Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Кафедра «Компьютерные технологии»

Д. Ю. Кочелаев

Методы динамической проверки правил непротиворечивости автоматной модели

Бакалаврская работа

Руководитель – В. С. Гуров

Санкт-Петербург
2007

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

При проектировании программного обеспечения в настоящее время для описания классов и поведения системы активно используется язык *UML* [1 – 3]. Возможно, одним из наиболее удобных способов описания поведения системы являются диаграммы состояний, описывающие поведение детерминированных конечных автоматов [4]. Напомним, что конечный автомат — в теории алгоритмов математическая абстракция, позволяющая описывать пути изменения состояния объекта в зависимости от его текущего состояния и входных данных, при условии, что общее возможное число состояний конечно. Важным моментом является то, что рассматриваемые конечные автоматы, описывающие поведение, детерминированные. Детерминированным конечным автоматом называется такой автомат, в котором при любой последовательности входных данных существует лишь одно состояние, в которое автомат может перейти из текущего состояния.

В автоматах переходы выполняются не только по некоторому событию, но и при определенных условиях. Также при переходе возможен вызов выходных воздействий, что позволяет изменять объекты управления.

Расширенным вариантом диаграммы состояний являются автоматы, используемые в *SWITCH*-технологии [5, 6], в которых в состояния могут быть вложены другие автоматы.

Любая диаграмма, описывающая какую-либо часть программного обеспечения, должна соответствовать некоторой метамодели. Метамодель («мета» обозначает находящийся вне, за пределами, сверх) — это модель, которая описывает структуру, принципы действия другой модели. Например, в случае диаграмм состояний метамодель будет описывать структуру всех таких диаграмм, а моделям будут конкретные экземпляры диаграмм состояний. При этом возникает проблема проверки соответствия моделей метамодели – проверка непротиворечивости метамодели.

Рассмотрим основные задачи, решаемые в данной работе.

Первая задача – построение описания автоматной метамодели. При этом необходимо учесть, что описание автоматной метамодели должно быть удобным для проверки соответствия моделей этому описанию.

Вторая задача – разработка эффективного алгоритма проверки непротиворечивости модели.

В главе 1 приведено описание автоматной метамодели, а также выполнен обзор существующих методов поддержки модели в непротиворечивом метамодели состоянии и алгоритмов проверки соответствию метамодели.

В главе 2 предлагается новый алгоритм проверки правил непротиворечивости модели и приведен пример автоматной модели. На этой модели в дальнейшем будет продемонстрирована работа нового алгоритма и проведено сравнение нового и существующих алгоритмов.

В главе 3 описывается реализация нового алгоритма и его внедрение в инструментальное средство *UniMod* [7 – 9].

В главе 4 будет проведено экспериментальное сравнение нового алгоритма и существующих алгоритмов проверки непротиворечивости автоматной модели.

ГЛАВА 1. АВТОМАТНАЯ МЕТАМОДЕЛЬ И СУЩЕСТВУЮЩИЕ АЛГОРИТМЫ ПРОВЕРКИ

1.1. Описание модели

В данной работе методы проверки непротиворечивости рассматриваются на примере автоматной метамодели. Рассмотрим *UML*-диаграмму, изображенную на рис. 1 и описывающую эту модель, и приведем ее более полное словесное описание.

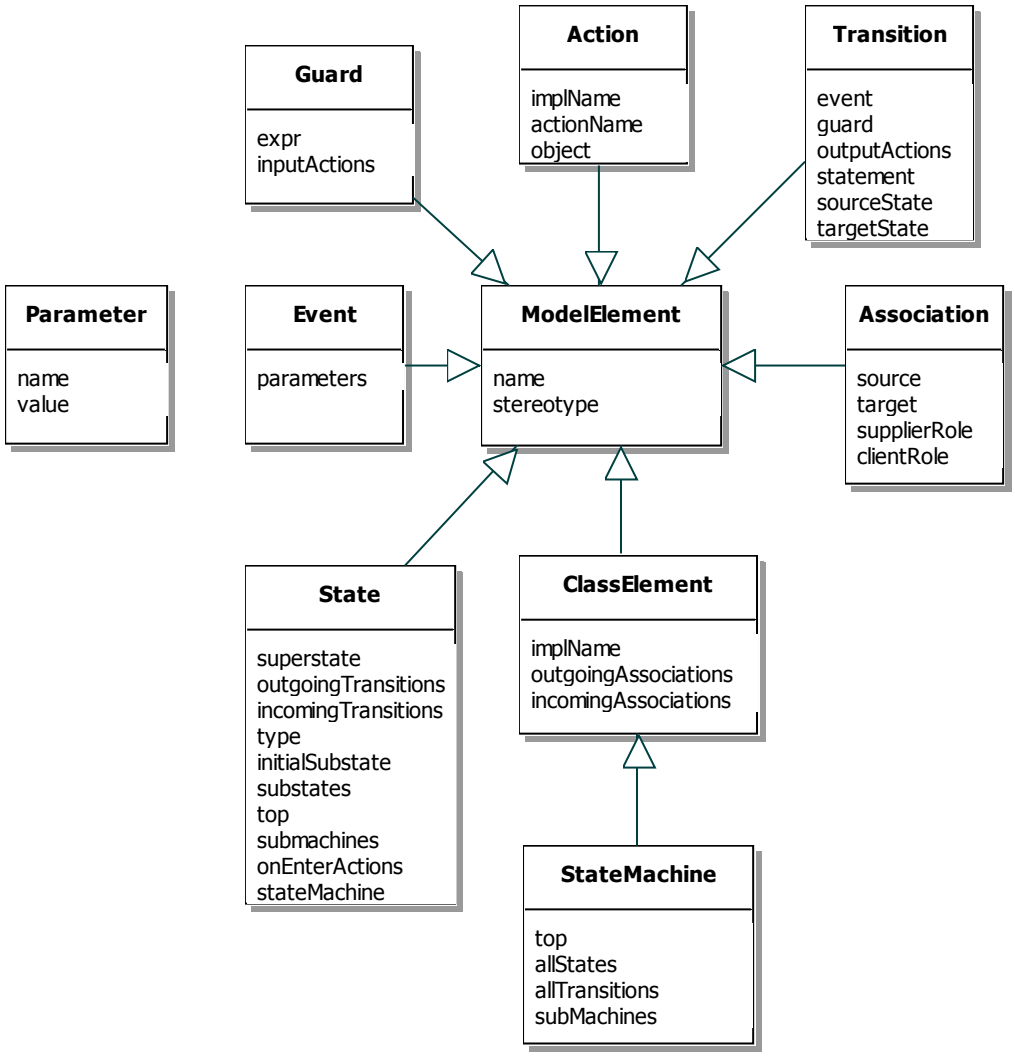

Рис. 1. Диаграмма классов автоматной модели

Базовым элементом модели является интерфейс *ModelElement*, все остальные элементы модели наследуются от него напрямую или косвенно – наследуя интерфейс, который наследует *ModelElement*. Атрибутами этого класса является имя элемента и его стереотип.

StateMachine — это интерфейс, описывающий конечный автомат, наиболее общий объект, которым будем оперировать в этой работе. В общем случае автомат представляет собой набор состояний и переходов между ними. При этом среди состояний должно быть выделено одно и только одно начальное состояние, с которого начинается работа с автоматом. Также среди состояний могут быть выделены конечные состояния. После перехода в конечное состояние работа автомата заканчивается. Таким образом, становится ясным одно из ограничений на модель – отсутствие переходов из конечных состояний, так как они никогда не будут задействованы. Также один автомат *A1* может быть ассоциирован с другим автоматом *A2*, в случае если в одно или несколько состояний автомата *A1* вложены экземпляры автомата *A2*.

Состояние автомата описываются интерфейсом *State*, который напрямую наследует интерфейс *ModelElement*. Атрибут *type* описывает тип состояния. Состояние может быть одного из трех типов: начальное, конечное и обычное. Обычное состояние может иметь вложенные в него состояния и автоматы. Доступ к ним можно получить через атрибуты *substates* и *submachines*. Если состояние вложенное, то на состояние, родительское для него, ссылается атрибут *superstate*. На вложенные состояния распространяются те же ограничения, что и на состояния, находящиеся непосредственно в автомате. Например, среди вложенных состояний должно быть одно и только одно начальное состояние. В связи с этим появляется возможность унифицировать представление состояний вложенных в другие состояния и находящихся непосредственно в автомате. Для этих целей все состояния, находящие в автомате, вложим в одно состояние, которое назовем корневым. Это будет единственное состояние находящееся непосредственно в автомате, также запретим добавлять другие такие состояния.

Как уже писалось выше, все состояния соединены переходами, которые описываются интерфейсом *Transition*. Каждое состояние имеет ссылки на все исходящие из него и входящие в него переходы. Переходы, в свою очередь, имеют атрибуты, ссылающиеся на состояния, находящиеся на обоих концах перехода. Кроме состояний, к которым привязан данный переход, хранится также событие, по которому автомат перейдет из исходного состояния перехода в его целевое состояние. Вместе с событием с переходом связано условие, которое должно выполняться при этом перехода. Таким образом, для одного события можно иметь набор переходов, но с разными условиями. Здесь возникает еще одно ограничение на модель. Для всех событий из состояния должны быть переходы. При этом при любом значении переменных, используемых в условиях на переходах, одно и только одно условие должно выполняться. Отметим, что переход может быть не только для самого состояния, но и для его родительского состояния или вложенных в него состояний.

