler в играх $G_{\preceq_{twa}, M_1, M_2}$ и $G^\delta_{\preceq_{twa}, M_1, M_2}$ совпадают. Это позволяет сравнивать значения наименьших неподвижных точек операторов g и g^δ на множестве состояний игрока Spoiler. В частности, справедлива следующая теорема.

Теорема 3.4. *Пусть m и m^δ — наименьшие неподвижные точки операторов g и g^δ соответственно. Тогда для любого состояния v игрока Spoiler значения $m(v)$ и $m^\delta(v)$ совпадают.*

Доказательство. Ограничимся описанием идеи доказательства. Заметим, что замена оператора g на оператор g^δ (в котором добавлены новые условия) равносильно удалению правил δ-переходов из игры $G_{\preceq_{twa},M_1,M_2}$. В самом деле, пусть m' — оценка состояний $G_{\preceq_{twa},M_1,M_2}$, а m'^δ — соответствующая ей оценка состояний $G^\delta_{\preceq_{twa},M_1,M_2}$. Тогда легко видеть, что $g(m')$ соответствует $g^\delta(m'^\delta)$. Следовательно, значения неподвижных точек операторов g и g^δ совпадают на состояниях игрока *Spoiler*, что и требовалось доказать. □

3.4.3. Игра проверки симуляции (символьный вариант)

Поскольку игра $G^\delta_{\preceq_{twa},M_1,M_2}$ проверки twa-симуляции в общем случае имеет бесконечно много состояний, невозможно привести алгоритм решения этой игры, в котором на каждом шаге будет анализироваться лишь конечное число игровых состояний. Поэтому в данной работе для решения игры предлагается использовать символьный подход. Вместо рассмотрения отдельных состояний игрока Spoiler мы будем рассматривать *мета-состояния*, задаваемые тройками $(s_1, s_2, Z)_S$, где $s_1 \in S_1$, $s_2 \in S_2$ и Z — подмножество $\mathbb{R}^{X_1 \cup X_2}_{\geq 0}$, заданное некоторой конечной структурой данных, также называемое регионом. Каждое мета-состояние характеризует потенциально бесконечное множество состояний $\{((s_1, v_1), (s_2, v_2))_S | v_1 \otimes v_2 \in Z\}$ игры $G^\delta_{\preceq_{twa},M_1,M_2}$ (напомним, что оператор $\otimes$ был определён в разделе 3.1.2). Аналогично, мета-состояния игрока Duplicator будут иметь вид $(s_1, s_2, Z, a)_D$.

Для задания устройства символьной игры мы будем использовать следующие операции над регионами: объединение ($\cup$), пересечение ($\cap$), дополнения ($\setminus$) а также операторы up, $down$, $reset$ и $free$ определённые следующим образом: $up(Z) = \{v + \delta | v \in Z \wedge \delta \in \mathbb{R}_{\geq 0}\}$, $down(Z) = \{v | up(\{v\}) \cap Z \neq \varnothing\}$, $reset(Z, Y) = \{v[Y] | v \in Z\}$, $free(Z, Y) = \{v | \exists v' \in Z(v[Y] = v'[Y])\}$ (Y — некоторое множество таймеров). Примеры применения данных операто-

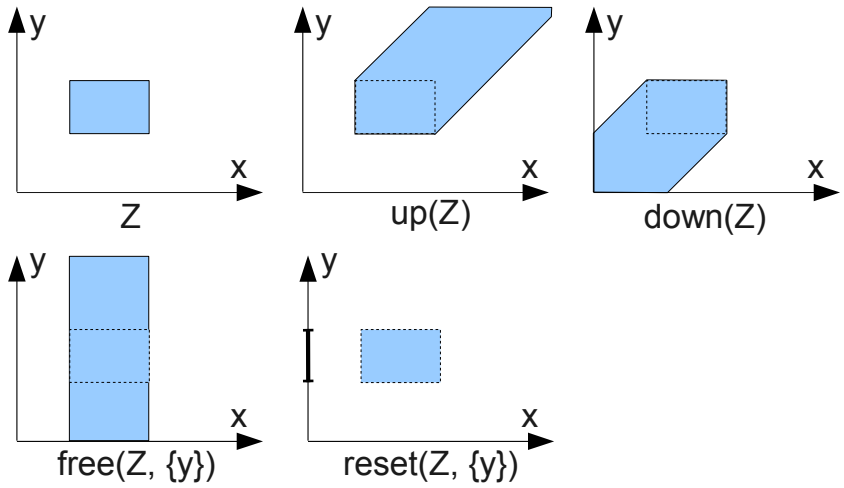

Рисунок 3.1. Операторы up, $down$, $reset$, $free$

ров изображены на рисунке 3.1. Поэтому символьный способ представления регионов должен быть замкнут относительно этих операций: если аргумент операции представим в символьном виде, то и результат также должен быть представим в символьном виде. Как было показано в работе [13], в качестве представления, удовлетворяющего этому требованию, можно использовать булевы формулы без отрицания над предикатами вида $x \sim n$ и $x - y \sim n$, где $\sim \in \{<, \leq, <, =, >, \geq\}$, $x, y \in X$ (X — множество таймеров) и $n \in \mathbb{Z}_{\geq 0}$. Будем использовать запись $\tilde{\mathcal{B}}(X)$ для обозначения такого множества формул. На практике для повышения эффективности символьных алгоритмов используется специальное представление таких формул, а именно представление в виде семейств матриц ограниченных разностей [12].

Матрицы ограниченных разностей

Пусть дано множество переменных X. Опишем устройство матриц ограниченных разностей (difference bounded matrixes, DBM) [12], которые могут использоваться для представления формул из множества $\mathcal{B}(X)$, определённого в разделе 3.1.2 (напомним, что формулы этого множества не содержат

дизъюнкций). Пусть множество X_0 получается добавлением в множество X константы 0: $X_0 = X \cup \{0\}$. Предположим, что множество X_0 перенумеровано, x_i — i-ый элемент этого множества, и $x_0 = 0$. Тогда любая формула из $\psi \in \mathcal{B}(X)$ может быть записана как коньюнкция ограничений вида $x - y \sim n$, где $x, y \in X_0$, $\sim \in \{<, \leq, =, >, \geq\}$, $n \in \mathbb{Z}_{\geq 0}$. Заменим все подформулы вида $x - y = n$ на $x - y \leq n \wedge y - x \leq n$, а $x - y > n$ и $x - y \geq n$ — на $y - x < n$ и $y - x \leq n$ соответственно. Если в полученной формуле есть несколько ограничений с одинаковой левой частью, то заменим эти ограничения на ограничение, эквивалентное их конъюнкции, например, заменим $x - 0 \leq 2$ и $x - 0 < 1$ на $x - 0 < 1$. Таким образом мы построим формулу $\psi' \in \mathcal{B}(X)$, эквивалентную ψ, в которой содержится не более $|X_0| \times |X_0|$ ограничений-коньъюнктов, и все они имеют вид $x - y < n$ или $x - y \leq n$. Формула ψ' может быть задана матрицей D размера $|X_0| \times |X_0|$, называемой матрицей ограниченных разностей, в которой каждый элемент имеет вид ∞, $(\leq, k)$ или $(<, k)$ (где $k \in \mathbb{Z}_{\geq 0}$). Если формула ψ' содержит конъюнкт $x_i - x_j \sim n$, то положим $D_{i,j} = (\sim, n)$; иначе, если в ψ' нет ограничения с левой частью $x_i - x_j$, то положим $D_{i,j} = \infty$.

Таким образом, мы показали, как по формуле из множества $\mathcal{B}(X)$ построить DBM. Верно и обратное: по DBM D можно построить формулу из $\mathcal{B}(X)$. Пусть формула $\psi \in \mathcal{B}(X)$ соотвествует DBM D, тогда определим $[\![D]\!]$ равным $[\![\psi]\!]$ (напомним, что $[\![\psi]\!] = \{v | v \models \psi\}$). Будем говорить, что DBM D_1 и D_2 эквивалентны, если $[\![D_1]\!] = [\![D_2]\!]$.

