Санкт-Петербургский государственный университет
информационных технологий, механики и оптики
Кафедра «Компьютерные технологии»

А. А. Борисенко

Интеграция верификации в интерактивный процесс разработки программ с явным выделением состояний

Бакалаврская работа

Научный руководитель – О. Г. Степанов

Санкт-Петербург
2010

Оглавление

ВВЕДЕНИЕ

Качественное программное обеспечение (ПО) – это, прежде всего, надежное ПО. Поэтому при разработке любой программы важное место занимает стадия тестирования. Как известно, вместе с увеличением сложности самой программы следует ожидать и увеличения числа ошибок в ней. В настоящее время все больше ПО реализует сложное поведение [1]. При этом во многих проектах цена ошибки может быть слишком высокой [2].

Другой важной чертой современных программных проектов является их частое изменение: модифицируются требования к системе, находятся и исправляются ошибки. Также реализация программных систем периодически подвергается рефакторингу. Для контроля качества ПО, соответствия его реализации и спецификации в современных проектах используется ряд методов: ручное и автоматизированное тестирование, контрактное программирование и верификация [3].

Для разработки программных систем со сложным поведением все чаще используются *программирование с явным выделением состояний*. При этом подходе по словесной спецификации сложного поведения проектируется соответствующий ему набор взаимодействующих автоматов [4,5]. Затем эта система автоматов реализуется с использованием одного из подходов к объектно-ориентированному программированию с явным выделением состояний [6 – 8]. При этом строится программа называемая *автоматной*. Современные методы позволяют автоматизировать изоморфный перенос спроектированной автоматной системы в код, или, по крайней мере, значительно сократить вероятность ошибки при таком переносе.

Таким образом, основным источником ошибок в автоматной программе является проектирование автоматной системы по спецификации. При этом отметим, что для проверки соответствия спроектированной автоматной системы спецификации в настоящее время используются тестирование [9] и верификация на модели [10].

При организации процесса контроля качества автоматной программы возникает ряд проблем. Во-первых, в существующих проектах обычно ограничиваются одним из видов контроля, в то время как различные подходы имеют свои преимуще-

ства и недостатки. Во-вторых, некоторые из подходов к обеспечению качества, принятых при разработке современных программных систем, не приспособлены к использованию в автоматном программировании [1]. В-третьих, отсутствуют инструменты, облегчающие контроль соответствия спецификации при программировании с явным выделением состояний.

В рамках настоящей работы сделана попытка найти решения этих проблем и приблизиться к созданию метода, позволяющего разрабатывать надежные объектно-ориентированные системы с явным выделением состояний. При этом, в частности, получены следующие результаты:

1. Предложен подход к программированию по контрактам (контрактному программированию) с явным выделением состояний. При этом отдельно рассмотрены внешние и внутренние контракты.
2. Разработан метод, позволяющий выбирать оптимальный способ формализации спецификаций к автоматным системам в зависимости от характера спецификации и особенностей автоматной системы.
3. Предложен способ интеграции действий по обеспечению соответствия реализованной автоматной системы спецификации в процесс разработки программного обеспечения.
4. Реализована часть функциональности инструмента, позволяющего проводить верификацию автоматной системы во время разработки.

Работа имеет следующую структуру. В первой главе рассмотрены существующие методы контроля качества современных программных систем и автоматных программ. Во второй главе изложен предлагаемый подход к использованию контрактов в автоматных системах, а также предложен метод формализации спецификаций к автоматным программам. В третьей главе изложена идея создания среды, позволяющей поддержать сразу три подхода к проверке качества программ с явным выделением состояний: проверку на модели, модульное тестирование и контракты. В четвертой главе изложены этапы практической реализации предложенного подхода. В последних двух главах работы рассмотрены вопросы внедрения результатов работы и демонстрационный пример.

1. КОНТРОЛЬ КАЧЕСТВА АВТОМАТНЫХ ПРОГРАММ

К автоматным программам могут быть успешно применены следующие методы анализа корректности [6]:

- тестирование;
- верификация: проверка на модели (*Model Checking*) и доказательная верификация;
- контракты.

Рассмотрим последовательно каждый из них, выделяя при этом характерные для данного метода достоинства и недостатки.

1.1. Тестирование

Тестирование – процесс выявления ошибок в программном обеспечении. Запуск программы на определенных входных данных, а также проверка различных сценариев выполнения, позволяют достаточно быстро (по сравнению с другими методами поиска ошибок) убедиться в корректности обработки заданных сценариев.

Тестирование, применяемое после окончательного написания программы, не способно найти все ошибки. Как заметил Э. Дейкстра, если при тестировании ошибки в программе не найдены, это еще не означает, что их там нет. Различают два основных вида тестирования [11]:

- тестирование белого ящика (*white-box testing*);
- тестирование черного ящика (*unit testing*).

В первом случае разработчик теста имеет доступ к исходному коду программ и может писать код, который связан с библиотеками тестируемого ПО.

Во втором случае специалист по тестированию имеет доступ к ПО только через те же интерфейсы, что и заказчик или пользователь, либо через внешние интерфейсы, позволяющие другому компьютеру либо другому процессу подключиться к системе для тестирования. Как правило, тестирование черного ящика производится с использованием спецификаций или иных документов, описывающих требования к системе. Как правило, в данном виде тестирования критерий покрытия складывается

из покрытия структуры входных данных, покрытия требований и покрытия модели (в тестировании на основе моделей).

Последнее время большую популярность получило модульное тестирование (*unit-testing*) – процесс, позволяющий проверить на корректность отдельных модулей исходного кода программы. Оно позволяет проверить, что компоненты конструкции до определенной степени работоспособны и устойчивы [12].

Важно отметить, что исправление одной ошибки может привести к появлению другой. В рамках данного подхода тесты создаются для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к *регрессии* – к появлению ошибок в уже написанных и оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

1.2. Верификация

Артефактом жизненного цикла ПО называются различные информационные сущности, документы и модели, создаваемые или используемые в ходе разработки и сопровождения ПО [13].

Верификация проверяет соответствие одних создаваемых в ходе разработки и сопровождения ПО артефактов другим, ранее созданным или используемым в качестве исходных данных, а также соответствие этих артефактов и процессов их разработки правилам и стандартам.

В частности, верификация проверяет соответствие между нормами стандартов, описанием требований (техническим заданием) к ПО, проектными решениями, исходным кодом, пользовательской документацией и функционированием самого ПО.

1.2.1. Формальная верификация

Формальная верификация представляет собой процесс доказательства с помощью формальных методов корректности или некорректности алгоритмов, программ и систем в соответствии с заданным описанием их свойств. Она требует вы-

сококвалифицированных специалистов в области формальных доказательств и логического вывода.

В общем случае, задача, решаемая в рамках этого подхода, является алгоритмически неразрешимой. При этом весь процесс формального доказательства связан с огромной ручной работой, что делает его малоприменимым на практике [3].

1.2.2. Верификация на модели

Под верификацией на модели понимают метод формальной верификации, позволяющий проверить, удовлетворяет ли заданная модель системы формальным спецификациям. Применение данного подхода позволяет для заданной модели поведения системы с конечным (возможно, очень большим) числом состояний проверить выполнимость некоторого требования (спецификации), которое обычно формулируется в терминах языка темпоральной логики (*LTL*, *CTL* и т.д.). Таким образом, можно проверить не только условия в определенном состоянии системы, но и историю развития во времени.