1.2. Ограничения при создании модели

В большинстве *UML*-редакторов используется алгоритм, который проверяет ограничения при попытке изменить модель и не разрешает изменение, какое-либо из ограничений было нарушено. В случае использования такого алгоритма, модель всегда находится в допустимом состоянии. При этом все ограничения, наложенные на эту модель, выполняются.

В этом есть как достоинства, так и недостатки. К достоинствам такого подхода можно отнести отсутствие необходимости строить множество проблемных элементов, в связи с их отсутствием. Так как построение этого множества требует некоторых временных затрат, то имеется ускорение алгоритма по сравнению с алгоритмами, поддерживающими противоречивое состояние модели. Однако, для проверки того, что очередное изменение не переведет модель в недопустимое со-

стояние, чаще всего осуществляется полная проверка всех правил. Это уменьшает экономию на построение множества проблемных элементов.

Рассмотрим недостатки ограничений во время построения модели. Возможны некоторые неудобства при редактировании модели, так как иногда удобнее построить модель, периодически переводя ее в недопустимое состояние. Другим минусом такого подхода является невозможность проверки ряда ограничений. Например, ограничения на достижимость всех состояний. Очевидно, что, например, построить автомат так, что бы в любой момент времени все его состояния были достижимы невозможно. Это объясняется тем, что во вновь добавленное состояние первоначально не ведет не один переход, а создать сначала переход, а потом состояние, или одновременно и переход, и состояние, невозможно. Теоретически, можно предусмотреть реализацию последнего варианта – одновременно создавать и переход и состояние, однако, редактирование такой модели нельзя будет назвать удобным.

Если же, не накладывать ограничения на построение модели – разрешать добавлять любые элементы, то для автоматной метамодели можно использовать только правила, касающиеся числа начальных состояний и переходов в них и из них (наличие одного начального состояния, отсутствие переходов в него, наличие ровно одного перехода из него и отсутствие переходов из конечного состояния). В то время как проверка менее тривиальных и, возможно, даже более полезных при построении модели правил, проверяющих ограничения на полноту переходов из состояния и достижимость всех состояний из начального состояния, невозможна. Поясним, почему сохранение модели в допустимом состоянии делает невозможным проверку полноты переходов из состояния. Если вспомнить, что для проверки полноты используются все переходы по некоторому событию, то становится ясным факт необходимости добавления сразу всех переходов по событию, а это аналогично добавлению перехода вместе с состоянием.

1.3. Полная проверка

Простейший алгоритм проверки правил, которые не накладывают ограничений на сами проверяемые правила, — проверка всех правил при любом изменении модели. Таким образом, при любом изменении модели вызывается проверка всех правил во всех допустимых контекстах. Например, в случае с автоматной моделью при добавлении перехода из состояния *S1* в состояние *S2* условия полноты и непротиворечивости переходов будут проверены для всех состояний. Причем проверка произойдет не только для состояний из того же автомата, что и состояния *S1* и *S2*, но для всех автоматов.

Данный алгоритм в отличие от алгоритма из разд. 1.2 не накладывает ограничений на типы правил и разрешает иметь модель с нарушенными ограничениями во время построения модели.

У этого алгоритма есть существенные недостатки. Все они сводятся к тому, что осуществляются лишние проверки при изменениях модели. Во-первых, при изменении некоторых атрибутов модели нет необходимости перепроверять какие-либо правила. Например, в случае автоматной метамодели, изменение имени состояния не при каких обстоятельствах не повлияет на выполнение правил. Во-вторых, правила проверяются во всех возможных контекстах, что излишне. Например, в случае автоматной метамодели, добавление перехода из состояния *S1* никак не повлияет на полноту и непротиворечивость переходов из состояния *S2*, однако, перепроверка будет выполнена.

1.4. Полная проверка с эвристиками

В инструментальном средстве *UniMod* версии 1.3 имелась эвристика, позволяющие устранить указанные недостатки алгоритма полной проверки. Атрибуты модели были разделены на «структурные» и нет. «Структурными» атрибутами будем называть атрибуты, которые изменяют структуру модели, а фактически это атрибуты, после изменения которых, возникает необходимость в перепроверке

хотя бы одного правила. Это усовершенствование позволяет избавить от части лишних проверок, однако, не решает проблемы выбора контекста.

Отметим также, что для проверки того, является ли атрибут «структурным», необходимо сохранять эту информацию вместе с самим атрибутом или отдельно. Это ведет к хранению дополнительных данных и разрастанию модели.

Выводы по главе 1

1. Приведено описание автоматной метамодели.
2. Выполнен краткий обзор трех известных алгоритмов динамической проверки непротиворечивости модели.
3. Выявлено, что эти алгоритмы осуществляют большое число лишних проверок, так как не учитывают контекст и тип изменения модели.

ГЛАВА 2. РЕШЕНИЕ ЗАДАЧИ

2.1. Постановка задачи

Необходимо выбрать удобный способ записи правил, описывающих ограничения метамодели, а также разработать алгоритм проверки состояний этих правил. Правило может находиться в одном из двух состояний («нарушено» или «выполняется») в зависимости от того нарушено или выполняется ограничение, накладываемое этим правилом. Разработанный алгоритм должен поддерживать корректное состояние для всех правил, наложенных на модель – обновлять состояние правила сразу же после соответствующих изменений модели. Под изменениями модели понимается изменения атрибутов элементов, входящих в модель, добавление новых элементов и удаление существующих элементов. Также при изменениях модели не должны перепроверяться те правила, на которые эти изменения не могли повлиять. Таким образом, при некоторых изменениях модели могут не проверяться никакие правила, что и приведет к более быстрой перепроверке правил на модели по сравнению с перепроверкой всех правил при любом изменении модели. Примером такого изменения для автоматной модели может служить переименование элемента модели.

2.2. Пример автомата

Для более наглядного описания алгоритма и ограничений на модель приведем пример задачи и решения этой задачи с помощью автоматного подхода. На полученном в результате автомате будут продемонстрированы ограничения на модель и работа алгоритма. Также будет проведена экспериментальная оценка числа операций, выполняемых при использовании разных подходов динамической проверки непротиворечивости автоматной метамодели.

Рассмотрим задачу управления простейшим кондиционером с таймером. Кондиционер может находиться в одном из двух состояний: включен или выклю-

чен. В состоянии «включен» есть два режима охлаждения: обычное охлаждение и интенсивное. У кондиционера имеется запрет на работу при температуре ниже 10 градусов по Цельсию (имеется встроенный термометр). Также имеется встроенный таймер, по которому кондиционер может включиться, выключить или сменить режим охлаждения.

Приведем решение этой задачи управления. Для этого необходимо ее формализовать и описать систему в терминах автоматного программирования – задать события, объекты управления, входные и выходные воздействия на них и автомат, который непосредственно обеспечивает управление. В системе имеется следующий набор событий:

e1 — включение кондиционера;

e2 — выключение кондиционера;

e3 — включение интенсивного режима охлаждения (при включенном кондиционере);

e4 — включение обычного режима охлаждения (при включенном кондиционере);

e5 — переход в режим настройки таймера;

e6 — возвращение из режима настройки таймера;

e7 — увеличение времени, через которое сработает таймер, на одну минуту;

e8 — уменьшение времени, через которое сработает таймер, на одну минуту;

e9 — выбор действия, которое будет совершено по срабатыванию таймера;

e10 — срабатывание таймера.

В устройстве имеются два объекта управления: термометр (*o1*) и таймер (*o2*). В объекте термометр существует всего одно выходное воздействие (*o1.x1*), возвращающее текущую температуру. Объект управления таймер не такой примитивный и имеет несколько входных и выходных воздействий, приведенных ниже:

o2.x1 — возвращает номер действия, которое необходимо выполнить по таймеру (включить кондиционер, переключить режим охлаждения или выключить);

o2.x2 — возвращает число минут, через которое должен сработать таймер;

o2.x3 — возвращает номер действия, которое должно быть установлено;

o2.z1 — увеличивает число минут до срабатывания таймера;

o2.z2 — уменьшает число минут до срабатывания таймера;

o2.z3 — подает сигнал о том, что произошла попытка установить отрицательное время срабатывания таймера;

o2.z11 — установка действия «включение кондиционера» по срабатыванию таймера;

o2.z12 — установка действия «смена режима охлаждения» по срабатыванию таймера;

o2.z13 — установка действия «выключение кондиционера» по срабатыванию таймера.

Приведенный на рис. 2 автомат решает задачу управления кондиционером.