Пусть $\prec$ — минимальное по включению отношение порядка на множестве ограничений, для которого справедливо: $(<, n) \prec (\leq, n)$, $(n, \leq) \prec \infty$, $(\leq, n_1) \prec (\leq, n_2)$ и $(<, n_1) \prec (<, n_2)$, если $n_1 < n_2$. Будем использовать запись $D' \prec D$ для обозначения того факта, $D'_{i,j} \prec D_{i,j}$ для любых $i, j \in 0..|X_0| - 1$. Будем говорить, что DBM D имеет канонический вид, если ни одно из огра-

ничений, входящих в неё, нельзя сделать более строгим, т. е. не существует такой DBM D', что $D' \prec D$. Операции над множествами, представленными в виде DBM, удобнее проводить, если исходные DBM имеют канонический вид. Для приведения DBM к каноническому виду может использоваться теоретико-графовый алгоритм, основанный на алгоритме Флойда — Уоршелла, приведённый в работе [13].

Множество регионов, представимых в виде DBM, замкнуто относительно операций пересечения, up, $down$, $reset$ и $free$ [13]. Например, если DBM D, находится в каноническом представлении, то DBM D', для которой справедливо $[\![D']\!] = up([\![D]\!])$, можно построить следующим образом: заменить все элементы $D_{i,0}$ на ∞; для построения DBM D'', для которой верно $[\![D'']\!] = down([\![D]\!])$, необходимо заменить все элементы $D_{0,i}$ на $(\geq, 0)$ (в обоих случаях $i = 1..|X_0| - 1$).

К сожалению, представление множеств значений таймеров в виде DBM позволяет задавать только выпуклые множества, следовательно это представление не замкнуто относительно операций дополнения и объединения. Для представления невыпуклых множеств используются конечные множества DBM, которые также называются семействами DBM (DBM federations) [13]. Будем говорить, что семейство DBM $\{D_1, D_2, \ldots, D_n\}$ характеризует регион Z, если $Z = \bigcup_{i=1..n} [\![D_i]\!]$. Представление регионов в виде семейств DBM замкнуто относительно операций пересечения, объединения, дополнения, up, $down$, $reset$ и $free$, таким образом, они могут использоваться для представления регионов в символьном алгоритме проверки симуляций.

Все необходимые операции над семействами DBM сводятся к операциям над парами DBM, входящими в эти семейства. В результате применения операций образуются семейства, некоторые элементы которых могут быть избыточны. Поэтому в процессе работы символьных алгоритмов некоторые

элементы семейств DBM заменяются на их объединения (когда объединение также представимо DBM), а избыточные элементы удаляются [34].

Из-за того, что целочисленные константы, входящие в DBM, неограничены, существует бесконечное число попарно неэквивалентных DBM, поэтому если регион Z, входящий в мета-состояние $(s_1, s_2, Z)_S$ (или $(s_1, s_2, Z, a)_D$), может быть задан произвольной DBM, то множество достижимых символьных состояний может оказаться бесконечно большим. Чтобы сделать это множество конечным, введём понятие нормализации.

Рассмотрим сначала случай, когда предохранители временных автоматов задаются формулами, не содержащих выражения вида $x - y \sim n$. Пусть k — наибольшая константа, используемая в формулах, описывающих инварианты состояний и предохранители некоторого временного автомата M, и X — множество таймеров этого автомата. Введём отношение эквивалентности $\simeq_k$ на множестве оценок $\mathbb{R}^X_{\geq 0}$ следующим образом: $v_1 \simeq_k v_2$ тогда и только тогда, когда для каждого таймера $x \in X$ выполняется $v_1(x) = v_2(x)$, или одновременно выполняется $v_1(x) > k$ и $v_2(x) > k$. Тогда любые два такие состояния (s, v_1) и (s, v_2), что $v_1 \simeq_k v_2$, бисимуляционно эквивалентны (имеется в виду отношение строгой бисимуляции на размеченной системе переходов, определяющей семантику M) [13].

Пусть теперь k — наибольшая константа, используемая в инвариантах состояний и предохранителях автоматов M_1 и M_2. Как было сказано выше, пары состояний $((s_1, v_1), (s_1, v_1'))$ и $((s_2, v_2), (s_2, v_2'))$ находятся в отношении строгой бисимуляции для любых дискретных состояний $s_1 \in S_1$ и $s_2 \in S_2$ и любых оценок v_1 и v_2 таких, что $v_1 \simeq_k v_1'$ и $v_2 \simeq_k v_2'$. Поэтому пара состояний $((s_1, v_1), (s_2, v_2))$ находится в отношении twa-симуляции H тогда и только когда, когда пара состояний $((s_1, v_1'), (s_2, v_2'))$ находится в отношении H. Следовательно, по теореме 3.3 состояния $((s_1, v_1), (s_2, v_2))_S$ и $((s_1, v_1'), (s_2, v_2'))_S$

игры $G_{\preceq_{twa},M_1,M_2}$ является эквивалентными в том смысле, что они одновременно являются выигрышными или проигрышными для игрока Spoiler. Поэтому в символьной игре нет смысла рассматривать такие мета-состояния $(s_1, s_2, Z)_S$, что существует таймер $x \in X$ и оценка $v \in Z$, для которых справедливо $v(x) > k$. Если же такое мета-состояние было построено, то множество Z необходимо нормализовать, т. е. заменить состояние $(s_1, s_2, Z)_S$ на $(s_1, s_2, norm(Z))_S$, где $norm(Z)$ содержит все такие оценки v, что для любого таймера $x \in X$ справедливо $v(x) \leq k$ и существует некоторая такая оценка $v' \in Z$, что $v \simeq_k v'$. Легко видеть, что область значений функции $norm$ конечно.

Оператор $norm$ может быть также реализован в виде операций над DBM [13]: необходимо заменить все такие элементы DBM $x - y \sim m$, что $m > k$, на ∞, и заменить все такие элементы $x - y \sim m$, что $m < -k$, на $x - y \sim -k$.

Если же в автоматах M_1 и M_2 присутствуют предохранители, содержащие выражения вида $x - y \sim n$, то в операторе нормализации необходимо учитывать и эти т. н. диагональные ограничения [13]. Результатом применения операции нормализации в этом случае будет уже не одна DBM, а семейство DBM. Если автоматы M_1 и M_2 содержат всего одно диагональное ограничение g, то результатом применения оператора нормализации к DBM D будет являться $\{norm(D'), norm(D'')\}$, где $norm$ — стандартный оператор нормализации (т. е. не учитывающий диагональных ограничений), а D' и D'' — это такие DBM, что $[\![D']\!] = [\![D]\!] \wedge g$ и $[\![D'']\!] = [\![D]\!] \wedge \neg g$. Если же автоматы M_1 и M_2 содержат более одного диагонального ограничения, то эту процедуру необходимо применить рекурсивно для всех остальных диагональных ограничений ко всем DBM полученного семейства [13].

3.4.4. Определение символьной игры

Пусть Z — некоторый регион, e — некоторый переход временного автомата $\langle S, s_0, X, \Sigma, R, Inv\rangle$ из дискретного состояния s в дискретное состояние s'. Обозначим $pred(Z, e) = \{v' | v \in Z \wedge (s, v') \xrightarrow{e} (s', v)\}$ и $post(Z, e) = \{v' | v \in Z \wedge (s, v) \xrightarrow{e} (s', v')\}$. Покажем, как значения операторов $pred(Z, e)$ и $post(Z, e)$ могут быть вычислены при помощи операций над DBM при условии, что Z тоже задано в виде DBM. Действительно, легко видеть, что $post(Z, e) = reset(Z \cap [\![Inv(s)]\!], Y) \cap [\![Inv(s')]\!]$ и $pred(Z, e) = release(Z \cap [\![Inv(s')]\!], Y) \cap [\![Inv(s)]\!]$.