Метод верификации на модели хорошо подходит для проверки поведения автоматных программ. Это связано с тем, что при верификации модели, обычно, по программе строится модель Крипке [10], которая, фактически, является специальным видом автомата. Это упрощает ее построение по автоматной программе [5]. Поэтому возможно создание формальных методов построения модели Крипке по автоматной программе и доказательство корректности этих методов. Таким образом, для программ этого класса можно исключить ошибки при построении модели по программе.

Отметим также простоту описания формальных требований к модели по спецификации автоматной программы. Например, такие требования, как «После состояния «Открыто» всегда следует состояние «Закрыто»» или «Выходное воздействие `z1` не должно появляться после входного воздействия `x1`» могут быть записаны в терминах темпоральной логики для состояний и переходов непосредственно автоматов, а не модели программы [6].

Так как для автоматных программ модель эквивалентна верифицируемой программе, обработка контрпримеров также упрощается. При этом путь, приводящий к ошибке в модели Крипке, может быть легко преобразован в путь автоматной программы [3]. Это позволяет наглядно продемонстрировать нарушение спецификации.

Таким образом, верификация автоматных программ является существенно более простой задачей, чем верификация программ общего вида. При этом основные этапы верификации могут быть автоматизированы, что позволяет более широко применять ее в реальных проектах. В данной работе предлагается подход, при котором верификация интегрирована в сам процесс разработки программы с целью повышения его эффективности.

1.3. Контракты

Под контрактами обычно понимают совокупность способов формализации и проверки требований к программе, в которую входят *инварианты*, *постусловия* и *предусловия* [9].

С помощью *инвариантов* удобно отслеживать выполнение требований к работе программы, которые должны исполняться на непрерывном отрезке ее жизненного цикла. Например, в программе, реализующей поведение лифта, можно потребовать, чтобы во время движения кабины двери лифта оставались закрытыми. Это эквивалентно введению инварианта «Двери закрыты» на последовательность состояний программы, в которых лифт находится в движении. Подобные условия помогают лучше понять логику поведения самой программы, а также разобраться в тех ее аспектах, изменение которых влечет за собой нежелательные последствия.

Как правило, инварианты можно выделить даже в самом простом фрагменте кода. При работе над большими проектами и системами со сложным поведением программист часто «держит в голове» некоторые инварианты, которые содержит написанный им код. Например, может считаться недопустимым превышение предела числа хранимых объектов или появление цикла в графе.

Все эти инварианты остаются недоступными для людей, не участвовавших в разработке кода, а тем более для тех, кто будет заниматься поиском ошибок в программе [14].

В качестве *предусловий* используются те ограничения, которые требуют проверки при входе процесса исполнения в некоторый метод или модуль. Например, при работе с объектами в языке *Java* часто требуется сначала проверить, что поданные на вход методу ссылки не равны `null`. Аналогичная ситуация возникает при выходе из метода, когда требуется проверить целостность данных или принадлежность полученного значения допустимому интервалу. Такие проверки осуществляются при помощи постусловий.

Выводы по главе 1

1. Рассмотрены методы контроля качества автоматных программ: тестирование, верификация и контракты. Каждый из них обладает характерными преимуществами и недостатками, которые более подробно будут рассмотрены в разд. 0.
2. Для того чтобы повысить надежность автоматных программ, интегрируем описанные подходы непосредственно в процесс разработки, предварительно изучив уже существующие решения, а также недостатки, которыми они обладают.

2.ФОРМАЛИЗАЦИЯ ТРЕБОВАНИЙ К АВТОМАТНОЙ ПРОГРАММЕ

Требования, предъявляемые к разрабатываемым программам, обычно формируются словесно. Для автоматической проверки их необходимо формализовать.

Перечислим факторы, затрудняющие автоматическую проверку программ. Во-первых, для того чтобы в процессе разработки программы можно было постоянно контролировать ее качество необходимо сделать саму спецификацию исполняемой. При этом требования, сформулированные словесно, невозможно использовать для автоматической верификации. Во-вторых, не существует универсального способа формализации требований. В настоящее время для формализации требований к автоматным программам используют темпоральные спецификации и модульное тестирование. В данной работе предлагается применять для этой цели также и контракты. Будем различать *внешние* и *внутренние* контракты:

- внешние – для интерфейсов автоматизированных объектов управления [1] (относятся к группе состояний, их структура неизвестна);
- внутренние – для состояний автоматной модели (относятся к отдельным состояниям, их структура известна).

Внешние контракты (по аналогии с тестированием «черного ящика) применимы в тех случаях, когда структура автоматной программы неизвестна. К этой группе относятся, в частности, контракты на интерфейсы автоматизированного объекта управления – совокупности управляющего автомата и объекта управления [6]. Кроме того, внешние контракты позволяют описывать требования, предъявляемые к некоторой совокупности состояний, не производя при этом жесткой фиксации ее внутренней структуры.

Внутренние контракты (как и в случае с тестированием «белого ящика») применимы лишь к тем системам, чья структура известна. Например, внутренние контракты могут фиксировать инварианты, постусловия и предусловия, свойственные отдельному состоянию автоматной программы. Особенностью данного типа контрактов является возможность явно выражать требования не к отдельному мето-

ду (как это обычно делается в объектно-ориентированном программировании), а к состоянию автоматной программы.

В рамках объектно-ориентированного программирования применяются внешние контракты, способные производить проверки, использующие лишь интерфейс некоторого класса. Такие же ограничения имеют место и для программирования с явным выделением состояний.

Основываясь на введенном в работе [1] понятии «автоматизированного объекта управления», рассмотрим применимость контрактов для программирования с явным выделением состояний. На основании того, что класс (в объектно-ориентированном подходе) является естественной реализацией для автоматизированного объекта управления, предлагается записывать контракты не на отдельные классы [15], а на состояния.

Обратимся к описанному выше разделению контрактов на внутренние и внешние. Оно отражает применимость предложенного подхода по записи предусловий, инвариантов и постусловий для заданного состояния автоматной модели. Действительно, опираясь на имеющийся интерфейс состояния, можно записать требования, не учитывающие его структуру. Другими словами, как и в случае с тестированием «черного ящика» (в объектно-ориентированном программировании), можно проверять различные свойства отдельной сущности, не имея данных о ее реализации.

Достоинством записи контрактов для некоторого состояния автоматной программы является также возможность использования сведений о его внутренней структуре. Действительно, при необходимости можно объединять сразу несколько состояний в отдельные группы и уже для них записывать предусловия, постусловия или же инвариант. Приведем пример, который наглядно иллюстрирует применимость данного подхода.

Рассмотрим систему (рис. 1), в которой централизованный диспетчер управляет запуском подзадач.