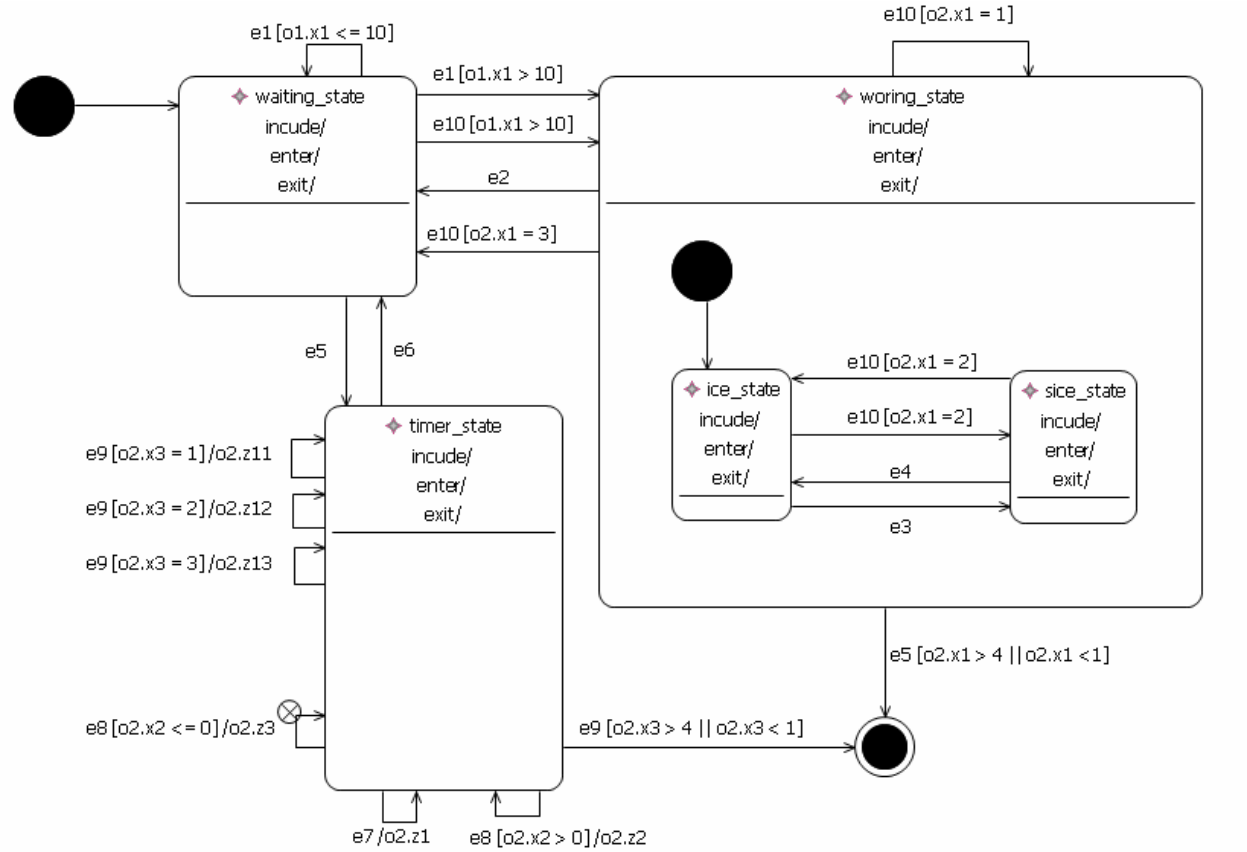

Рис. 2. Пример автомата

Автомат имеет восемь состояний (включая начальное и конечное), три из которых вложенные. Эти состояния управляют режимом охлаждения, когда конди-

ционер включен. Рассмотрим подробнее процесс работы кондиционера. В начале работы кондиционер находится в состоянии *waiting_state*, из которого можно перейти, либо в состояние *working_state*, описывающее работающий кондиционер, либо в состояние *timer_state*, в котором происходит установка параметров таймера (время срабатывания и действия). В случае ошибки, например, неизвестное действие по таймеру, автомат переходит в конечное состояние.

2.3. Ограничения на модель

Определим теперь правила, описывающие ограничения, которые должны соблюдаться в случае автоматной метамодели. Правила для описанной выше модели можно условно разделить на две группы — правила, описывающие ограничения, которые затрагивают только изменившееся состояние, и правила, описывающие ограничения, затрагивающие все состояния автомата, вне зависимости от того, какое состояние было изменено.

Ограничения для правила из первой группы, требующие рассмотрения только одного состояния, проверяются в контексте того состояния, для которого произошло изменение. В эту группу входит два правила.

1. Проверка полноты условий на переходах из состояния. Под полнотой условий подразумевается то, что для каждого события существует хотя бы один переход при любом значении переменных. В приведенном на рис. 3 автомате отсутствует переход для состояния *ice_state*, вложенного в состояние *working_state*, по событию *e10* и при условии *o2.x1 = 2*. Этот переход может начинаться как непосредственно в *ice_state*, так и в родительском состоянии, – в данном случае в *working_state.*

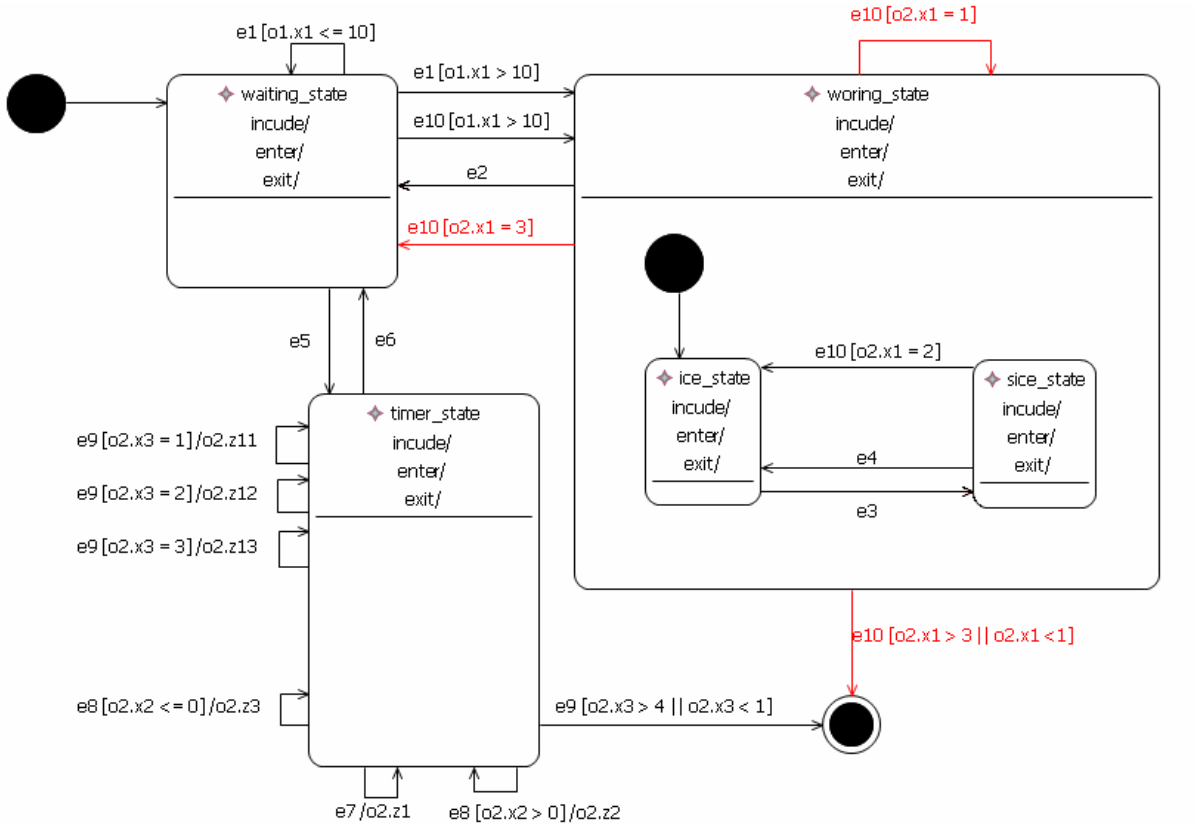

Рис. 3. Пример автомата, в котором нарушается ограничение полноты условий на переходах

Проверка непротиворечивости условий на переходах из состояния. Под непротиворечивостью условий подразумевается то, что для каждого набора значений переменных для любого события имеется только один допустимый переход. На автомате, изображенный на рис. 4, для состояний *ice_state*, *sice_state* и *working_state* есть переходы для одного и того же события (*e10*) при одном наборе значений переменных. В данном случае в условиях на переходах задействована только одна переменная *o2.x1*. Для состояния *ice_state* при *o2.x1 = 2* существует два перехода по событию *e10* — в состояние *sice_state* и в конечное состояние (переход в конечное состояние есть у родительского для *ice_state* состояния). Аналогично, для состояния *sice_state* при тех же условиях есть переходы в *ice_state* и в конечное состояние, а для состояния *working_state* имеются два перехода по событию *e10* при *o2.x1 = 3*.