Зададим символьную игру $G^s_{\preceq_{twa}, M_1, M_2} = \langle V^s_S, V^s_D, E^s_S, E^s_D, v^s_0\rangle$, состояния и переходы которой соответствуют некоторым *множествам* состояний и переходов игры $G^\delta_{\preceq_{twa}, M_1, M_2}$. Каждый переход игры $G^s_{\preceq_{twa}, M_1, M_2}$ будет помечен переходом одного из автоматов M_1 и M_2, т.е. $E^s_S \subseteq V^s_S \times (R_1 \cup R_2) \times (V^s_S \cup V^s_D)$ и $E^s_D \subseteq V^s_D \times (R_1 \cup R_2) \times (V^s_S \cup V^s_D)$. Опишем все компоненты игры $G^s_{\preceq_{twa}, M_1, M_2}$:

- $V^s_S = S_1 \times S_2 \times \tilde{\mathcal{B}}(X_1 \cup X_2)$,
- $V^s_D = S_1 \times S_2 \times \tilde{\mathcal{B}}(X_1 \cup X_2) \times \Sigma$
- $E^s_S = E^s_{S,1} \cup E^s_{S,2} \cup E^s_{S,3} \cup E^s_{S,4}$, где:
  - $((s_1, s_2, Z)_S, e, (s_1, s'_2, Z', a)_D) \in E^s_{S,1}$, если $e \in R^c_2$, $e = (s_2, g, a, Y, s'_2)$, $a \neq \tau$ и $Z' = post(Z, e)$
  - $((s_1, s_2, Z)_S, e, (s'_1, s_2, Z', a)_D) \in E^s_{S,2}$, если $e \in R^u_1$, $e = (s_1, g, a, Y, s'_1)$, $a \neq \tau$ и $Z' = post(Z, e)$
  - $((s_1, s_2, Z)_S, e, (s_1, s'_2, Z')_S) \in E^s_{S,3}$, если $e \in R^c_2$, $e = (s_2, g, \tau, Y, s'_2)$ и $Z' = norm(up(post(Z, e)) \cap [\![Inv_2(s'_2)]\!])$
  - $((s_1, s_2, Z)_S, e, (s'_1, s_2, Z')_S) \in E^s_{S,4}$, если $e \in R^u_1$, $e = (s_1, g, \tau, Y, s'_1)$ и $Z' = norm(up(post(Z, e)) \cap [\![Inv_2(s_2)]\!])$

- $E^s_D = E^s_{D,1} \cup E^s_{D,2} \cup E^s_{D,3}$, где:
  - $((s_1, s_2, Z, a)_D, e, (s'_1, s_2, Z')_S) \in E^s_{D,1}$, если $e \in R^c_1$, $e = (s_1, g, a, Y, s'_1)$ и $Z' = up(post(Z, e)) \cap [\![Inv_2(s_2)]\!]$
  - $((s_1, s_2, Z, a)_D, e, (s_1, s'_2, Z')_S) \in E^s_{D,2}$, если $e \in R^u_2$, $e = (s_2, g, a, Y, s'_2)$ и $Z' = up(post(Z, e)) \cap [\![Inv_2(s'_2)]\!]$
  - $((s_1, s_2, Z, a)_D, e, (s_1, s'_2, Z', a)_D) \in E^s_{D,3}$, если $e \in R^u_2$, $e = (s_2, g, \tau, Y, s'_2)$ и $Z' = post(Z, e)$
- $v_0 = (s_{10}, s_{20}, norm(up(\{\vec{0}\}) \cap [\![Inv_2(s_{20})]\!]))_S$.

Каждый ход символьной игры соответствует некоторому множеству переходов модифицированной игры, описанной в разделе 3.4.2. Например, переход $((s_1, s_2, Z)_S, e, (s_1, s'_2, Z', a)_D) \in E^s_{S,1}$ (где $e \in R^c_2$, $e = (s_2, g, a, Y, s'_2)$ и $a \neq \tau$ и $Z' = post(Z, e)$) соответствует множеству переходов $\{((q_1, q_2)_S, (q_1, q'_2, a)_D) | v \in Z \wedge q_1 = (s_1, v|_{X_1}) \wedge q_2 = (s_2, v|_{X_2}) \wedge q_2 \xrightarrow{e} q'_2\}$. Если игра оказывается в состоянии игрока Spoiler, то происходит нормализация временной зоны, входящей в это состояние.

3.4.5. Определение выигрышных состояний символьной игры

Каждое состояние игры $G^s_{\preceq_{twa}, M_1, M_2}$ соответствует *множеству* состояний игры $G^\delta_{\preceq_{twa}, M_1, M_2}$, и некоторые состояния из этого множества могут быть выигрышными для игрока Spoiler, а другие выигрышными для игрока Duplicator. В данном разделе мы опишем способ определения множества выигрышных состояний в игре $G^s_{\preceq_{twa}, M_1, M_2}$.

Будем считать, что функция $D(v^s)$ возвращает регион, входящий в состояние v^s, т. е. $D(v^s) = Z$, если $v^s = (s_1, s_2, Z)_S$ или $v^s = (s_1, s_2, Z, a)_D$. Оценкой игровых состояний символьной игры будем называть функцию, воз-

вращающую для каждого $v^s \in V_S^s \cup V_D^s$ некоторое подмножество множества $D(v^s)$. Наша цель — получить оценку m, характеризующую множество выигрышных состояний игрока Spoiler.

Пусть $E^s = E_S^s \cup E_D^s$. Определим оператор g^s, действующий на множестве оценок, следующим образом:

$$g^s(m^s)(v_1^s) = \begin{cases} D(v_1^s) \cap down\Big(D(v_1^s) \cap \big(\neg[\![Inv_1(s_1)]\!] \cup \bigcup_{(v_1^s,e,v_2^s)\in E_S^s} pred(m^s(v_2^s), e)\big)\Big) \\ \qquad \text{если } v_1^s \in V_S^s \text{ и } v_1^s = (s_1, s_2, z)_S \\ D(v_1^s) \setminus \bigcup_{(v_1^s,e,v_2^s)\in E_D^s} pred(D(v_2^s) \setminus m^s(v_2^s), e) \\ \qquad \text{если } v_1^s \in V_D^s \end{cases}$$

Оператор g^s представляет собой адаптацию ранее введённого оператора g^δ на случай символьного описания игры проверки симуляции. Как и оператор g^δ, оператор g^s является монотонным.

Легко видеть, что если $m^s(v^s)$ представимо DBM для каждого $v^s \in V^s$, то и значения $g(m^s(v^s))$ представимы в виде DBM для каждого $v^s \in V^s$. На множестве оценок определим отношение порядка $\leq$ следующим образом: $m_1^s \leq m_2^s$ тогда и только тогда, когда для любого $v^s \in V_S^s \cup V_D^s$ выполняется $m_1(v^s) \subseteq m_2(v^s)$. Оператор g^s монотонный (если в качестве отношения порядка выбрать $\leq$), и множество нормализованных регионов конечно, следовательно, по теореме Клини у него есть наименьшая неподвижная точка.

Теорема 3.5. *Пусть m_w^s — наименьшая неподвижная точка оператора g^s. Пусть $(s_1, s_2, Z)_S$ — произвольное состояние игрока Spoiler в игре $G^s_{\preceq_{twa},M_1,M_2}$. Тогда для любого $v \in Z$ справедливо: состояние $((s_1, v|_{X_1}), (s_2, v|_{X_2}))_S$ игры $G_{\preceq_{twa},M_1,M_2}$ является выигрышным для игрока Spoiler тогда и только тогда, когда v лежит в множестве $m_w^s((s_1, s_2, Z)_S)$.*

Доказательство. Будем говорить, что оценка m^s символьной игры

$G^s_{\preceq_{twa},M_1,M_2}$ соответствует оценке m модифицированной игры $G^\delta_{\preceq_{twa},M_1,M_2}$, если для любого состояния $(s_1, s_2, Z)_S \in E^s_S$ и для любого $v \in Z$ выполняется: $m(((s_1, v|_{X_1}), (s_2, v|_{X_2}))_S) = 1$ тогда, и только тогда, когда $v \in m^s((s_1, s_2, Z)_S)$. Аналогичное требование должно выполняться и для состояний игрока Duplicator.