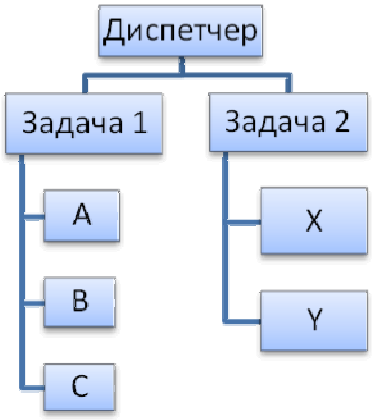


Рис. 1. Пример системы, для которой запись спецификации получается громоздкой

Предположим, что выполнение каждой новой задачи требует некоторых подготовительных действий, осуществляемых диспетчером. Другими словами, в качестве требования к такой системе можно выдвинуть условие на запуск каждой из подзадач только после того, как закончит работу диспетчер.

Попытка записать темпоральную спецификацию для данной системы приводит к громоздкой формуле, длина которой растет экспоненциально с увеличением числа подзадач. Если же использовать внешние контракты, то можно сформулировать множество однотипных условий для каждой из подзадач (`A`, `B`, `C`, `X`, `Y`).

В этом случае удобнее использовать внутренние контракты. При этом подзадачи `A`, `B` и `C` объединим в первую группу («Задача 1»), а подзадачи `X` и `Y` – во вторую. Таким образом, вместо множества формул вида $!(A \rightarrow \mathbf{X}Y), !(B \rightarrow \mathbf{X}Y)$ и т.д. достаточно потребовать для любой задачи с порядковым номером i (в данном случае, $i = 1, 2$), чтобы выполнялось предусловие:

$$\underline{\mathbf{Pred}}[\textit{Задача}_i] = \textit{Диспетчер}.$$

ФОРМАЛИЗАЦИЯ ТРЕБОВАНИЙ, СФОРМУЛИРОВАННЫХ НА ЕСТЕСТВЕННОМ ЯЗЫКЕ

В качестве примера, наглядно иллюстрирующего проблему формализации вербальных требований в форме, пригодной для автоматической проверки, рассмотрим упрощенную модель холодильника. Зададим список требований, предъявляемых к разрабатываемой программе:

1. При закрытой дверце внутренняя лампа не должна гореть.
2. При выходе показаний датчика напряжения за пределы допустимого диапазона управляющее реле выключит питание.
3. В момент отключения охлаждающего элемента показания термостатов не должны превышать допустимые.
4. Пока дверца не будет открыта, лампа не будет включена.
5. Если открыть дверцу холодильника, прогреть его основной объем и закрыть дверь, то:
    - будет включена лампа;
    - при достижении критической температуры будет включен охлаждающий элемент;
    - лампа будет выключена.

Приведенный список требований можно условно разделить на три группы. В первую группу отнесем требования 1–3. Они подходят для записи контрактов. Первое из них («*При закрытой дверце внутренняя лампочка не должна гореть*») является инвариантом – условием, которое должно соблюдаться в течение всего процесса выполнения программы. В требовании 2 проверяется выполнимость некоторого условия при выходе из заданного состояния, а в требовании 3 – при входе в заданное состояние. Формализуем их при помощи пост- и предусловий, оперирующих показаниями термодатчиков.

Ко второй группе можно отнести требование 4: «*Пока дверца не будет открыта, лампочка не будет включена*». Оно содержит выражение, зависящее от по-

следовательности выполнения программы во времени, и его можно компактно записать в качестве темпоральной спецификации:

(*Лампа_не_включена* **U** *Дверь_открыта*)

Ее можно применить для верификации на модели. Занесем полученные результаты в таблице, используя в формальной записи инварианты (`Inv`), постусловия (`Post`) и предусловия (`Pred`).

Таблица 1. Примеры записи формальных спецификаций

1. При закрытой дверце внутренняя лампа не должна гореть	**Inv** [DoorClosed]: lamp.*isTurnedOff*
2. При выходе показаний датчика напряжения из допустимого диапазона управляющее реле выключит питание	**Post** [VoltageOutOfRange]: powerAdapter.*isTurnedOff*
3. В момент отключения охлаждающего элемента показания термостатов не должны превышать допустимые значения	**Pred** [FreezerTurnedOff] : thermoSensor.*valuesInRange*
4. Пока дверца не будет открыта, лампа не будет включена	lamp.*isTurnedOff* **U** DoorOpened

Заметим, что не всякую темпоральную спецификацию удается легко верифицировать. Предположим, что проверке подлежит требование «*лампа в холодильнике зажигается строго после открытия двери*». В таком случае, придется исключить в темпоральной спецификации все возможные переходы, допустимые заданной автоматной моделью при открытии двери за исключением собственно зажигания лампы. Полученная формула стала бы громоздкой и перестала бы быть удобочитаемой.

Более того, полученная таким образом спецификация полностью становилась бы привязанной к конкретной реализации модели, а это обстоятельство затруднило бы процесс контроля качества программы при изменении ее реализации.

Для описанного случая хорошо подходит тестирование. Именно к нему целесообразно прибегнуть и при проверке последнего, пятого требования. Оно представляет собой сценарий исполнения. Его легко записать и проверить, используя, например, модульное тестирование.

Отвечающая требованиям из табл. 1 модель холодильника была реализована как с явным выделением состояний, так и без него. При этом была выявлена тенденция, наблюдаемая во многих системах, реализующих сложное поведение – трудность верификации неавтоматной программы.

Попытки отказаться от автоматного подхода в больших проектах, реализующих сложное поведение, могут дать выигрыш по времени на начальном этапе, так как при этом отпадает необходимость строить автоматную модель программы, проектировать ее состояния, переходы и т. д. Сложности возникают на этапе обеспечения качества программы, построенной традиционным путем [8].

Верифицировать можно и объектно-ориентированную программу без явного выделения состояний. При ее верификации на модели должна быть построена структура Крипке, описывающая возможные переходы между состояниями программы, которая затем передается на вход верификатору [6]. При этом возникают сложности, связанные с построением модели Крипке по заданной неавтоматной программе, и возрастает вероятность появления ошибок.

Неудобство формулировки требований к неавтоматной программе затягивает весь процесс контроля ее качества. При этом затраты на построение автоматной модели может с лихвой окупиться удобством ее верификации. Описанный выше пример с моделью холодильника – подтверждение тому. Сложности с записью темпоральной спецификации для неавтоматных программ осложняет их верификацию на модели. Поэтому для таких программ на практике наиболее распространенным средством контроля их качества остается тестирование [12].

Выводы о применимости

Как было показано выше, при проверке некоторых условий удобно применять верификацию на модели. Однако проверять таким образом требование «*событие **A***

произошло сразу же после события ***B***» весьма сложно, так как темпоральные спецификации при этом получаются громоздкими. В некоторых случаях удается сократить, а порой и существенно упростить спецификации, представив ее в качестве контракта. В большей степени это касается «внешних» инвариантов, объединяющих в себе требования к целой группе состояний. Отметим, что обратное преобразование (из контрактов в темпоральную спецификацию) можно задать тремя правилами:

- Инвариант «всегда верно *P*» записывается как ***G*** *P*.
- Предусловие «перед выполнением *P* должно соблюдаться *Q*» записывается как *!(!Q* → ***X****P)*.
- Постусловие «после выполнения *P* должно быть соблюдено *Q*» записывается как *P* → ***X****Q*.