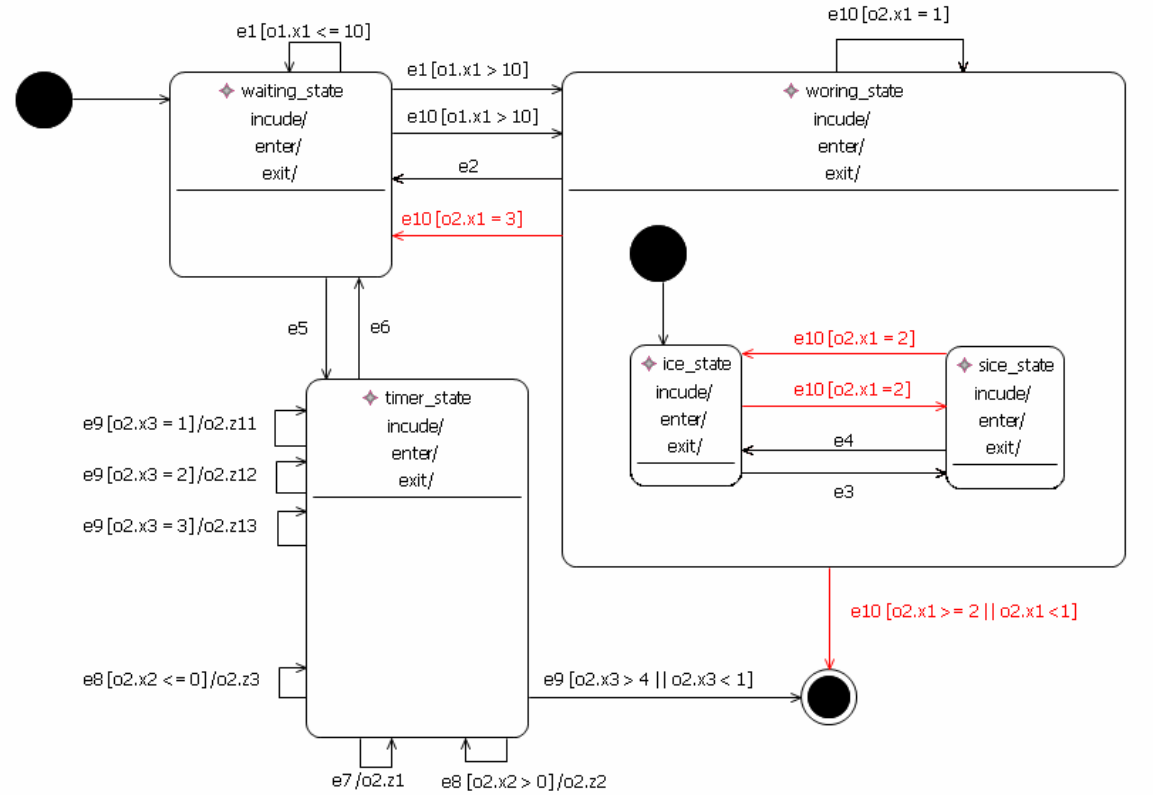

Рис. 4. Пример автомата, в котором нарушается ограничение непротиворечивости условий на переходах

Отметим, что состояния этих правил могут измениться только при добавлении и удалении переходов из состояния, а также при изменении условий на переходах.

Рассмотрим теперь правила второй группы – правила, описывающие ограничения, которые касаются всех состояний автомата. Правила этой группы будут проверяться в контексте всех состояний автомата. В группу входят следующие пять правил.

1. Наличие одного и только одного начального состояния в автомате. На рис. 5 можно видеть вариант автомата, нарушающего это правило:

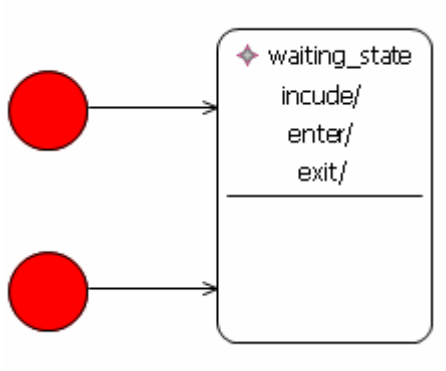

Рис. 5. Пример автомата, в котором нарушается ограничение наличия одного состояния в автомате

2. Наличие одного и только одного перехода из начального состояния. Напомним, что при запуске автомат находится в состоянии, в которое ведет переход из начального состояния. Таким образом, при наличии более чем одного перехода возникает неоднозначность – в каком из состояний начинать работу. Пример нарушения этого ограничения приведен на рис. 6.

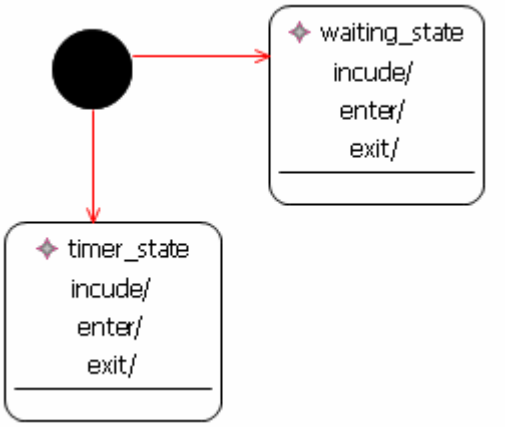

Рис. 6. Пример автомата, в котором нарушается наличия одного перехода из начального состояния

3. Отсутствие переходов в начальное состояние. Автомат никогда не находится в начальном состоянии (даже в начале работы автомат находится в состояние, в которое ведет переход из начального состояния), и поэтому перейти в это состояние по ходу работы нельзя. Пример автомата, нарушающего это правило, приведен на рис. 7.

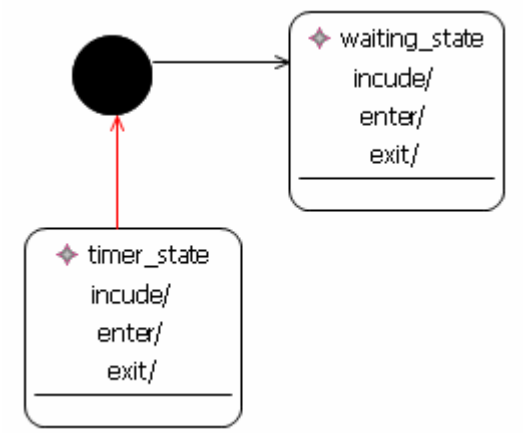

Рис. 7. Пример автомата, в котором нарушается отсутствия переходов в начальное состояние

4. Отсутствие переходов из конечного состояния. После перехода в конечное состояние автомат завершает работу. Поэтому переходы из конечного состояния никогда не будут задействованы.
5. Достижимость всех состояний автомата из начального состояния. Рассмотрим автомат как направленный граф [10]. Состояние *S* считается достижимым, если есть путь из начального состояния в состояние *S*. Например, в автомате на рис.8 отсутствует путь в состояние *timer_state*.

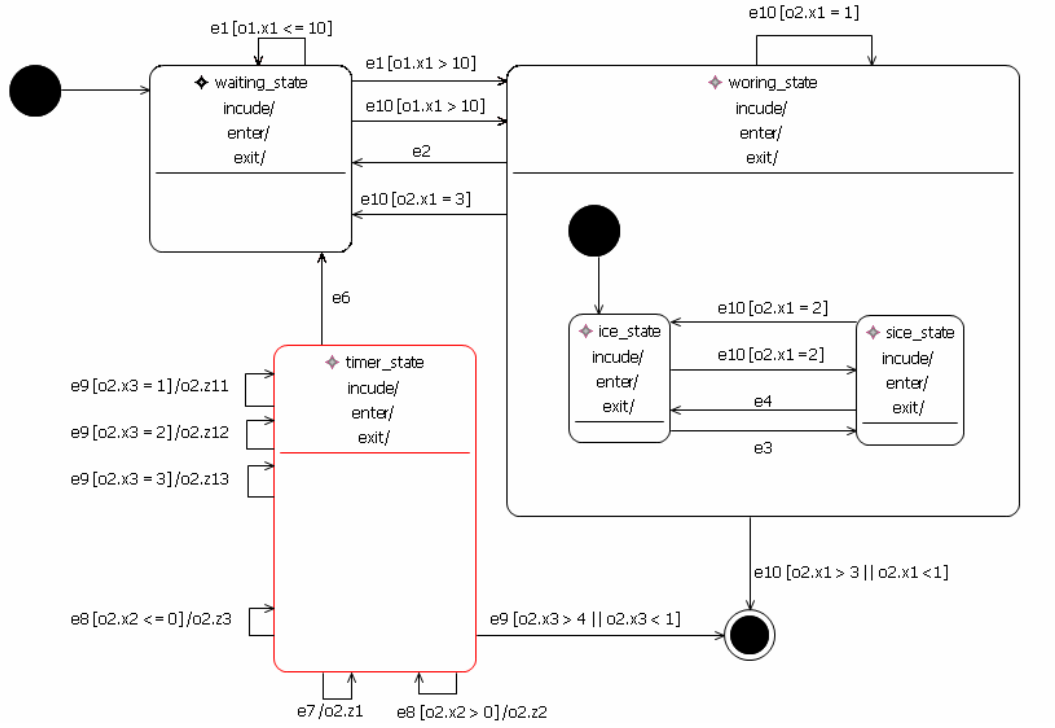

Рис. 6. Пример автомата, в котором нарушается достижимость всех состояний из начального состояния

Поясним, почему изменение некоторых атрибутов одного состояния может повлечь за собой нарушение приведенных выше правил этой группы. Определим для каждого правила события, при которых состояние правила может измениться.

Состояние первого правила зависит от числа начальных состояний в автомате и может быть изменено при следующих событиях:

- если было добавлено новое состояние в автомат, так как это состояние могло быть начальным, то необходимо проверить число начальных состояний в автомате;
- было удалено состояние из автомата, так как могло быть удалено начальное состояние;
- был изменен типа состояния (это событие может повлиять на указанные правила, так как исходно состояние могло быть начальным или после изменения типа оно стало начальным).

Правила два, три и четыре зависят от переходов между состояниями, а те, в свою очередь, могут быть удалены при удалении состояний. Таким образом, состояния этих правил могут измениться при следующих событиях: был добавлен переход (это событие может изменить состояния правил, так как переход мог быть из начального (конечного) состояния или в начальное состояние); был удален переход (это событие влияет на состояния правил по тем же причинам, что и предыдущее событие); удаление состояния, так как вместе с состоянием удаляются и все переходы, с ним связанные.