Пусть m — некоторая оценка состояний игры $G^\delta_{\preceq_{twa},M_1,M_2}$, а m^s — соответствующая ей оценка символьный игровых состояний в игре $G^s_{\preceq_{twa},M_1,M_2}$. Тогда непосредственно из определений операторов g^s и g^δ следует, что оценка $g^s(m^s)$ соотвествует оценке $g^\delta(m)$. Следовательно, наименьшая неподвижная точка оператора g^s соответствует наименьшей неподвижной точке оператора g^δ. По теореме 3.4 эта неподвижная точка m^s_w соотвествует и наименьшей неподвижной точке оператора g, откуда следует справедливость доказываемого утверждения. □

3.4.6. Алгоритм решения символьной игры проверки twa-симуляции

По теореме 3.5 задача проверки twa-симуляции между TGA M_1 и M_2 эквивалентна нахождению неподвижной точки m^s оператора g^s и проверки того, что $\vec{0} \in m^s(v_0)$, где v_0 — начальное состояние игры $G^s_{\preceq_{twa},M_1,M_2}$. Количество состояний игры $G^s_{\preceq_{twa},M_1,M_2}$ бесконечно (поскольку в каждое состояние входит некоторая формула из $\tilde{\mathcal{B}}(X_1 \cup X_2)$), однако легко видеть, что из-за использования оператора нормализации $norm$ множество *достижимых* состояний конечно. Таким образом, символьный алгоритм проверки twa-симуляции схож с алгоритмом, описанным в разделе 2.4, и выглядит следующим образом:

- Построение множества достижимых состояний игры $G^s_{\preceq_{twa},M_1,M_2}$.

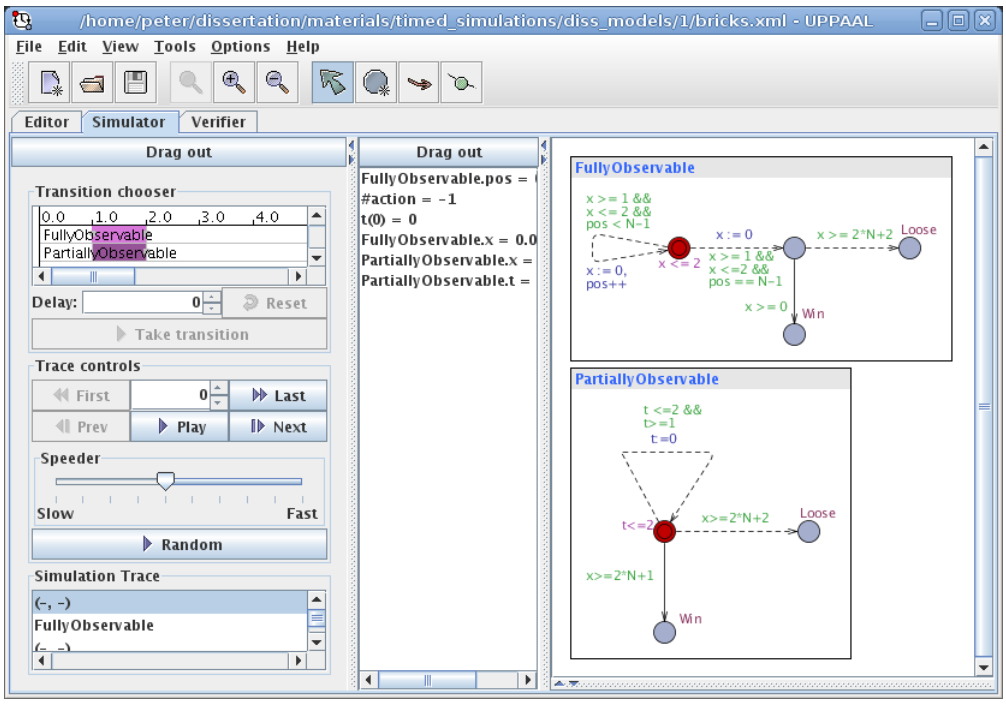

Рисунок 3.2. Проверка симуляции в средстве UPPAAL Tiga

- Построение последовательности Клини оператора g^s и нахождение наименьшей неподвижной точки m^s этого оператора.
- Проверка принадлежности $\vec{0}$ множеству $m^s(v_0^s)$, где v_0^s — начальное состояние игры $G^s_{\preceq_{twa},M_1,M_2}$.

В работе [40] мы предложили модификацию данного подхода, в которой множество выигрышных состояний (т. е. неподвижная точка оператора g^s) вычисляются одновременно с построением множества достижимых состояний. Это позволяет построить контрпример, показывающий что симуляции не существует, до построения всех достижимых состояний.

3.5. Описание практической реализации

Разработанный алгоритм проверки twa-симуляции был реализован в рамках среды анализа временных игровых автоматов UPPAAL Tiga [103]. Дан-

ная программная реализация была создана Александром Давидом (Alexandre David) из университета города Ольборг, Дания.

Средство UPPAAL Tiga позволяет задавать временные игровые автоматы как графически, так и в текстовом виде. В случае, если симуляция не выполняется, то пользователю предлагается удостовериться в этом, сыграв в игре проверке twa-симуляции за игрока Duplicator, а если симуляция выполняется — то за игрока Spoiler.

3.6. Результаты экспериментальных исследований

В данном разделе будет описано экспериментальное исследование реализации разработанного алгоритма twa-симуляции.

Задача синтеза контроллера (устройства управления) заключается в разработке такого алгоритма его работы, что при использовании этого алгоритма объект управления никогда не перейдёт в некорректное состояние, или наоборот рано или поздно окажется в целевом состоянии. Один из подходов к решению задачи *автоматического* синтеза контроллера основывается на использовании формальных моделей. В этом подходе строится модель S системы автоматического управления, представляюща собой игру, в которой задача контроллера — обеспечить корректное функционирование объекта управления, как бы не вели себя среда и объект управления. Задача автоматического синтеза контроллера сводится к задаче поиска его выигрышной стратегии в S.

Формализм временных игровых автоматов, описанный в данной работе, может использоваться для задания моделей систем автоматического управления, а среда анализа таких моделей, например, UPPAAL Tiga — для синтеза выигрышной стратегии контроллера.

В некоторых случаях состояние модели S, включает в себя скрытую от контроллера информацию о состоянии среды или объекта управления, которая не должна использоваться в алгоритме работы контроллера. Например, оно может включать в себя информацию о местоположении робота в пространстве, однако контроллеру робота должны быть доступны только показания датчиков, но не его точное местоположение. В таких случаях возникает задача поиска стратегии контроллера, которая не зависит от скрытой от него информации.

Игры с неполной информацией впервые были описаны в работе [61]. Позднее в работах [2, 91, 101, 112] были рассмотрены различные постановки задачи поиска выигрышной стратегии в играх с неполной информацией. В большинстве случаев задача анализа игры с неполной информацией G решалась путём автоматического сведения к задаче анализа игры с полной информацией G', каждое состояние которой соответствует некоторому множеству состояний исходной игры G.

В данной диссертационной работе предлагается альтернативный подход, основанный на ручном построении абстрактной модели, состояние которой содержит только видимую контроллеру информацию, и проверке того, что выигрышная стратегия контроллера для абстрактной модели подходит и для полной модели. Предлагаемый подход лучше ранее существовавших аналогов тем, что после его применения в наличии остаётся не только выигрышная стратегия контроллера, но и абстрактная модель, корректность которой доказана автоматически и с которой вдальнейшем можно работать независимо от исходной модели.