При этом подобный переход к темпоральным спецификациям не всегда бывает оправдан. На практике часто встречаются ситуации, при которых и контракт, и темпоральная спецификация не поддаются упрощению. В таких случаях предлагается применять тестирование, описывая сценарий исполнения программы. При этом тестирование, правда, не гарантирует отсутствия ошибок в программе, а позволяет лишь проверить определенные сценарии исполнения.

Все неявные предположения, сделанные в процессе написания программы, в настоящей работе предлагается выражать с помощью контрактов: записывать инварианты класса, проверять предусловия при входе в тело метода и постусловия при выходе из него. Подобная практика приводит к повышению качества кода, делая его более понятным и надежным. Важно отметить, что ее применению мешает отсутствие единого средства управления процессом контроля качества ПО [16].

Контроль качества в процессе разработки автоматных программ

Процесс верификации на модели обычно состоит из следующих шагов.

1. Построение модели программы.
2. Задание требований в терминах выбранного типа темпоральной логики.

3. Верификация модели с целью проверки выполнения формализованных требований.
4. Анализ контрпримера в случае несоответствия программы требованиям.

Для того чтобы проверять свойства автоматных моделей с помощью уже существующего верификатора, необходимо последовательно реализовать следующие операции:

- транслировать автоматную модель во входной язык верификатора;
- в случае если спецификация неверна, транслировать полученный контрпример обратно в автоматную модель.

Существует несколько работ [3, 4], в которых упомянутые операции выполнялись вручную. При этом шаги 1–4 выполняются, как правило, в различных программах, а при переходе от одного шага к другому, соответствующие преобразования делаются вручную, что может приводить к ошибкам.

Такие ошибки опасны, как минимум, по двум причинам. Первая из них состоит в том, что транслированная с ошибками модель программы может лишиться некоторых своих свойств, которые не удовлетворяют требованиям спецификации. В этом случае, даже если оставшиеся операции будут произведены правильно, результаты верификации будут неверны. В частности, возможно получение сообщения о соответствии модели указанным требованиям, в то время как исходная программа данным требованиям не удовлетворяет.

Вторая причина состоит в том, что при верно заданной модели программы вероятно появление ошибки при записи спецификации. В некоторых случаях, об ошибке сообщит верификатор, но если спецификация записана синтаксически правильно, то процесс верификации может запуститься. В этом случае будет осуществлена проверка не той спецификации, которая требовалась, соответственно, и результаты всей верификации будут неверны.

Инструментальное средство *UniMod* [1] позволяет моделировать и реализовывать объектно-ориентированные программы со сложным поведением на основе автоматного подхода. Отметим, что модель системы для указанного средства строится

с помощью двух типов *UML*-диаграмм: диаграмм классов (в форме диаграмм связей) и диаграмм состояний.

С использованием инструментального средства *UniMod* выполнены проекты, доступные по адресу http://is.ifmo.ru/unimod-projects/. Они показали эффективность применения автоматного программирования и средства *UniMod* при реализации систем со сложным поведением, но также выявили и ряд недостатков:

- ввод диаграмм состояний с помощью графического редактора трудоемок;
- данное инструментальное средство помогает создавать и модифицировать автоматы, но позволяет производить проверку корректности только синтаксиса;
- отсутствуют современные средства эффективной разработки автоматных программ с интеграцией средств верификации.

Выводы по главе 0

1. Для того чтобы повысить качество разрабатываемой автоматной программы, необходимо хранить ее формальную спецификацию непосредственно «рядом» с императивным кодом. Для того чтобы поддерживать эффективную разработку программы и проверку ее качества при внесении изменений в исходный код, спецификация должна быть исполняемой.
2. Большое значение в процессе разработки автоматных программ имеет возможность интеграции не только синтаксической, но и семантической проверки корректности описываемых автоматов.
3. Наряду с описанными проблемами разработки и контроля качества автоматных программ, возникают сложности с формализацией предъявляемых к ним требований.
4. Преобразование требований, предъявляемых к программе к форме пригодной для автоматической проверки, является непростой задачей. Тем не менее, решив ее, можно свести к минимуму «человеческий фактор» при дальнейшем процессе контроля качества программы. Из сказанного выше следует, что вербальное требование

к автоматной программе можно сформулировать как контракт, темпоральную спецификацию или тест. После этого интерактивная среда разработки сообщит результат проверки качества, не требуя вмешательства пользователя.

3. ПРЕДЛАГАЕМОЕ РЕШЕНИЕ

Изложенные в главе 1 методы контроля качества автоматных программ предлагается интегрировать в процесс их разработки, учитывая недостатки существующих подходов, описанные в главе 0. Таким образом, в настоящей работе предлагается объединить достоинства сразу трех подходов для проверки качества программ с явным выделением состояний:

- проверка спецификаций (проверка на модели);
- проверка предусловий и постусловий, инвариантов (контракты);
- unit testing (модульное тестирование).

В результате процесс создания автоматных программ (с внедренными в него указанными подходами) становится эффективным, так как в нем будут сочетаться возможности современной среды разработки (статические проверки, рефакторинг, автодополнение и т.д.) и семантическая проверка автоматного кода.

МУЛЬТИЯЗЫКОВАЯ СРЕДА *MPS*

Для устранения основных недостатков инструментального средства *UniMod*, рассмотренных в разд. 0, было предложено использовать новый подход к разработке автоматных программ [17]. При этом используется мультиязыковая среда *MPS* (*http://www.jetbrains.com/mps*), которая позволяет не только создавать новые языки, так и расширять языки, уже созданные с ее помощью.

Языки, разрабатываемые с помощью *MPS*, не являются текстовыми в традиционном понимании, так как пользователь пишет не текст программы, а вводит программу в виде *абстрактного синтаксического дерева* (АСД). Такой подход позволяет обойтись без создания лексических и синтаксических анализаторов при создании новых языков, а также настроить преобразования АСД в код на конкретном языке программирования и задать удобную среду для его редактирования. Кроме того, пользователь получает возможность после трансляции программы, написанной на языке пользователя, созданном в *MPS*, получить код, не зависящий от этой системы.

Каждый язык, разработанный в среде *MPS,* определяется набором *концептов* (сущностей), определяющих его возможности. Всякий концепт содержит описание детей, ссылок и свойств. *Дети концепта* определяют то, какие узлы могут находиться в поддереве экземпляра этого концепта. Описание ребенка состоит из *роли*, концепта ребенка и *кардинальности связи*. Кардинальность определяет число детей с этой ролью, которое может быть у экземпляра данного концепта.

Ссылки концепта определяют набор ссылок вне собственного поддерева, которые может иметь экземпляр данного концепта. Описание ссылки аналогично описанию ребенка, за исключением того, что кардинальность может принимать только значения 0 и 1. Свойства концепта – это описания строковых, булевских и числовых атрибутов экземпляров этого концепта. Описание свойства состоит из имени и типа его значения. Примером свойства является имя переменной у концепта «объявление переменной».