Состояние же пятого правила может измениться при добавлении и удалении переходов между состояниями (так как они и определяют достижимость состояния) и добавлении новых состояний в автомат.

2.4. Алгоритм

Рассмотрим теперь основные идеи алгоритма динамической проверки этих правил. Условно алгоритм можно разбить на два этапа: этап предварительных вы-

числений и этап динамической перепроверки правил. Первый этап выполняется один раз за выполнение алгоритма, второй – при каждом изменении модели.

Как было описано выше, первый этап – это предварительные вычисления, необходимые для быстрого определения правил, состояние которых могло измениться в результате изменения модели. Напомним, что под состоянием правила подразумевается одно из двух: ограничение, описанное правилом, выполняется или не выполняется. Так как за время работы с метамоделью первый этап выполняется всего один раз, то можно значительно не ограничивать трудоемкость этого этапа. Предлагается сопоставить всем атрибутам элементов модели правила, которые от них зависят. Для вычисления всех правил, зависящих от атрибута, проведем анализ всех правил. При этом будем добавлять правила в список, связанный с атрибутом, в том случае, если этот атрибут используется в правиле. Таким образом, будет получен список атрибутов, к каждому из которых привязан список правил, зависящих от атрибута. Также на этом этапе для каждого правила необходимо вычислить контекст, в котором правило должно проверяться.

На втором этапе составляются множества правил, которые необходимо перепроверить после изменения модели, и выполняется их проверка. Так как этот этап выполняется многократно за время жизни модели, то его работа должна быть максимально эффективна с точки зрения времени работы. Оптимальное время работы достигается за счет проверки только тех правил, чье состояние могло поменяться при изменении того или иного атрибута. Список таких правил был составлен на первом этапе работы алгоритма. Поэтому множество правил, чье состояние необходимо обновить при изменении того или иного атрибута, может быть вычислено за $O(k)$ [10], где k — число атрибутов, при изменении которых требуется пересчитать хотя бы одно правило. Однако при оптимальном хранении этих списков становится возможным получить список правил по атрибуту за $O(1)$.

Также отметим, что проверять выполнение правила необходимо в контексте произошедшего изменения модели, а не для всех таких элементов. Например, если был добавлен переход из состояния, то необходимо перепроверять правила о

полноте и непротиворечивости только для этого состояния, а не всех состояний автомата. Так как контекст, в котором необходимо перепроверять правила был вычислен на первом этапе, то это условие не уменьшит эффективность второго этапа алгоритма. Отметим, что в случае автоматной метамодели в случае любых изменений будет проверено *O(1)* правил. Исключение составляют случаи проверки полноты и непротиворечивости условий на переходах, для состояний с вложенными состояниями. В этом случае число проверяемых правил будет пропорционально числу вложенных состояний.

Также была проведена аналитическая оценка времени работы проверки каждого правила. Пусть m — число автоматов в модели; k — число состояний во всех автоматах; k_j — число состояний в j-ом автомате; t — суммарное число переходов; t_j — суммарное число переходов в j-ом автомате; t_i — число переходов из i-го состояния. В табл. 1 приведена асимптотическая оценка времени проверки каждого правила.

Таблица 1. Оценка времени проверки правил

Правило	Время работы
Одно начальное состояние	$O(k_j)$
Один переход из начального состояния	$O(k_j)$
Отсутствие переходов в начальное состояние	$O(k_j)$
Отсутствие переходов из конечного состояния	$O(k_j)$
Достижимость всех состояний	$O(k_j+t_j)$
Полнота условий на переходах	$O(k_j t_i)$

Непротиворечивость условий на переходах	$O(k_j t_i)$

Таким образом, во время динамической проверки непротиворечивости метамодели будут проверяться только правила, состояние которых могло измениться, и только в том контексте, в котором произошло изменение модели, которое повлекло необходимость перепроверки правил. Как будет показано далее, это позволит существенно улучшить время работы по сравнению с алгоритмами, которые перепроверяют все правила на всех элементах модели при любом изменении этой модели.

Выводы по главе 2

1. Сформулирована задача, решаемая в работе.
2. Описаны ограничения на автоматные модели.
3. Приведен пример автоматной модели, на котором показаны примеры нарушения всех описанных ранее ограничений на автоматную модель.
4. Описан новый алгоритм динамической проверки непротиворечивости автоматной модели. Полученный алгоритм учитывает как тип события, так и контекст, в котором произошло событие.

ГЛАВА 3. РЕАЛИЗАЦИЯ И ВНЕДРЕНИЕ АЛГОРИТМА ДИНАМИЧЕСКОЙ ПРОВЕРКИ НЕПРОТИВОРЕЧИВОСТИ АВТОМАТНОЙ МОДЕЛИ

3.1. Метамодель и ограничения

При реализации описанная выше автоматная метамодель была выражена с помощью фреймворка *EMF* (*Eclipse Modeling Framework*) [11].

Для записи ограничений метамодели предлагается использовать *OMG OCL* (*Object Constraint Language*) [12]. Этот язык был разработан в корпорации *IBM* специально для описания ограничений модели, поэтому хорошо подходит для решаемой задачи. При реализации рассматривались три фреймворка для работы с *OCL*: *Eclipse OCL* [13] — реализация *OCL* написанная в сообществе *Eclipse*; *Kent OCL* [14] — реализация *OCL*, разработанная в университете города Кент; *Dresden OCL Toolkit* [15] — реализация *OCL*, разработанная в Дрезденском Техническом Университете.

Далее будет приведено сравнение этих трех фреймворков, и на его основании будет выбран фреймворк для выражения всех правил для автоматной метамодели рассматриваемых в этой работе. Основными характеристиками при сравнении фреймворков являются эффективность работы фреймворка (время работы), множество поддерживаемых типов правил, простота использования и возможность интеграции с *EMF*. Последняя характеристика крайне важна, так как при невозможности такой интеграции возникает проблема использования этого фреймворка совместно с описанной в этой работе реализацией автоматной метамодели.

Первым рассмотрим *Dresden OCL Toolkit*. Последняя версия этого фреймворка полностью соответствует спецификации *OCL2*. Это означает, что поддерживаются все виды правил, которые можно выразить с помощью *OCL2*. Однако есть существенный минус, который делает использование *Dresden OCL Toolkit* невозможным при решении поставленной задачи, — в этом фреймворке используется

собственная реализация моделей. Таким образом, интеграция с *EMF* становится почти невозможной. Решением проблемы проверки ограничений с помощью *Dresden OCL Toolkit* может быть перевод модели, которая описана с помощью *EMF*, в собственную модель *Dresden OCL Toolkit*. Таким образом, потребуется переводить модель из одного представления в другое после каждого изменения, которое требует перепроверку каких-либо правил, что является трудоемкой задачей и значительно снизит эффективность алгоритма. Другим решением данной проблемы будет поддержание модели, которая описана с помощью *Dresden OCL Toolkit* параллельно основной модели. Это решение также не является эффективным, так как требует хранения избыточной информации, а также дополнительной работы (необходимо вносить изменения во вторую модель при каждом изменении оригинальной модели описанной на *EMF*). Таким образом, значительное уменьшение эффективности алгоритма перепроверки правил делает использование *Dresden OCL Toolkit* нежелательным.

Проведем сравнение *Kent OCL* и *Eclipse OCL*. Оба этих фреймворка работают на *EMF* моделях [16, 17], что решает вопрос интеграции с реализацией автоматной метамоделью описанной в этой работе. Также как и *Dresden OCL Toolkit* последние версии данных реализаций соответствуют спецификации *OCL2* и поддерживают необходимые действия, такие как проверка ограничения и запросы к модели. Однако последняя версия фреймворка *Eclipse OCL* еще не поддерживает динамическую проверку ограничений на модели. Таким образом, нет возможности использовать свой алгоритм динамического построения множества правил, которые необходимо перепроверить при изменении модели. Использование же предыдущей версии *Eclipse OCL* не рекомендовано сообществом *Eclipse*.

Фреймворк созданный в университете города Кент лишен указанных выше недостатков. Это стало решающим фактом при выборе его для проверки правил при решении поставленной задачи.