В этом подходе вначале строится модель A, в которую не входит информация о скрытом состоянии среды и объекта управления. Затем при помощи средства анализа временных автоматов строится выигрышная стратегия

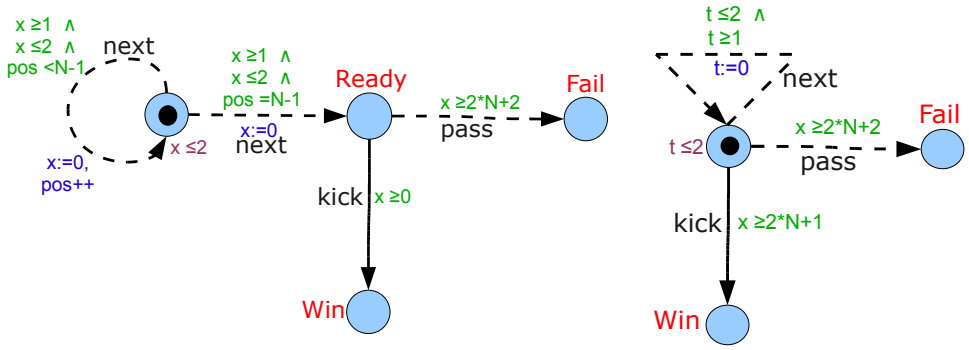

Рисунок 3.3. Временные автоматы, на которых проводилось экспериментальное исследование разработанного алгоритма

N	1000	2000	3000	4000	5000	6000	7000
Time	16s	1m8s	2m26s	4m43s	7m42s	10m51s	15m10s

N — параметр модели, задающий количество этапов обработки изделия,

Time — занятое время при выполнении на процессоре AMD Athlon 5200+.

Рисунок 3.4. Результаты экспериментального исследования разработанного алгоритма проверки twa-симуляции

контроллера f^c в игре (A, φ), где φ — спецификация, выполнимость которой необходимо обеспечить. Если такая стратегия существует, то необходимо проверить, что эта же стратегия будет выигрышной и для S. В этом можно убедиться, проверив существования twa-симуляции между S и A, т. е. проверив выполнимость $\mathsf{S} \preceq_{twa} \mathsf{A}$. Если это отношение симуляции существует, то по следствию 3.2 теоремы 3.1 стратегия f^c будет выигрышной и в игре (S, φ).

Рассмотрим применение данного подхода на следующем примере. Пусть имеется движущийся конвейер, на который ставится изделие. Затем изделие движется по конвейеру и последовательно проходит N этапов обработки, каждый этап занимает от 1 до 2 единиц времени. Задача контроллера — после завершения последнего этапа переместить готовое изделие с конвейе-

ра в коробку. Если этого не произойдёт в течение $2N + 2$ единиц времени с начала работы конвейера, то изделие будет потеряно, и, таким образом, задача контроллера не будет выполнена. Данная система описывается TGA S, изображённом на рисунке 3.3 слева. Сплошными дугами обозначены контролируемые переходы, а прерывистыми — неконтролируемые. В этом TGA используется целочисленная переменная pos, соответствующая положению изделия на конвейере, значение которой в начальном состоянии равно 1. В определении временного автомата 3.1 не позволялось использовать целочисленные переменные, однако, поскольку значение целочисленной переменной pos ограничено (оно принадлежит диапозону $1..N$), то данный автомат эквивалентен автомату, в котором нет целочисленных переменных и в котором имеется N копий каждого состояния автомата S. Задача контроллера в автомате S — перевести автомат в дискретное состояние Win, т. е. обеспечить выполнимость формулы $A\{next\}U\{kick\}$. Одна из возможных выигрышных стратегий контроллера f^c — подождать, пока автомат перейдёт в дискретное состояние $Ready$, а затем выполнить переход в состояние Win. Предположим теперь, что в реальной ситуации контроллеру неизвестно текущее положение изделие на конвейере. Тогда стратегия f^c не может использоваться в моделируемой (реальной) ситуации. Рассмотрим модель, изображённую на рисунке 3.3 справа. В состоянии этой модели уже не содержится информации о местоположении изделия, но тем не менее у контроллера существует выигрышная стратегия в игре $(\mathsf{A}, A\{next\}U\{kick\})$, которую можно построить при помощи средства UPPAAL Tiga. Для того, чтобы проверить, что данная стратегия будет выигрышной и для автомата S, необходимо проверить существование симуляции $\mathsf{S} \preceq_{twa} \mathsf{A}$. К сожалению, используя описанный в данной работе алгоритм проверки twa-симуляции, это нельзя сделать для модели с произвольным размером конвейера, задавамым параметром N. Однако, мы удосто-

верились в существовании симуляции для нескольких конкретных значений N, и на рисунке 3.4 приведены полученные времена работы алгоритма.

3.7. Область применимости

В литературе описан целый ряд расширений определения временных автоматов. Наиболее часто применяются:

- игровые временные автоматы [71], которые были описаны в разделе 3.1.3;
- взвешенные временные автоматы (weightened/priced timed automata) [52], которые используются для построения оптимальных расписаний и моделирования потребления энергии в системах реального времени;
- вероятностные временные автоматы (probalistic timed automata) [99], используемые для моделирования рандомизированных протоколов и для проверки свойств отказоустойчивых систем.

К сожалению, среди перечисленных типов временных автоматов приведённый в данной работе алгоритм подходит только для проверки симуляций между временными автоматами (базовой версии) и временными игровыми автоматами.

Заключение

В данной диссертационной работе исследованы отношения подобия, возникающие в задачах реструктуризации и верификации программ.

Работа состоит из двух частей. В первой части исследована задача проверки схожести в устройстве программ (синтаксического подобия), а именно, рассмотрена проблема обнаружения похожих фрагментов кода (клонов, дубликатов кода). Во второй части исследуется схожесть поведения программ (семантическое подобие): рассмотрены отношения симуляции — отношения подобия между моделями программ, сохраняющие выполнимость формул темпоральных логик. В качестве формализма задания моделей программ использовались структуры Крипке и временные автоматы.

В диссертационной работе получены следующие основные результаты:

1. Разработан новый алгоритм обнаружения подобных фрагментов в исходных кодах программ (программных клонов), позволяющий находить клоны, от которых можно избавиться при помощи реорганизации кода. Данный алгоритм был реализован в средстве, распространяемом под открытой лицензией. Проведено экспериментальное сравнение результатов работы Clone Digger и двух других средств Asta и CloneDR на основе методики, общепринятой при изучении эффективности средств выявления клонов. Эксперименты показали, что разработанное нами средство превосходит два другие средства как по полноте так и по точности поиска.

2. Разработан формальный язык, позволяющий задавать различные отношения подобия (симуляции) между структурами Крипке посредством правил антогонистической игры. Также разработано программно-

инструментальное средство, которое позволяет по описанию симуляции на этом языке и описанию двух моделей на языке NuSMV, проверять выполняется ли данная симуляция между двумя данными моделями. Проведено экспериментальное исследование разработанного средства при инкрементальной разработке моделей программ.

3. Впервые было разработано отношение симуляции, сохраняющее выполнимость формул логики ATCTL на множестве временных игровых автоматов; приведён алгоритм проверки выполнимости данного отношения. Проведено практическое испытание реализации данного алгоритма в задаче синтеза контроллера для системы автоматического управления, модель которой содержит невидимую контроллеру информацию.

Приношу глубокую благодарность своим научным руководителям проф. Р.Л. Смелянскому и доценту В.А. Захарову за постановку задачи и обсуждения в процессе её решения.

Литература

[1] The algorithmic analysis of hybrid systems / R. Alur, C. Courcoubetis, N. Halbwachs et al. // *Theoretical Computer Science.* — 1995. — Vol. 138. — Pp. 3–34.

[2] Algorithms for omega-regular games with imperfect information / J.-F. Raskin, K. Chatterjee, L. Doyen, Th. A. Henzinger // *Logical Methods in Computer Science.* — 2007. — Vol. 3, no. 3.

[3] Alternating refinement relations / R. Alur, Th. A. Henzinger, O. Kupferman, M. Y. Vardi // CONCUR. — Vol. 1466 of *LNCS.* — Springer, 1998. — Pp. 163–178.

[4] *Alur R., Courcoubetis C., Dill D. L.* Model-checking in dense real-time // *Inf. Comput.* — 1993. — Vol. 104, no. 1. — Pp. 2–34.

[5] *Alur R., Dill D.* A theory of timed automata // *Theoretical Computer Science.* — 1994. — Vol. 126, no. 2. — Pp. 183–235.

[6] *Alur R., Henzinger Th. A., Kupferman O.* Alternating-time temporal logic // FOCS. — 1997. — Pp. 100–109.