Выше было отмечено, что редактор абстрактного синтаксического дерева в *MPS* реализован таким образом, чтобы воспроизвести основные особенности процесса редактирования текстовых программ, привычного для программистов. Для этого в основу редактора была положена клеточная структура. Все редактируемое дерево представлено в виде набора вложенных клеток: корневому узлу соответствует клетка, занимающая все поле редактора. Дети узлов представлены клетками внутри клетки родительского узла. Клетки могут быть конечными и составными. Клетки второго типа представляют собой набор вложенных клеток, расположенных на экране с соответствии с выбранной политикой: вертикально, горизонтально или горизонтально с переносом на следующую строку. Атомарные клетки могут быть константными, содержимое которых задано, и редактируемыми, содержимое которых может изменяться пользователем. Редактируемые клетки могут использоваться, например, для отображения и редактирования атрибутов узла. Кроме того, клетки могут реагировать на нажатие определенных клавиш и изменять соответствующим образом дерево.

Описание редактора концепта определяет набор клеток, которыми узлы этого концепта будут представлены в редакторе. Процесс создания редактора реализует

принцип *WYSIWYG* (*What You See Is What You Get*). Это позволяет оценивать результат непосредственно при редактировании.

Помимо редактирования значений свойств, редактор должен обеспечивать удобство создания новых экземпляров концептов. За это отвечает меню подстановки. При нажатии комбинации клавиш *Control+Space* активизируется меню, с помощью которого можно выбрать концепт создаваемого узла. По умолчанию список содержит все доступные конкретные концепты, расширяющие требуемый, но это поведение возможно изменить. Другой важной возможностью является механизм левых и правых трансформаций, позволяющих преобразовывать дерево при нажатии определенных клавиш. Например, при нажатии после числовой константы клавиши «+», она будет преобразована в узел операции сложения.

Генератор языка содержит набор правил, преобразующих синтаксическое дерево программы на одном языке в дерево на другом языке, для которого уже задана семантика. Обычно таким языком является *Java*, для которого используется стандартный компилятор и среда исполнения. Если один язык расширяет другой, то его генератор может преобразовывать программу в дерево на расширяемом языке. После этого стандартный генератор преобразует его в код на языке *Java*.

Совмещение императивного кода, спецификаций и контрактов

Предложенный в настоящей работе способ создания безопасных программ с явным выделением состояний базируется на совмещении в процессе разработки императивного кода, автоматного представления, а также трех вышеописанных подходов к проверке качества ПО: верификации на модели, программирования по контрактам и тестирования.

Достоинством такого подхода является привязка спецификаций и проверяемых контрактов к программе, записанной в терминах конечных автоматов. При этом все управление осуществляется централизовано на базе свободно распространяемой среды метапрограммирования *MPS*. Отдельно отметим наличие разработанного в этой среде языка *stateMachine* [17], позволяющего для *Java*-класса, реализующего

сложное поведение, добавить автоматную модель, указав набор состояний и переходов между ними.

Выводы по главе 3

1. Интеграция сразу трех распространенных подходов проверки качества автоматных программ непосредственно в процесс их разработки позволит повысить надежность создаваемых программ.
2. У разработчиков появится возможность не только эффективно создавать автоматную модель, используя при этом средства *MPS*, но и самостоятельно выбирать, какой способ формализации использовать в зависимости от поставленной перед ними задачи.
3. Запущенный процесс контроля качества автоматически выполнит необходимые преобразования, сообщив разработчику результаты проверки.
4. Перечислим некоторые дополнительные свойства, которые можно получить при создании централизованного средства проверки программ на базе *MPS*:

- автоматическое завершение ввода текста при описании модели;
  программы, спецификации и контрактов;
- удобная форма записи спецификации и контрактов в редакторе с использованием ввода: оператор – операнд – оператор;
- проверка совместимости типов операндов на этапе записи спецификации, возможность расширять правила типизации выражений.

4. РЕАЛИЗАЦИЯ ПРЕДЛОЖЕННОГО ПОДХОДА

Для добавления верификации и проверки контрактов в процесс разработки автоматных программ потребуется:

- создать язык темпоральных спецификаций и контрактов, который был назван *stateSpec*;
- описать операторы языка темпоральной логики *LTL*;
- настроить систему типов темпоральных операторов;
- внедрить секцию со спецификацией в описание автоматных программ на языке *stateMachine*;
- разработать плагин для запуска внешнего верификатора *NuSMV*;
- указать сочетания клавиш, запускающие верификацию или проверку контрактов;
- преобразовать автоматную модель к форме, пригодной для автоматической верификации;
- настроить обработку полученных данных верификации;
- выделить сообщение о результатах верификации;
- в случае обнаружения контрпримера, заменить исходными названия состояний и событий в цепочке исполнения программы, приводящей к нарушению спецификации или контракта.

СОЗДАНИЕ ЯЗЫКА ТЕМПОРАЛЬНЫХ СПЕЦИФИКАЦИЙ И КОНТРАКТОВ

Данный раздел подробно описывает процесс создания языка *stateSpec* в мультиязыковой среде *MPS*. Ниже будет показано, как с его помощью можно записывать и проверять темпоральные спецификации и контракты для автоматных программ.

Как отмечено выше, в среде *MPS* язык представляет собой набор *концептов* (*concepts*). Наиболее важные из них: структура языка, редактор, генератор и система типов (рис. 2).

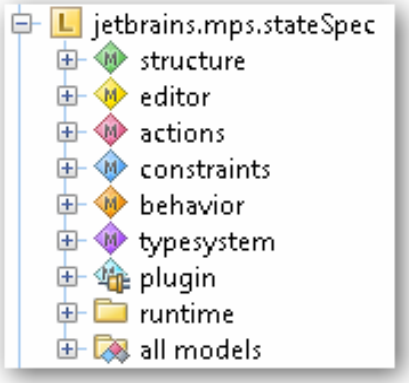


Рис. 2. Составные части языка *stateSpec* в среде *MPS*

Структура языка определяет набор узлов для построения синтаксического дерева (например, операторы и операнды). Редактор для данного концепта определяет то, как будет отображены его экземпляры в коде самой программы. Генератор задает семантику языка (явно описывая операции, необходимые для получения исполняемого кода), а система типов позволяет наложить ограничения на использование экземпляров данного концепта. Концепты могут наследовать друг друга и образовывают иерархию.

Это позволяет выделять общую функциональность в базовые концепты с целью их повторного использования. В частности, далее будет описано создание концептов *BaseTemporalExpression* и *StateContract* для представления темпоральных операторов и контрактов, соответственно.

Описание сущностей для языка *LTL*

Опишем все операторы языка темпоральной логики *LTL*. Как известно, этот язык состоит из трех унарных (**X**, **G**, **F**) и двух бинарных операторов (**U**, **R**):

- *Next (X)* – в следующий момент времени верно, что …;
- *Global (G)* – в любой момент времени верно, что …;
- *Future (F)* – когда-либо будет верным, что …
- *Until (U)* – первое условие верно, пока не будет выполнено второе;
- *Release (R)* – первое условие верно, пока не будет выполнено второе.