Напомним, какие ограничения имеют место в случае автоматной модели, и приведем их запись на *OCL*:

1. Одно начальное состояние в автомате:

```
context coremodel::State inv:
   self.substates->select(x|x.initial)->size()=1
```

2. Один переход из начального состояния:

```
context coremodel::State inv:
   self.substates-
>select(x|x.initial).outgoingTransitions->size()=1
```

3. Отсутствие переходов в начальное состояние:

```
context coremodel::State inv:
   self.substates->select(x|x.initial).
   incomingTransitions->size()=0
```

4. Отсутствие переходов из конечного состояния:

```
context coremodel::State inv:
   self.substates-
>select(x|x.final).outgoingTransitions->size()=0
```

5. Достижимость всех состояний автомата из начального состояния.

```
context coremodel::State inv:
   let reachableStates =
      self.substates->iterate(
```

```
         s;
y:Set(coremodel::State)=self.initialSubstate->asSet()
|
         y->union(
            y->collect(
               e | e.outgoingTransitions->collect(t |
t.targetState)
            )
         )
      )->asSet() in
   self.substates->iterate(s; re-
sult:Set(coremodel::State)=Set{} |
      if reachableStates.includes(s) then
         result
      else
         result->union(s->asSet())
      endif
   )->size() = 0
```

6. Полнота условий на переходах из состояния.

```
context coremodel::State inv:
   self.outgoingTransitions->collect(x | x.event)-
>union(
      self.superstate.outgoingTransitions->collect(x
| x.event)
   )->asSet()->iterate(x;y=true |
      y and self.isComplete(
```

```
         self.outgoingTransitions->select(z | z.event
= x)->
            collect(z | z.guard)->union(
               self.superstate.outgoingTransitions->
               select(z | z.event = x)->collect(z |
z.guard)
            )
         )
      )
```

7. Непротиворечивость условий на переходах из состояния.

```
context coremodel::State inv:
   self.outgoingTransitions->collect(x | x.event)->
   union(
      self.superstate.outgoingTransitions->
      collect(x | x.event)
   )->asSet()->
   iterate(x;result=true |
      let transitions =
         self.outgoingTransitions->select(z | z.event
= x)-> union(
            self.superstate.outgoingTransitions->
            select(z | z.event = x)
         ) in
      result and transitions->iterate(t1; y1=true |
         y1 and transitions->iterate(t2; y2=true |
            y2 and (isConsistent(t1.guard, t2.guard)
or t1 = t2)
```

```
                )
            )
        )
```

Отметим, что, не смотря на то, что первые шесть правил затрагивают не одно состояние, а автомат целиком, контекстом для ограничений, которые описаны этими правилами, является состояние. Это связано с представлением автомата в нашей модели. В этом представлении все состояния автомата вложены в *top*-состояние автомата. Таким образом, эти правила проверяются в контексте родительского состояния для состояния, в котором произошло изменение модели.

3.2. Алгоритм

Как описано в разд. 2.4, на первом этапе алгоритма для каждого атрибута модели составляется список правил, состояние которых могло измениться при изменении этого атрибута. Эти списки будут храниться в структуре данных *HashMap* [10], время, затрачиваемое на поиск элемента в такой структуре данных, есть *O(1)*. В качестве ключа будет использована пара имя класса и название атрибута, значением же будет список правил.

При изменении модели из структуры данных, заполненной на первом этапе, извлекается список правил, которые необходимо перепроверить, если этот список не пуст, то продолжается дальнейшее выполнение второго этапа. Если выполнение было продолжено, то с помощью фреймворка *Kent OCL* осуществляется перепроверка всех правил из полученного списка. Таким образом, для каждого правила из списка будет получено его новое состояние.

3.3. Внедрение в *UniMod*

Рассмотрим внедрение такого метода проверки непротиворечивости модели в средство разработки *UniMod*. Проверка непротиворечивости модели осуществляется в расширении к среде *Eclipse* [7, 16]. Это расширение выполняет три основ-

ных задачи: первая — отслеживание изменений модели; вторая — перепроверка набора правил, который соответствует произошедшему в модели изменению; третья — визуальное отображение нарушенных ограничений.

Произведем более детальное рассмотрение реализации этих трех частей, а также этапа инициализации расширения отвечающего за проверку непротиворечивости модели. Этап инициализации соответствует первому этапу в алгоритме, описанном в разд. 2.4, в течение дальнейшей работы расширения выполняется второй этап алгоритма.

Рассмотрим этап инициализации. На этом этапе происходит создание *HashMap*'а с парами событие — набор правил, которые необходимо перепроверить по этому событию. Вместе с правилом хранится дополнительная информация, которая необходима для выполнения проверки этого правила в контексте произошедшего события. Такой информацией являются следующие три запроса на языке *OCL*. Первый запрос — запрос для проверки ограничения. Второй запрос — запрос для определения элемента, на котором надо выполнять запрос для проверки ограничения. Третий запрос выбирает элементы, на которых было нарушено ограничение, которое описано правилом (если ограничение было нарушено). Таким образом, теперь будем называть правилом не один запрос, а совокупность трех перечисленных запросов. Отметим также, что для разных событий (изменения разных атрибутов модели) будут храниться разные правила. Эти правила будут отличаться запросом, который определяет элемент, на котором надо проверять ограничение.

Для отслеживания изменений модели используется механизм точек расширения (extension points) [18]. В ядре инструментального средства *UniMod* имеется точка расширения для подключения «слушателей» изменений модели. Расширение, которое осуществляет проверку непротиворечивости метамодели, регистрирует с помощью этой точки расширения своего «слушателя», который расположен в классе *com.unimod.validation.KentAdaptor*. При изменениях модели вызывается метод *notifyChanged*, из которого, в свою очередь, и вызывается метод, кото-

рый непосредственно осуществляет проверку ограничений. Напомним, что ограничения проверяются только для события, которое было передано в метод *notifyChanged*, то есть перепроверяются только те правила, состояние которых могло измениться при изменении атрибута модели.

При изменении некоторого атрибута модели необходимо перепроверить часть ограничений, наложенных на модель. При изменении модели, в метод, который осуществляет перепроверку ограничений, передается следующая информация. Объект, который был инициатором события, то есть объект, атрибут которого был изменен, и сам измененный атрибут. Этой информации будет достаточно, чтобы провести перепроверку только тех правил, состояние которых могло измениться. Для этого из структуры данных, которая была создана на этапе инициализации, достается набор правил, который соответствует произошедшему изменению модели. Для каждого правила запускается запрос, который определяет контекстный элемент. Этот запрос запускается на том элементе, который был изменен и стал причиной обрабатываемого события. После этого в зависимости от результата проверки ограничения соответствующим образом обновляется список нарушенных ограничений и набор элементов, которые необходимо отметить визуально, как проблемные.

В случае автоматной метамодели запросы для выбора контекстного элемента достаточно просты, чаще всего это выбор родительского состояния или состояния, атрибут которого был изменен. Выбор проблемных элементов для правил не столь тривиален. Приведем запросы, осуществляющие выбор проблемных элементов:

1. Пусть нарушено правило о наличии одного начального состояния в автомате, проблемными являются все начальные состояния:

```
context coremodel::State
inv: self.substates->select(x|x.initial)
```

2. В случае наличия более чем одного перехода из начального состояния выбираются все переходы из начального состояния:

```
context coremodel::State
inv: self.substates->select(x|x.initial)
```

3. Если в начальное состояние ведут переходы, то они являются проблемными:

```
context coremodel::State
inv:                                    self.substates-
>select(x|x.initial).incomingTransitions
```

4. Аналогично все переходы, которые ведут из конечного состояния являются проблемными:

```
context coremodel::State
inv:                                    self.substates-
>select(x|x.final).outgoingTransitions
```

5. Все состояния недостижимые из начального состояния также являются проблемными:

```
context coremodel::State inv:
   let reachableStates =
      self.substates->iterate(
         s; y:Set(coremodel::State)=
            self.initialSubstate->asSet() |
         y->union(
```

```
            y->collect(
               e | e.outgoingTransitions->collect(t |
t.targetState)
            )
         )
      )->asSet() in self.substates->iterate(s; re-
sult:Set(coremodel::State)=Set{} |
      if reachableStates.includes(s) then
         result
      else
         result->union(s->asSet())
      endif
   )
```

6. Состояния, для которых условия на переходах не описывают все возможные случаи (то есть, при некотором значении переменных, нет перехода из состояния для одного из событий), не удовлетворяют условию полноты и являются проблемными:

```
context coremodel::State inv:
   self.outgoingTransitions->collect(x | x.event)->
   union(
      self.superstate.outgoingTransitions->collect(x
| x.event)
   )->asSet()->
   iterate(x;result:Set(coremodel::Transition)=Set{}
|
      let transitions =
```

```
         self.outgoingTransitions->select(z | z.event
= x)->
         union(
            self.superstate.outgoingTransitions->
            select(z | z.event = x)
         ) in
      if self.isComplete(transitions->collect(z |
z.guard)) then
         result
      else
         result->union(transitions->asSet())
      endif
   )
```

7. Переходы из одного состояния, которые имеют противоречивые условия для одного события (то есть условия, которые могут одновременно выполняться при некотором наборе переменных, что делает автомат недетерминированным), также являются проблемными:

```
context coremodel::State inv:
   self.outgoingTransitions->collect(x | x.event)->
   union(
      self.superstate.outgoingTransitions->collect(x
| x.event)
   )->asSet()->
   iterate(x;result:Set(coremodel::Transition)=Set{}
|
      let transitions =
```

```
          self.outgoingTransitions->select(z | z.event
= x)->
          union(self.superstate.outgoingTransitions->
             select(z | z.event = x)
          ) in
       result->union(transitions->iterate(t1;
          y:Set(coremodel::Transition)=Set{} |
             y->union(transitions->select(
                t2 | (not isConsistent(t1.guard,
t2.guard)) and t1 <> t2
                )
             )
          )
       )
   )
```

На первый взгляд может показаться, что запросы, возвращающие проблемные элементы, можно совместить с запросами, которые осуществляют проверку ограничений. Определять нарушается правило или нет, можно было бы, проверив наличие проблемных элементов. Однако обратим внимание на запрос осуществляющий поиск проблемных элементов для ограничений на число начальных состояний. В случае если в автомате одно начальное состояние, то ограничение не нарушено, при этом запрос вернет одно начальное состояние. Этот случай не единственный, можно построить и другие ограничения, с которыми возникнут схожие проблемы, поэтому построение множества проблемных элементов требует дополнительных запросов.