[7] *Andersen H. R.* Model checking and boolean graphs // *Theoretical Computer Science.* — 1994. — Vol. 126, no. 1. — Pp. 3–30.

[8] Assessing the benefits of incorporating function clone detection in a development process / B. Lague, D. Proulx, J. Mayrand et al. // ICSM '97: Proceedings of the International Conference on Software Maintenance. — Washington, DC, USA: IEEE Computer Society, 1997. — Pp. 314–321.

[9] *Baader F., Snyder W.* Unification theory // Handbook of Automated Reasoning / Ed. by J.A. Robinson, A. Voronkov. — Elsevier Science Publishers, 2001. — Vol. I. — Pp. 447–533.

[10] *Baker B. S.* A program for identifying duplicated code // Computer Science and Statistics: Proc. Symp. on the Interface. — 1992. — March. — Pp. 49–57.

[11] *Baker B. S.* On finding duplication and near-duplication in large software systems // WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering. — Washington, DC, USA: IEEE Computer Society, 1995. — Pp. 86–95.

[12] *Bellman R.* Dynamic Programming. — Princeton University Press, 1957.

[13] *Bengtsson J., Yi W.* Timed automata: Semantics, algorithms and tools // Lecture Notes in Computer Science. — New York, NY, USA: Springer, 2004. — Pp. 87–124.

[14] *Bille P.* A survey on tree edit distance and related problems // *Theor. Comput. Sci.* — 2005. — Vol. 337, no. 1-3. — Pp. 217–239.

[15] *Browne M. C., Clarke E. M., Grümberg O.* Characterizing finite kripke structures in propositional temporal logic // *Theoretical Computer Science.* — 1988. — Vol. 59, no. 1-2. — Pp. 115–131.

[16] *Bryant Randal E.* Graph-based algorithms for boolean function manipulation // *IEEE Transactions on Computers.* — 1986. — Vol. 35, no. 8. — Pp. 677–691.

[17] *Bulychev P.E., Konnov I.V., Zakharov V.A.* Computing (bi)simulation relations preserving ctlx logic for ordinary and fair kripke structures // *Tpy-*

ды Института системного программирования РАН. — 2007. — Т. 12. — С. 59–76.

[18] *Bulychev P., Kostylev E., Zakharov V.* Anti-unification algorithms and their applications in program analysis // PSI. — 2009.

[19] *Burd E., Munro M.* Investigating the maintenance implications of the replication of code // in the Proceeedings of the International Conference on Software Maintenance; ICSM'97. — IEEE Press, 1997. — Pp. 1–3.

[20] *Bustan D., Grumberg O.* Simulation-based minimization // *ACM Transactions on Computational Logic.* — 2003. — Vol. 4, no. 2. — Pp. 181–206.

[21] *Butler M.* Incremental design of distributed systems with event-b // Engineering Methods and Tools for Software Safety and Security - Marktoberdorf Summer School. — IOS Press, 2008. — Pp. 131–160.

[22] *Cerans K.* Decidability of bisimulation equivalences for parallel timer processes. // CAV. — Vol. 663 of *LNCS.* — Springer, 1992. — Pp. 302–315.

[23] *Cerans K., Godskesen J. C., Larsen K. G.* Timed modal specification – theory and tools. // CAV. — Vol. 697 of *LNCS.* — Springer, 1993. — Pp. 253–267.

[24] *Chockler H., Kupferman O., Vardi M. Y.* Coverage metrics for temporal logic model checking // *Formal Methods in System Design.* — 2006. — Vol. 28, no. 3. — Pp. 189–212.

[25] *Clarke E. M., Emerson E. A., Sistla A. P.* Automatic verification of finite-state concurrent systems using temporal logic specifications // *ACM Trans. Program. Lang. Syst.* — 1986. — Vol. 8, no. 2. — Pp. 244–263.

[26] Clone detection using abstract syntax trees / I. D. Baxter, A. Yahin, L. Moura et al. // ICSM '98: Proceedings of the International Conference on Software Maintenance. — Washington, DC, USA: IEEE Computer Society, 1998. — P. 368.

[27] Comparison and evaluation of clone detection tools / S. Bellon, R. Koschke, G. Antoniol et al. // *IEEE Transactions on Software Engineering.* — 2007. — Vol. 33, no. 9. — Pp. 577–591.

[28] *Cooper K. D., McIntosh N.* Enhanced code compression for embedded risc processors // PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation. — ACM, 1999. — Pp. 139–149.

[29] *Cordy James R.* Comprehending reality - practical barriers to industrial adoption of software maintenance automation // IWPC. — 2003. — Pp. 196–206.

[30] Counterexample-guided abstraction refinement for symbolic model checking / E. Clarke, O. Grumberg, S. Jha et al. // *Journal of the ACM.* — 2003. — September. — Vol. 50, no. 5. — Pp. 752–794.

[31] Cp-miner: Finding copy-paste and related bugs in large-scale software code / Z. Li, S. Lu, S. Myagmar, Y. Zhou // *IEEE Transactions on Software Engineering.* — 2006. — Vol. 32, no. 3. — Pp. 176–192.

[32] Cudd: Colorado university bdd package. — `http://vlsi.colorado.edu/~fabio/CUDD`.

[33] *D. Stéphane, R. Matthias, D. Serge.* A language independent approach for

detecting duplicated code // International Conference on Software Maintenance. — IEEE, 1999. — Pp. 109–118.

[34] *David A.* Merging DBMs efficiently // 17th Nordic Workshop on Programming Theory. — DIKU, University of Copenhagen, 2005. — October. — Pp. 54–56.

[35] *De Nicola R., Vaandrager F.* Three logics for branching bisimulation // *Journal of the ACM.* — 1995. — Vol. 42, no. 2. — Pp. 458–487.

[36] Detection of software clones: Tool comparison experiment. — `http://www.bauhaus-stuttgart.de/clones/index.html`.

[37] *Dijkstra E. W.* Hierarchical ordering of sequential processes // *Acta Informatica.* — 1971. — Vol. 1. — Pp. 115–138.

[38] *Ducasse S., Rieger M., Demeyer S.* A language independent approach for detecting duplicated code. — IEEE, 1999. — Pp. 109–118.

[39] *Edelkamp S.* Symbolic exploration in two-player games: Preliminary results // In Proceedings of the Sixth International Conference on AI Planning and Scheduling (AIPS-02) Workshop on Model Checking. — 2002. — Pp. 40–48.

[40] Efficient on-the-fly algorithm for checking alternating timed simulation / P. Bulychev, T. Chatain, A. David, K. G. Larsen // 7th International Conference on Formal Modelling and Analysis of Timed Systems. — 2009.

[41] Efficient on-the-fly algorithms for the analysis of timed games / F. Cassez, A. David, E. Fleury et al. // CONCUR. — Vol. 3653 of *LNCS.* — 2005. — Pp. 66–80.

[42] An empirical study of operating systems errors / A. Chou, J. Yang, B. Chelf et al. // SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles. — New York, NY, USA: ACM, 2001. — Pp. 73–88.

[43] *Etessami K.* A hierarchy of polynomial-time computable simulations for automata // CONCUR 2002. — 2002. `citeseer.ist.psu.edu/etessami02hierarchy.html`.

[44] *Etessami K., Wilke T., Schuller R. A.* Fair simulation relations, parity games, and state space reduction for büchi automata // *SIAM Journal on Computing.* — 2005. — Vol. 34, no. 5. — Pp. 1159–1175.

[45] *Evans W. S., Fraser C. W., Ma F.* Clone detection via structural abstraction // *Reverse Engineering.* — 2007. — Vol. 0. — Pp. 150–159.

[46] *F. Jeanne, O. Karl J., W. Joe D.* The program dependence graph and its use in optimization // *ACM Transactions on Programming Languages and Systems.* — 1987. — Vol. 9. — Pp. 319–349.

[47] *Fernandez J.-C.* An implementation of an efficient algorithm for bisimulation equivalence // *Science of Computer Programming.* — 1989. — Vol. 13. — Pp. 13–219.