Создадим иерархию сущностей для работы с темпоральными операторами языка *LTL*, во главе которой будет абстрактный concept *BaseTemporalExpression*. Его задачей является хранение ссылки на выражение типа *Expression*, которое в дальнейшем можно будет полиморфно анализировать. Также в эту сущность будет добавлено ограничение: применять темпоральную спецификацию разрешено лишь в секции для спецификаций (рис. 3).

```
concepts constraints BaseTemporalExpression {
  default concrete concept: <no defaultConcreteConcept>

  can be child
  (operationContext, scope, parentNode, link, childConcept)->boolean {
    parentNode.ancestor<concept = Specification, +> != null;
  }

  can be parent <none>

  <<property constraints>>

  <<referent constraints>>

  default scope
  <no default scope>

}
```

Рис. 3. Запрет на использование темпоральных операторов вне заданной секции для спецификаций

Затем опишем две сущности – отдельно для унарных и бинарных операторов, настроив для каждого редактор и возможность иметь псевдоним (рис. 4, 5).

```
concept UnaryTemporalExpression extends    BaseTemporalExpression
                                implements <none>
  children:
  BaseTemporalExpression | expression | 1 | specializes: | <none>
  concept properties:
  abstract
```

Рис. 4. Описание сущности, базовой по отношению к унарным темпоральным операторам

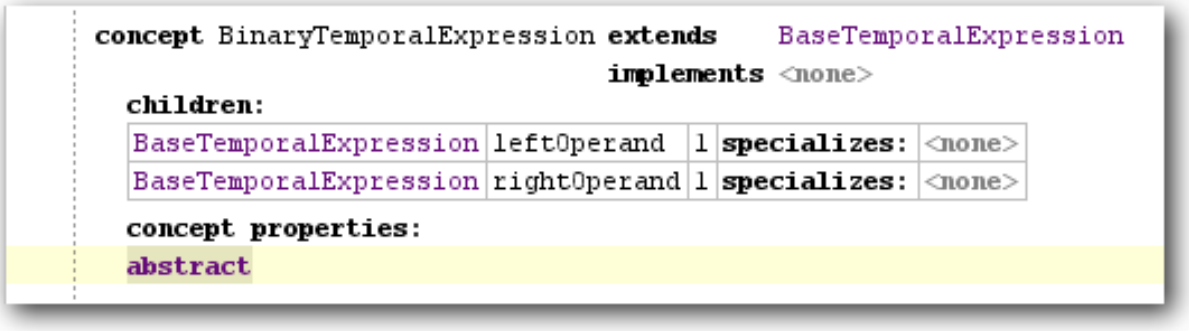


Рис. 5. Описание сущности, базовой по отношению к бинарным темпоральным операторам

Определим также удобный редактор для работы с выражениями на языке темпоральных спецификаций. В случае с бинарными операторами редактор выглядит следующим образом (рис. 6).

Рис. 6. Настройка редактора для бинарных темпоральных операторов

Это позволит в дальнейшем записывать бинарные операторы в спецификациях с использованием ввода следующего вида: оператор – операнд – оператор.

Проделаем аналогичные операции с *UnaryTemporalExpression* и реализуем все необходимые операторы языка *LTL* (рис. 7).

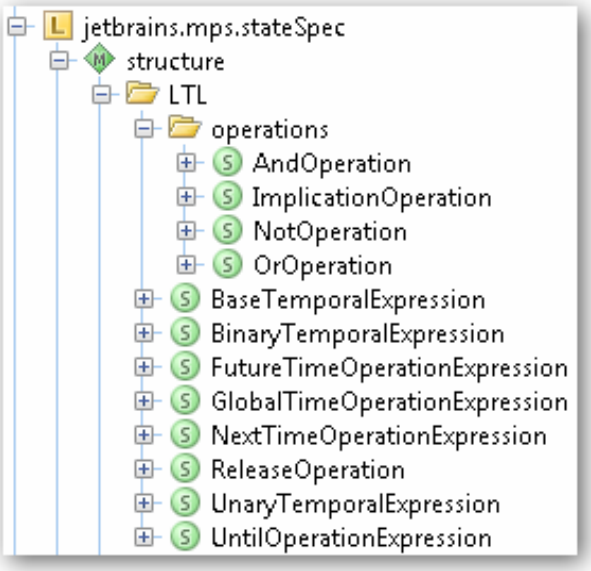


Рис. 7. Структура созданного языка с поддержкой темпоральных операторов и секций для спецификаций

При этом создадим сущности, описывающие логические операторы булевой логики:

- *And (&&)* – И;
- *Or (||)* – ИЛИ;
- *Not (!)* – НЕ;
- *Implication(→)* – Импликация.

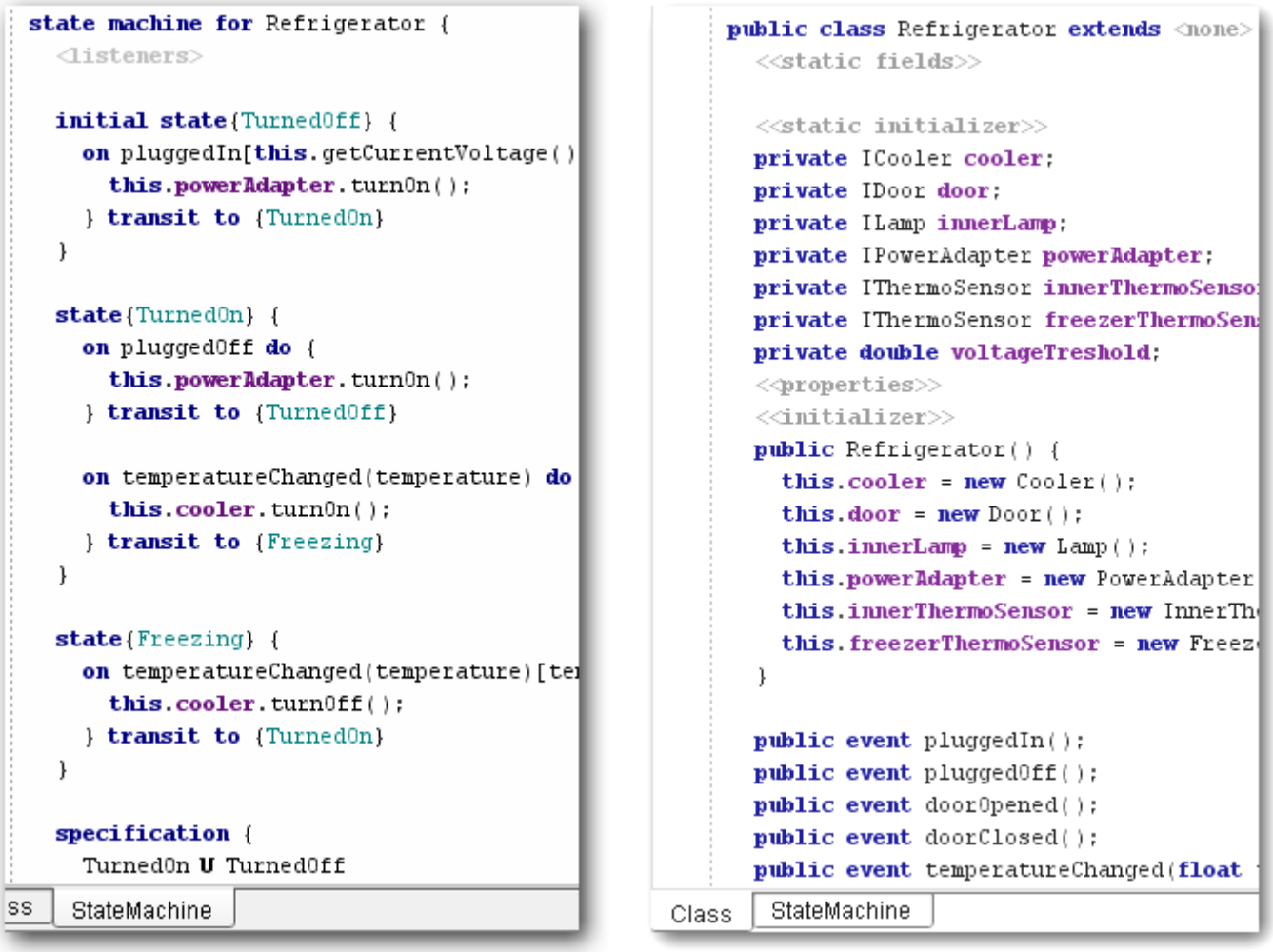
Проблема добавления в созданный нами язык поддержки логических булевых операторов решается настройкой системы типов: в качестве типа возвращаемого значения и типа операнда темпоральным операторам разрешим использовать встроенный тип *Boolean*.