Теперь рассмотрим подробно процесс построения списка проблемных элементов. Отметим, что при каждом изменении состояний правил необходимо формировать список проблемных элементов для всех правил, а не только для тех, чье

состояние изменилось. Это обусловлено особенностями реализации визуального отображения проблем в диаграммах в *UniMod* – метод, который отвечает за эту визуализацию, принимает список всех проблемных элементов. Отметим, что перевычисление всех проблемных элементов при изменении состояний правил нецелесообразно. Предлагается сохранять ошибки после предыдущих проверок состояний правил. Однако хранить проблемные элементы простым списком не представляется возможным, по причине невозможности обновления этого списка. Например, отсутствует возможность исключения проблемных элементов из списка при изменении состояния правила на "выполняется". Для поддержания простого обновления этого списка предлагается привязывать проблемные элементы к правилам, которые в них нарушаются.

Покажем, что для более эффективного обновления проблемных элементов необходимо усложнить структуру, в которой хранятся проблемные элементы. Очевидно, что одно правило может нарушаться в контексте сразу многих элементов, а в нашем случае состояние правила изменяется только в контексте одного элемента. Соответственно необходимо добавлять в список (или удалять из списка) только те проблемные элементы, которые касаются изменения правила в контексте конкретного элемента. Для решения этой задачи предлагается хранить для каждого правила не список проблемных элементов, а набор элементов, в контексте которых нарушается правило. Вместе с каждым таким контекстным элементом будем хранить список элементов, которые являются проблемными. То есть список элементов с нарушенным ограничением, к которому прикреплен контекстный элемент.

Алгоритм обновления описанной выше структуры значительно проще, чем просто списка проблемных элементов. Пусть некоторое правило *X* после проверки в контексте элемента *E* поменяло свое состояние на "выполняется", тогда из набора контекстных элементов соответствующих правилу *X* будет удален элемент *E*, а также список проблемных элементов прикрепленных к элементу *E*. Теперь пусть правило Y изменило свое состояние на "нарушено" в контексте элемента *E*. Доба-

вим элемент *E* в набор контекстных элементов, связанных с правилом Y. А к элементу *E* прикрепим список из проблемных элементов, которые вычисляются с помощью третьего и запросов, хранящихся в правиле.

После проверки всех правил ассоциированных с произошедшим изменением модели производится визуальное отображение проблемных элементов. Для этого производится обход описанной выше структуры и формируется список проблемных элементов. Далее этот список передается в расширение *UniMod*'а, которое отвечает за визуализацию диаграмм. На диаграмме проблемные элементы подсвечены красным, также добавляются сообщения в стандартный список проблем *Eclipse*.

Выводы по главе 3

1. Проведено сравнение различных фреймворков для работы с *OCL* [12–14].
2. Выбран фреймворк для внедрения реализации алгоритма в инструментальное средство *UniMod*.
3. Все ограничения описаны на языке *OCL*. Описаны запросы для вычисления проблемных элементов для нарушенного ограничения.
4. Описано внедрение алгоритма проверки непротиворечивости автоматной модели в инструментальное средство *UniMod*.

ГЛАВА 4. СРАВНЕНИЕ АЛГОРИТМОВ ДИНАМИЧЕСКОЙ ПРОВЕРКИ НЕПРОТИВОРЕЧИВОСТИ АВТОМАТНОЙ МОДЕЛИ

Экспериментальное сравнение алгоритмов

Проведем сравнительный анализ алгоритмов полной проверки, полной проверки с эвристикой и алгоритма, который описан в данной работе. Для этого оценим число правил, которые были проверены при построении автомата из примера. При этом обратим внимание на то, в какие моменты делались лишние проверки. Также убедимся, что при использовании описанного в этой работе алгоритма число лишних проверок будет сведено к минимуму.

В табл. 2 приведены результаты замеров, выполненных при построении автомата, управляющего работой кондиционера. При этом определялось число событий, которое было обработано алгоритмом, и число проверок, которые выполнил алгоритм.

Таблица 2. Экспериментальная оценка числа проверяемых правил

Алгоритм[1]	Число событий	Число обработанных событий	Число проверок ограничений	Общее число проверок
A	322	322	6987	6987
B		160	3498	3498
C		160	398	796

[1] В графе «Алгоритм» алгоритмы имеют следующие обозначения:

- A — алгоритм полной проверки;
- B — алгоритм полной проверки, с эвристикой по структурным правилам;
- C — алгоритм, рассматриваемый в этой работе;

Для удобства рассмотрения этих данных представим их на графике (рис. 9).

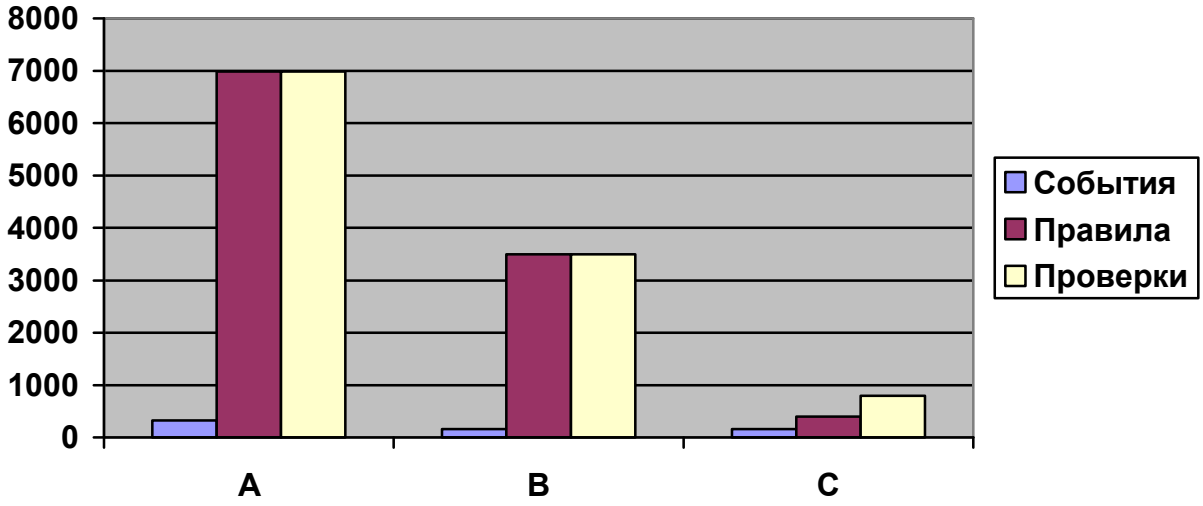

Рис. 9. Экспериментальная оценка числа проверяемых правил

Как видно из диаграммы, число проверяемых правил в случаях алгоритмов полной проверки и полной проверки всех правил при структурных изменениях модели значительно превышает число правил, проверяемых описанным в этой работе алгоритмом. Если быть точнее, то для алгоритма полной проверки проверяется в 17,5 раз больше правил, а для алгоритма полной проверки с эвристикой по структурным изменениям — в 4,4 раза.

Покажем, что эти числа не являются константами и зависят от размеров модели. Для этого проведем анализ того, откуда берутся лишние проверки в первых двух случаях.

Разница в числе правил, проверяемых алгоритмами полной проверки без учета типа события и с ним, появляется за счет увеличения числа событий, по которым была осуществлена эта проверка. Действительно, число событий обработанных первым алгоритмом в два раза больше, поэтому число проверенных правил тоже больше примерно в два раза. Отсюда можно сделать вывод, что число проверяемых правил прямопропорционально число событий, на которые эта проверка была запущена. Это действительно так, ввиду того, что на каждое событие осуществляется полная проверка всех правил во всех контекстах. Из-за того, что число контекстов изменяется при построении модели (например, добавляются новые

состояния), то отношение числа проверенных первым алгоритмом правил к числу правил, проверенных вторым алгоритмом, не равно отношению числа обработанных событий в первом и втором случаях.

Рассмотрим теперь разность в числе проверенных правил в случаях алгоритма, осуществляющего проверки только при структурных изменениях модели, и алгоритма рассматриваемого в этой работе. Отметим, что число обработанных событий в данном случае не играет роли, так как оно равно для обоих алгоритмов. Таким образом, влияют следующие два отличия рассматриваемого в этой работе алгоритма. Во-первых, проверка не всех правил при изменении – по некоторому изменению модели, перепроверяется состояние только тех правил, у которых оно могло измениться. Во-вторых, в рассматриваемом в этой работе алгоритме правила перепроверяются только в том контексте, в котором произошло изменение.

Напомним, что это значит, например, для автоматной модели и правил полноты и не противоречивости состояния. Пусть в модель был добавлен переход из состояния *S1* в состояние *S2*, а также в ней имеются состояния *S3* .. *S10*. В случае алгоритмов полной проверки выполнение правил полноты и непротиворечивости будет проверено для всех десяти состояний *S1*.. *S10*. Алгоритм, предложенный в этой работе, осуществит проверку только в контексте произошедшего изменения – для перехода из состояния *S1* в состояние *S2*. Таким образом, правила полноты и непротиворечивости будут проверены только для состояния *S1*.

Определим теперь основные недостатки и достоинства алгоритма, рассматриваемого в этой работе, по сравнению с алгоритмами, которые не допускают состояний модели с нарушением ограничений, а также по сравнению с алгоритмами полной проверки.