[48] Gplag: Detection of software plagiarism by program dependence graph analysis / C. Liu, C. Chen, J. Han, P. S. Yu // In the Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'06. — ACM Press, 2006. — Pp. 872–881.

[49] Graphviz - open source graph drawing tools / J. Ellson, E. R. Gansner, E. Koutsofios et al. // *Graph Drawing.* — 2001. — Pp. 483–484.

[50] *Groote J. F., Vaandrager F. W.* An efficient algorithm for branching bisimulation and stuttering equivalence // Automata, Languages and Programming. — 1990. — Pp. 626–638.

[51] Guided controller synthesis for climate controller using UPPAAL-TIGA / J. J. Jessen, J. I. Rasmussen, K. G. Larsen, A. David // Proceedings of the 19th International Conference on Formal Modeling and Analysis of Timed Systems. — LNCS no. 4763. — Springer, 2007. — Pp. 227–240.

[52] Guiding and cost-optimality in uppaal / Gerd Behrmann, Ansgar Fehnker, Thomas Hune et al. // AAAI Spring Symposium Model-Based Validation of Intelligence. — 2001.

[53] *Gusfield D.* Algorithms on strings, trees, and sequences : computer science and computational biology. — Cambridge Univ. Press, 2007. — January.

[54] *Henzinger M. R., Henzinger Th. A., Kopke P. W.* Computing simulations on finite and infinite graphs // 36th IEEE Symp. on Foundations of Comp. Sci. — IEEE Computer Society Press, 1996. — Pp. 453–462.

[55] *Henzinger Th. A.* The theory of hybrid automata. — IEEE Computer Society Press, 1996. — Pp. 278–292.

[56] *Henzinger Th. A., Kupferman O., Rajamani S. K.* Fair simulation // *Information and Computation.* — 2002. — Vol. 173, no. 1. — Pp. 64–81.

[57] *Holzmann G. J.* The model checker spin // *Software Engineering.* — 1997. — Vol. 23, no. 5. — Pp. 279–295.

[58] *Jensen H. E., Larsen K. G., Skou A.* Scaling up Uppaal automatic verification of real-time systems using compositionality and abstraction. // FTRTFTS. — Vol. 1926 of *LNCS.* — Springer, 2000. — Pp. 19–30.

[59] *Johnson J.H.* Substring matching for clone detection and change tracking. — 1994.

[60] *Johnson J. H.* Identifying redundancy in source code using fingerprints // Proc. Conf. Centre for Advanced Studies on Collaborative research (CASCON). — IBM Press, 1993. — Pp. 171–183.

[61] *Jones N. D.* Blindfold games are harder than games with perfect information. — 1978. — Vol. 6:4-7.

[62] *Kamiya T., Kusumoto S., Inoue K.* Ccfinder: A multilinguistic token-based code clone detection system for large scale source code // *IEEE Transactions on Software Engineering.* — 2002. — Vol. 28, no. 7. — Pp. 654–670.

[63] *Kapser C., Godfrey M. W.* "cloning considered harmful"considered harmful // WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering. — IEEE Computer Society, 2006. — Pp. 19–28.

[64] *Komondoor R., Horwitz S.* Using slicing to identify duplication in source code // In Proceedings of the 8th International Symposium on Static Analysis. — Springer-Verlag, 2001. — Pp. 40–56.

[65] *Konnov I., Zakharov V.* An invariant-based approach to the verification of asynchronous parameterized networks // *Journal of Symbolic Computation.* — 2009.

[66] *Koschke R., Falke R., Frenzel P.* Clone detection using abstract syntax suffix trees // WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering. — Washington, DC, USA: IEEE Computer Society, 2006. — Pp. 253–262.

[67] *Krinke J., Softwaresysteme L.* Identifying similar code with program dependence graphs. — 2001. — Pp. 301–309.

[68] *Kupferman O., Vardi M. Y.* Module checking // Eighth International Conference on Computer Aided Verification CAV. — Vol. 1102. — Springer Verlag, 1996. — Pp. 75–86.

[69] *Lamport L.* What good is temporal logic? // IFIP 83. — 1983. — Pp. 657–668.

[70] *Lanubile F., Mallardo T.* Finding function clones in web applications // *Software Maintenance and Reengineering, European Conference on.* — 2003. — Vol. 0. — P. 379.

[71] *Maler O., Pnueli A., Sifakis J.* On the synthesis of discrete controllers for timed systems // STACS. — Vol. 900. — Springer, 1995. — Pp. 229–242.

[72] *Marcus A., Maletic J. I.* Identification of high-level concept clones in source code. — 2001.

[73] *Mayrand J., Leblanc C., Merlo E. M.* Experiment on the automatic detection of function clones in a software system using metrics // *Software Maintenance, IEEE International Conference on.* — 1996. — Vol. 0. — P. 244.

[74] *Milner R.* An algebraic definition of simulation between programs: Tech. rep. — Stanford, CA, USA: 1971.

[75] *Milner R.* A Calculus of Communicating Systems. — Springer-Verlag New York, Inc., 1980.

[76] *Nielsen M., Clausen C.* Bisimulation, games and logic // Dep. of Computer Science, University of Aarhus. — Springer-Verlag, 1994. — Pp. 289–306.

[77] Nusmv: a new symbolic model checker / A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri // *International Journal on Software Tools for Technology Transfer.* — 2000. — Vol. 2.

[78] *Oancea C., So C., Watt S. M.* Generalization in maple. — 2005.

[79] *Olson D. L., Delen D.* Advanced Data Mining Techniques. — Springer, 2008.

[80] *Paige R., Tarjan R. E.* Three partition refinement algorithms // *SIAM Journal on Computing.* — 1987. — Vol. 16, no. 6. — Pp. 973–989.

[81] *Park D.* Concurrency and automata on infinite sequences // Proceedings of the 5th GI-Conference on Theoretical Computer Science. — London, UK: Springer-Verlag, 1981. — Pp. 167–183.

[82] *Parr T. J., Quong R. W.* Antlr: A predicated-ll(k) parser generator // *Software Practice and Experience.* — 1994. — Vol. 25. — Pp. 789–810.

[83] Pattern matching for clone and concept detection / K. A. Kontogiannis, R. Demori, E. Merlo et al. // *Journal of Automated Software Engineering.* — 1996. — Vol. 3. — P. 108.

[84] Planning Robust Temporal Plans A Comparison Between CBTP and TGA Approaches / Y. Abdeddaïm, E. Asarin, M. Gallien et al. // Proceedings of the International Conference on Automated Planning and Scheduling International Conference on Automated Planning and Scheduling (ICAPS'07). — AAAI, 2007.

[85] *Plotkin G.D.* A note on inductive generalization // Machine Intelligence. — Edinburgh University Pres, 1970. — Vol. 5. — Pp. 153–163.

[86] *Pnueli A.* A temporal logic of concurrent programs // *Theoretical computer science.* — Vol. 13. — Pp. 45–60.

[87] Pycudd: an enhanced python interface to the colorado university bdd package. — `http://www.ece.ucsb.edu/bears/pycudd.html`.

[88] *Ramadge P. J. G., Wonham W. M.* The control of discrete event systems // *Proceedings of the IEEE.* — 1989. — Vol. 77, no. 1. — Pp. 81–98.

[89] *Ranzato F., Tapparo F.* Computing stuttering simulations // CONCUR 2009: Proceedings of the 20th International Conference on Concurrency Theory. — Berlin, Heidelberg: Springer-Verlag, 2009. — Pp. 542–556.

[90] *Raza A., Vogel G., Plödereder E.* Bauhaus - a tool suite for program analysis and reverse engineering // Ada-Europe. — 2006. — Pp. 71–82.

[91] *Reif John H.* The complexity of two-player games of incomplete information // *J. Comput. Syst. Sci.* — 1984. — Vol. 29, no. 2. — Pp. 274–301.

[92] *Reynolds J. C.* Transformational systems and the algebraic structure of atomic formulas // Machine Intelligence. — Edinburgh University Pres, 1970. — Vol. 5. — Pp. 135–151.