ИНТЕГРАЦИЯ В АВТОМАТНУЮ МОДЕЛЬ

На рис. 7, а также в язык *temporalLogicLanguage* были добавлены сущности *StateReference* и *Specification*. Первая из них представляет собой ссылку на состояние автоматной модели, описанную на языке *StateMachine* [18].

Вторая сущность требуется для того, чтобы интегрировать секцию спецификаций в описание автоматной модели. В качестве примера на рис. 8 приведены составные части модели холодильника из разд.0: императивный код в виде *Java*- класса и

описание конечного автомата. В качестве примера спецификации в секции *specification* приведена формула «пока холодильник не выключат, он будет оставаться включенным».



а б

Рис. 8. Описание автоматной модели холодильника
с помощью конечного автомата (а), императивное (б).

После создания языка для описания темпоральных спецификаций, необходимо решить проблему записи автоматной модели (заданной в среде *MPS*) на языке *SMV*, с которым работает свободно распространяемый верификатор *NuSMV* (http://nusmv.irst.itc.it/).

Поддержка контрактов

Для того чтобы поддержать проверку контрактов в разрабатываемом языке *stateSpec* воспользуемся результатами разд. 0. В нем были предложены формулы для записи инвариантов, предусловий и постусловий через темпоральные спецификации.

На основе результатов, полученных ранее, можно переформулировать заданные к автоматной программе контракты на языке *LTL*, а затем провести их статическую проверку, используя все тот же верификатор *NuSMV*. Таким образом, пользователь сможет записывать и проверять темпоральные спецификации или контракты в тех случаях, когда это будет удобнее.

Следует отметить, что верификатор *NuSMV* работает со своим языком описания автоматных программ. При этом, как уже отмечалось выше, преобразование модели к этому языку вручную чревато появлением привнесенных ошибок. Поэтому преобразование автоматной модели, созданной в среде *MPS,* к форме, понятной верификатору *NuSMV*, необходимо автоматизировать. Остановимся на этом вопросе подробнее.

Описание автоматной модели на языке *SMV*

Модель на языке *SMV* содержит набор перечислимых переменных и правила переходов. Для записи модели необходимо явно задать правила, согласно которым эти переходы могут осуществляться. Наглядно проиллюстрируем синтаксис данного языка на следующем примере:

```
MODULE main
VAR
    request : boolean;
    state : ready, busy;
ASSIGN
    init(state) := ready;
    next(state) := case
        state = ready & request = 1 : busy;
        1 : ready, busy;
```

```
esac;
```

Описанная модель имеет два состояния: `ready` (начальное) и `busy`. Если переменная `request` имеет в данный момент значение `true`, а также `state = ready`, то состояние изменяется на `busy`. Иначе, смена состояний происходит недетерминировано.

В случае, когда автоматная программа написана на языке *stateMachine,* то для ее записи на языке *SMV* достаточно проанализировать условия на переходах и заменить их на булевские переменные, а также добавить информацию о возможных состояниях и правилах переходов. На данном этапе можно сохранить в памяти таблицу соответствия исходных условий на переходах, а также имен булевских переменных, которыми они были заменены. Она понадобится в дальнейшем для преобразования результатов работы верификатора в термины исходной модели.

Автоматизация проверки контрактов и верификации

В разд. 0– 0 был описан процесс разработки языка спецификаций *LTL* и перевод автоматной модели (со всеми ее темпоральными спецификациями и контрактами) в формат *SMV*. Для того чтобы полностью автоматизировать процесс верификации автоматных программ и проверки контрактов, описанных в среде *MPS*, необходимо разработать инструмент, который позволил бы выполнять последовательно все вышеописанные преобразования, запускать верификатор *NuSMV* и выводить полученные результаты (в терминах исходной модели) пользователю.

Для этого создадим в среде *MPS* отдельный *плагин* (дополнительную функциональность, доступную по нажатию заданного сочетания клавиш). В его тело добавим вызовы соответствующих методов, разработанных для преобразования контрактов и темпоральных спецификаций. После выше описанных преобразований исходной модели будет запускаться внешний верификатор *NuSMV* (работа с ним описана в классе *NuSMVProcess* на рис. 9).

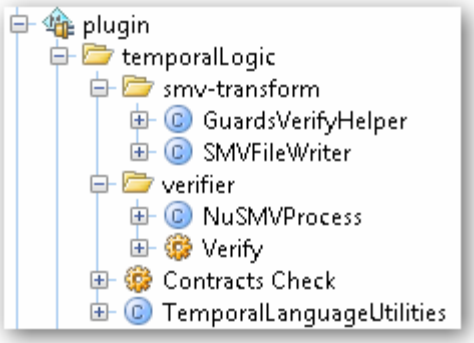


Рис. 9. Плагин для запуска верификатора *NuSMV*

Для удобства зададим сочетание клавиш, которое будет его вызывать (в данном случае, *ALT+V*) и ограничим область его использования секциями с темпоральными спецификациями и контрактами. Как отмечено выше, для концептов, описывающих автоматную модель в среде *MPS* (как и для обычных *Java*-классов) доступны инструменты модульного тестирования. В случаях, когда требования к программе удобнее записать в виде темпоральных спецификаций или контрактов, необходимо создать соответствующую секцию в описании автомата.

Для наглядности приведем в качестве примера запись требований, сформулированных для модели холодильника, рассмотренной в разд. 0. В качестве контрактов запишем инвариант на допустимые значения напряжения при работе холодильника, а также постусловие, требующее, чтобы в состояние готовности он приходил лишь строго после того как будет включен (рис. 10).

```
initial state{ReadyToUse} {

  invariant {
    voltageTreshold == 220
    currentVoltage < voltageTreshold
  }

  require for ReadyToUse
    TurnedOn
  }
}
```

Рис. 10. Контракты, записанные для состояния *ReadyToUse*

В секции для темпоральных спецификаций запишем требование: лампа в холодильнике выключена до тех пор, пока не будет открыта его дверца (рис. 11).

```
specification {
   this.lamp.isTurnedOff() U DoorOpened
}
```

Рис. 11. Темпоральная спецификация

Далее можно запустить проверку контрактов или спецификаций, выбрав соответствующий пункт из контекстного меню (рис. 12).