По сравнению с алгоритмами, не допускающими состояний модели с нарушенными правилами, алгоритм, рассматриваемый в этой работе, имеет те же основные преимущества и недостатки, что и все алгоритмы, поддерживающие состояния модели с нарушенными правилами. Рассмотрим в начале недостатки такого подхода. Основным из них является большее в общем случае время работы.

Это связано с тем, что при нарушении ограничений возникает необходимость вычисления элементов модели, на которых эти ограничения были нарушены. Таким образом, при нарушении правила не просто отменяется последнее изменение, а происходит вычисление еще одного правила, которое как раз и находит элементы, на которых было нарушено ограничение.

Однако имеется и ряд серьезных преимуществ. Во-первых, наглядность построения модели. При некоторых изменениях может быть непонятно, какое ограничение было нарушено, и почему изменение было отменено алгоритмом проверки. В случае же алгоритмов поддерживающих состояние модели с ошибками, элементы являющиеся проблемными будут выделены, что позволит быстрее найти ошибку в модели, так как возможно она была внесена не последним изменением, а каким-то из предыдущих изменений. Однако те изменения не нарушали правила наложенные на модель, а последнее изменение их нарушило. Вторым преимуществом является возможность демонстрации того, как те или иные изменения модели выведут ее из допустимого состояния. Этот факт может быть использован, например, в учебных целях.

Основным же достоинством предлагаемого алгоритма является поддержка более широкого класса правил. Для алгоритмов, которые не допускают состояний модели с нарушенными ограничениями, существуют правила, которые невозможно использовать. Эти правила чаще всего неоднократно нарушаются при построении модели. Таким образом, возникает проблема наличия ограничений, которые невозможно проверить с помощью рассматриваемых алгоритмов. Например, к таким правилам, для автоматной метамодели, можно отнести правило достижимости всех состояний из начального состояния. В некоторых случаях, проблему можно решить, допуская комплексные изменения модели. Таким образом, изменения, при которых изменяется более одного атрибута модели за один раз или добавляется более одного элемента. Если вернуться к ограничению на достижимость состояний, то таким комплексным изменением могло бы быть добавление перехода из состояния вместе с состоянием, в которое осуществляется переход.

Однако, например, для ограничения на полноту условий на переходах по некоторому событию такое комплексное изменение ввести невозможно. Для такого правила потребовалось бы единовременное добавление всех переходов по событию из состояния.

В целом редактирование модели при недопустимости состояний с нарушенными ограничениями лишает свободы действий. Это иногда бывает достаточно неудобным, даже если не учитывать правила, проверка которых становится невозможной. Это связано с тем, что иногда удобнее и быстрее вывести модель из допустимого состояния, чтобы с помощью некоторых изменений вернуть ее в допустимое состояние, но уже другое.

Проведем теперь сравнение с алгоритмами полной проверки. Есть смысл рассматривать только алгоритм проверки с учетом типа событий, так как он более эффективен, нежели просто алгоритм полной проверки. Напомним, что его единственным недостатком в сравнении с алгоритмом без учета типа событий является необходимость хранения типа для атрибутов модели – являются ли они структурными или нет. Однако, память необходимая, для сохранения этого свойства незначительна, а преимущество в числе проверяемых правил, в случае учета типа событий, столь велико, что нет смысла сравнивать алгоритм, рассматриваемый в этой работе, с алгоритмом полной проверки без чета типа событий.

Выделим основные недостатки рассматриваемого в данной работе алгоритма, в сравнении с алгоритмом полной проверки с учетом типа событий, который в настоящее время используется в инструментальном средстве *UniMod*. Основным минусом является необходимость вычислять контекст, в котором будут проверяться правила. Так как вычисление контекста происходит при каждой проверке ограничений, то получается, что число правил, которые необходимо вычислить, для проверки непротиворечивости метамодели, удваивается.

Рассмотрим теперь преимущества нового алгоритма. Вычисление контекста имеет также и положительную сторону, так как состояние правил вычисляется заново только для тех элементов модели, для которых оно могло измениться. Это

дает существенное уменьшение число проверок, для больших моделей. Например, в случае, автоматных моделей, при изменении модели, касающегося состояний (изменение типа состояния или добавления перехода) правила будут перепроверены только в контексте этого состояния, а не всех. В серьезных же моделях число состояний может исчисляться десятками. Аналогично ограничения, наложенные на автомат (например, достижимость всех состояний из начального состояния), будут перепроверены только в контексте автомата, в котором произошло изменение модели, а не всех автоматов, что тоже позволяет значительно сократить число проверок.

Также отметим, что в новом алгоритме при изменении модели перепроверяются не все правила, а только те, чье состояние может быть затронуто произошедшим изменением. Однако для этого не сохраняется какая-то дополнительная информация в модели, как в случае со структурными и не структурными изменениями. Определение того, затронет ли изменение состояние правила, происходит на основе самого правила. Напомним, что построение соответствий между типами изменений и правилами происходит на первом этапе алгоритма, то есть один раз за время работы с моделью.

Выводы по главе 4

1. Проведено экспериментальное сравнение существующих и предложенного в настоящей работе алгоритмов проверки непротиворечивости модели.
2. Сформулированы достоинства и недостатки нового алгоритма по сравнению с существующими алгоритмами.

ЗАКЛЮЧЕНИЕ

В данной работе получены следующие результаты.

1. Построена автоматная метамодель. Выделены основные элементы этой модели и описаны ограничения на модели.
2. Ограничения на модель выражены на языке *OCL*. Это сделано для удобства проверки непротиворечивости автоматной метамодели.
3. Исследован подход по поддержанию модели в допустимом состоянии (не противоречащим метамодели).
4. Исследованы существующие алгоритмы проверки непротиворечивости моделей. Также проведено сравнение этих алгоритмов между собой для выявления их положительных и отрицательных сторон.
5. Предложен новый алгоритм динамической проверки правил непротиворечивости автоматной модели. Предложенный алгоритм значительно уменьшает число правил, которые необходимо проверить при изменении модели, за счет учета контекста и сопоставления атрибутов модели и правил.
6. Предложенный алгоритм реализован и внедрен в инструментальное средство *UniMod 2*. Построены *OCL*-запросы для поиска проблемных элементов при нарушении ограничений.
7. Произведено экспериментальное сравнение существующих и предложенного алгоритмов динамической проверки правил непротиворечивости модели.

Возможны следующие пути развития работы.

1. Автоматическое построение *OCL*-запросов для выявления проблемных элементов при нарушении ограничения.
2. Автоматическое построение *OCL*-запросов для вычисления контекстных элементов для проверки ограничений на них.

ИСТОЧНИКИ

1. *Гома Х.* UML. Проектирование систем реального времени, распределенных и параллельных приложений. М.: ДМК, 2002.
2. *Буч Г.*, *Рамбо Д.*, *Джекобсон А.* Язык UML. Руководство пользователя. М.: ДМК, 2000.
3. *Larman C.* Applying UML and Patterns. Prentice Hall PTR, 1997.
4. *Mellor S.*, *Scott K.*, *Uhl A.*, *Weise A.*, Introduction to Model Driven Architecture. Addison-Wesley, 2003.
5. *Шалыто А. А.* Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/switch/6
6. *Гуров В. С.*, *Нарвский А. С.*, *Шалыто А. А.* Автоматизация проектирования событийных объектно-ориентированных программ с явным выделением состояний //Труды X Всероссийской научно-методической конференции "Телематика-2003". СПб.: СПбГИТМО (ТУ). 2003. http://tm.ifmo.ru.
7. *Гуров В. С.*, *Мазин М. А.* Веб-сайт проекта *UniMod*. http://unimod.sourceforge.net/
8. *Гуров В. С.*, *Мазин М. А.*, *Нарвский А. С.*, *Шалыто А. А.* UML. SWITCH-технология. Eclipse. //«Информационно-управляющие системы», 2004, № 6, с.12–17. http://is.ifmo.ru/works/uml-switch-eclipse/
9. *Гуров В. С.*, *Мазин М. А.*, *Шалыто А. А.* UniMod — Инструментальное средство для автоматного программирования. //Начуно-технический вестник. Вып. 30. Фундаментальные и прикладные исследования информационных систем и технологий. СПбГУ ИТМО. 2006, с. 32–44. http://is.ifmo.ru/works/_instrsr.pdf
10. *Кормен Т.*, *Лейзерсон Ч.*, *Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 1999.

11. *Веб-сайт* проекта *Eclipse Modeling Framework.* http://www.eclipse.org/modeling/emf/
12. *Rumbaugh J., Jacobson I., Booch G.* The Unified Modelling Language Reference Manual, Second Edition. Addison-Wesley, 2004.
13. *Веб-сайт* проекта *Eclipse OCL.* http://www.eclipse.org/modeling/mdt/?project=ocl#ocl
14. *Веб-сайт* проекта *Kent OCL.* http://www.cs.kent.ac.uk/projects/ocl/
15. *Веб-сайт* проекта *Dresden OCL Toolkit.* http://dresden-ocl.sourceforge.net/
16. *Wahler M.* Using OCL to interrogate EMF model, 2004.
17. *Warmer J.*, *Kleppe A.*. The Object Constraint Language: Getting Your Models Ready for MDA, Second Edition. Addison-Wesley, 2003.
18. *Веб-сайт Eclipse SDK*. http://help.eclipse.org/