[93] *Roy C. K., Cordy J. R.* A survey on software clone detection research // *SCHOOL OF COMPUTING TR 2007-541, QUEEN'S UNIVERSITY.* — 2007. — Vol. 115.

[94] *Sabelfeld A.* Confidentiality for multithreaded programs via bisimulation // Andrei Ershov International Conference on Perspectives of System Informatics. — Springer-Verlag, 2003. — Pp. 260–273.

[95] Sigref – a symbolic bisimulation tool box / R. Wimmer, M. Herbstritt, H. Hermanns et al. // Proceedings of the 4th International Symposium on Automated Technology for Verification and Analysis (ATVA). — Vol. 4218 of *Lecture Notes in Computer Science*. — Springer-Verlag, 2006. — Pp. 477–492.

[96] Software quality analysis by code clones in industrial legacy software / A. Monden, D. Nakae, T. Kamiya et al. // METRICS '02: Proceedings of the 8th International Symposium on Software Metrics. — Washington, DC, USA: IEEE Computer Society, 2002. — Pp. 87–94.

[97] *Sørensen M. H., Glück R.* An algorithm of generalization in positive supercompilation // ILPS. — 1995. — Pp. 465–479.

[98] *Stirling C.* Modal and temporal logics for processes // Summer School in Logic Methods in Concurrency. — 1993.

[99] Symbolic model checking for probabilistic timed automata / M. Kwiatkowska, G. Norman, J. Sproston, F. Wang // Proc. Joint Conference on Formal Modelling and Analysis of Timed Systems and Formal Techniques in Real-Time and Fault Tolerant Systems (FORMATS/FTRTFT'04) / Ed. by Y. Lakhnech, S. Yovine. — Vol. 3253 of *LNCS*. — Springer, 2004. — Pp. 293–308.

[100] *Tarski A.* A lattice-theoretical fixpoint theorem and its applications. — 1955.

[101] Timed control with observation based and stuttering invariant strategies / A. David, F. Cassez, K. G. Larsen et al. // 5th International Symposium on Automated Technology for Verification and Analysis (ATVA 2007). — Lecture Notes in Computer Science. — Springer, 2007.

[102] *Turing Alan M.* Computing machinery and intelligence // *Mind.* — 1950. — Vol. LIX. — Pp. 433–460.

[103] UPPAAL-TIGA: Time for playing games! / G. Behrmann, A. Cougnard, A. David et al. // Proceedings of the 19th International Conference on Computer Aided Verification. — LNCS no. 4590. — Springer, 2007. — Pp. 121–125.

[104] *van Glabbeek R.* The linear time - branching time spectrum // CONCUR '90 Theories of Concurrency: Unification and Extension / Ed. by J. Baeten, J. Klop. — Springer Berlin / Heidelberg, 1990. — Vol. 458 of *Lecture Notes in Computer Science.* — Pp. 278–297.

[105] *van Glabbeek Rob J., Weijland W. Peter.* Branching time and abstraction in bisimulation semantics // *Journal of the ACM.* — 1996. — Vol. 43, no. 3. — Pp. 555–600.

[106] *van Langevelde I.A.* Symmetry in labeled transition systems. — 2003.

[107] *Vardi M. Y.* Alternating automata and program verification // In Computer Science Today. LNCS 1000. — Springer-Verlag, 1995. — Pp. 471–485.

[108] *Walenstein A., Lakhotia A.* The software similarity problem in malware analysis // Duplication, Redundancy, and Similarity in Software / Ed. by Rainer Koschke, Ettore Merlo, Andrew Walenstein. — Dagstuhl Seminar Proceedings no. 06301. — Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[109] *Weise C., Lenzkes D.* Efficient scaling-invariant checking of timed bisimulation // STACS. — Vol. 1200 of *LNCS.* — 1997. — Pp. 177–188.

[110] *Wettel R., Marinescu R.* Archeology of code duplication: Recovering duplication chains from small duplication fragments // *Symbolic and Numeric Algorithms for Scientific Computing, International Symposium on.* — 2005. — Vol. 0. — Pp. 63–70.

[111] *Wimmer R., Herbstritt M., Becker B.* Minimization of large state spaces using symbolic branching bisimulation // Proceedings of the 9th IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS). — IEEE Computer Society Press, 2006. — April. — Pp. 9–14.

[112] *Wulf M. D., Doyen L., Raskin J.-F.* A lattice theory for solving games of imperfect information // HSCC, LNCS 3927. — Springer, 2006. — Pp. 153–168.

[113] *Yang W.* Identifying syntactic differences between two programs // *Software - Practice and Experience.* — 1991. — Vol. 21. — Pp. 739–755.

[114] *А. Ахо Р. Сети Д. Ульман.* Компиляторы. Принципы, технологии, инструменты. — Вильямс, 2003.

[115] *Даль В.И.* Толковый словарь живого великорусского языка. — С.-Петербург - Москва: Издание книгопродавца-типографа М. О. Вольфа, 1882.

[116] *Дж. Ту, Р. Гонсалес.* Принципы распознавания образов. — Мир, 1978.

[117] *Ершов А. П.* Современное состояние теории схем программ // *Проблемы кибернетики.* — 1973. — Vol. 27. — Pp. 87–110.

[118] *Кларк Э. М., Грамберг О., Пелед Д.* Верификация моделей программ: Model checking. — МЦНМО, 2002.

[119] *Костылев Е.* Алгоритмы антиунификации и их применение для вычисления инвариантов программ: Ph.D. thesis / Моск. гос. ун-т им. М.В. Ломоносова. — 2008.

[120] *Котов В.Е., Сабельфельд В.К.* Теория схем программ. — Наука, 1991. — P. 248.

[121] *Фаулер М.* Рефакторинг. Улучшение существующего кода. — Символ-Плюс, 2008.

Приложение А

Руководство по использованию разработанных программных средств

А.1. Запуск Clone Digger

Среда обнаружения клонов может запускаться двумя способами: либо из среды разработки Eclipse, либо из командной строки.

Плагин к Eclipse был разработан сторонним разработчиком в рамках программы Google Summer of Code 2008, его описание можно найти на странице[1].

Синтаксис запуска из командной строки имеет следующий вид:

```
python clonedigger.py
     [--lang=LANGUAGE] [--distance-threshold=DISTANCE_THRESHOLD]
     [--size-threshold=SIZE_THRESHOLD] [--hashing-depth=HASHING_DEPTH]
     [--output=OUTPUT_FILENAME]
     LIST_OF_FILES
```

Аргументы командной строки имеют следующее значение:

- `LANGUAGE` — используемый язык. Поддерживаемые значения: `Python`, `Java`, `Lua`, `Javascript`. По умолчанию подрузамевается `Python`.
- `DISTANCE_THRESHOLD` — максимальное расстояние по антиунификации между фрагментами, составляющими клон. Значение по умолчанию зависит от языка программирования.

[1] http://clonedigger.sourceforge.net/eclipse_plugin_manual/eclipse_plugin_manual.html

- `SIZE_THRESHOLD` — минимальный размер фрагмента кода, входящий в клон, в строках кода. Значение по умолчанию зависит от языка программирования.

- `HASHING_DEPTH` — глубина хеширования, введённая в разделе 1.4.1.

- `LIST_OF_FILES` — разделённый пробелами список файлов в которых необходимо искать клоны. Если в этом списке встречается имя директории, то она обрабатывается рекурсивно.

- `OUTPUT_FILENAME` — имя файла, куда записывается отчёт с найденными клонами. По умолчанию `output.html`.

A.2. Запуск универсальной среды проверки симуляции

Среда запускается из командной строки Linux. Синтаксис запуска следующий:

```
python gsim.py --game=game.gam
  --larger-model=larger_model.smv --smaller-model=smaller_model.smv
  [--hidden-vars-regexp=hidden_vars]
  [--visible-vars-regexp=visible_vars]
```

, где `game.gam` — имя файла описания правил игры, `larger_model.smv` и `smaller_model.smv` — имена файлов описания моделей, `visible_vars` и `hidden_vars` — регулярные выражения, задающие множества видимых и скрытых переменных моделей соответственно. По умолчанию все переменные моделей считаются видимыми.