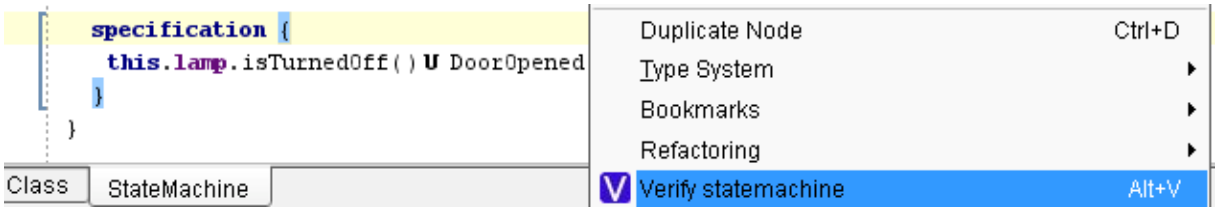


Рис. 12. Запуск процесса верификации

Полученные результаты выводятся пользователю в диалоговом окне (рис. 13)

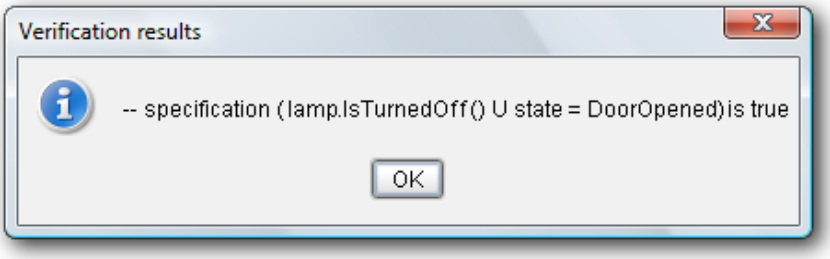


Рис. 13. Сообщение о результатах верификации

В случае если для какой-либо из формул будет найден контрпример, он будет сформулирован в терминах исходной автоматной модели.

Выводы по главе 4

1. В среде *MPS* имеется возможность модульного тестирования для программ, написанных на языке *stateMachine*. Поэтому предложенный подход позволяет разработчику выбирать один из наиболее распространенных на сегодня средств контроля качества программ со сложным поведением: тестирование, верификацию на модели или проверку контрактов (в рамках данной работы, выполняется только статическая проверка контрактов).
2. Полная автоматизация процессов верификации и проверки контрактов исключает «человеческий фактор», позволяя при этом избежать привнесенных ошибок.

3. Статическая проверка, хотя и не позволяет воспользоваться всей мощью контрактов, но покрывает собой многие случаи, когда формализация требований в виде темпоральных спецификаций или тестов затруднена. Для осуществления динамической проверки контрактов требуется тесная интеграция со средой выполнения. Подход с использованием верификатора *NuSMV* в этом случае неприменим. Решение подобной задачи представляет собой одно из возможных направлений дальнейших исследований.

5. ВНЕДРЕНИЕ ПОЛУЧЕННЫХ РЕЗУЛЬТАТОВ

Для практического внедрения описанного в главе 4 инструментального средства была выбрана программа учета дефектов *YouTrack*, разработанная в компании ООО «ИнтеллиДжей Лабс» (работает на мировом рынке под брендом *JetBrains*). Эта программа представляет собой интернет-приложение для работы с базами дефектов. С ее помощью можно добавлять информацию о новых дефектах, а также осуществлять поиск дефектов, уже имеющихся в базе данных. Также имеется возможность редактирования различной сопутствующей информации о каждом из дефектов. Интерфейс программы содержит систему контекстных подсказок и автодополнение вводимых запросов, повышая тем самым удобство работы с программой (рис. 14).

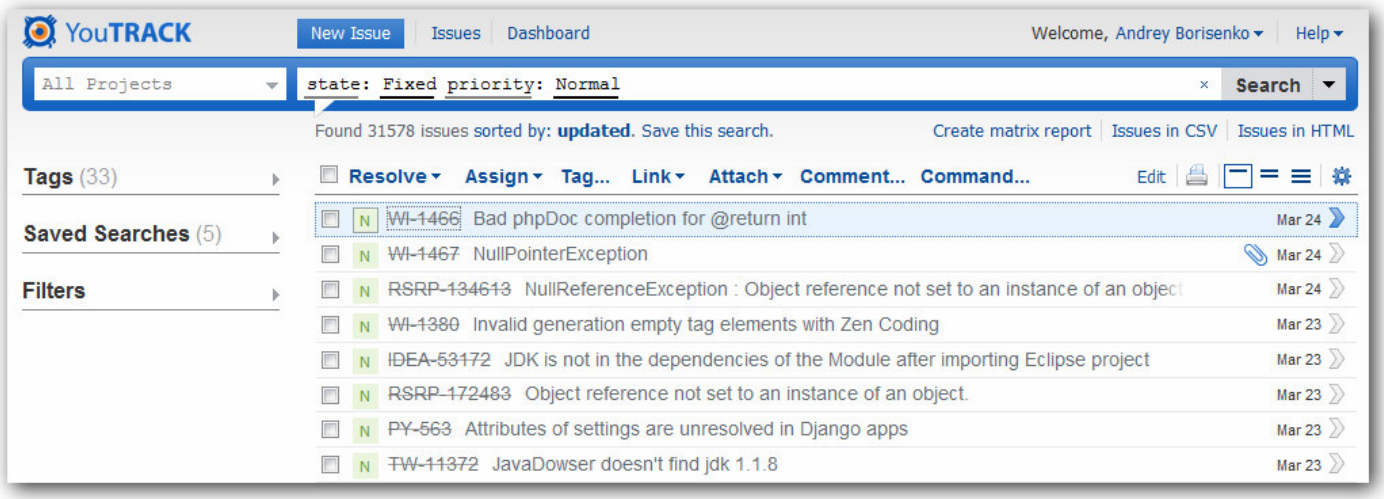


Рис. 14. Пользовательский интерфейс программы *YouTrack*

Область внедрения

Программа *YouTrack* реализована в виде системы следующих взаимодействующих модулей:

- серверный модуль, работающий с базой данных;
- серверный модуль, реализующий логику приложения;
- клиентский модуль, работающий в браузере пользователя и реализующий пользовательский интерфейс.

Часть поведения этой программы реализована в виде автоматов. В частности, автоматы используются в серверной части для синтаксического анализа запросов и в клиентской части для реализации логики пользовательского интерфейса. В качестве области внедрения в рамках данной работы была выбрана именно клиентская часть программы *YouTrack*. Остановимся подробнее на ее устройстве.

Особенности системы учета дефектов *YouTrack*

Особенностью программы *YouTrack* является то, что она реализуется на базе системы *JetBrains MPS*, описанной в разд. 0. При создании этой программы, кроме упомянутого выше языка описания автоматов *stateMachine*, также используются следующие предметно-ориентированные языки системы *MPS*:

- *baseLanguage* – язык, повторяющий синтаксис языка *Java*, применяется для написания императивного кода;
- *dnq* – язык работы с данными, позволяющий выполнять запросы к спискам записей;
- *webr* – язык для разработки веб-приложений;

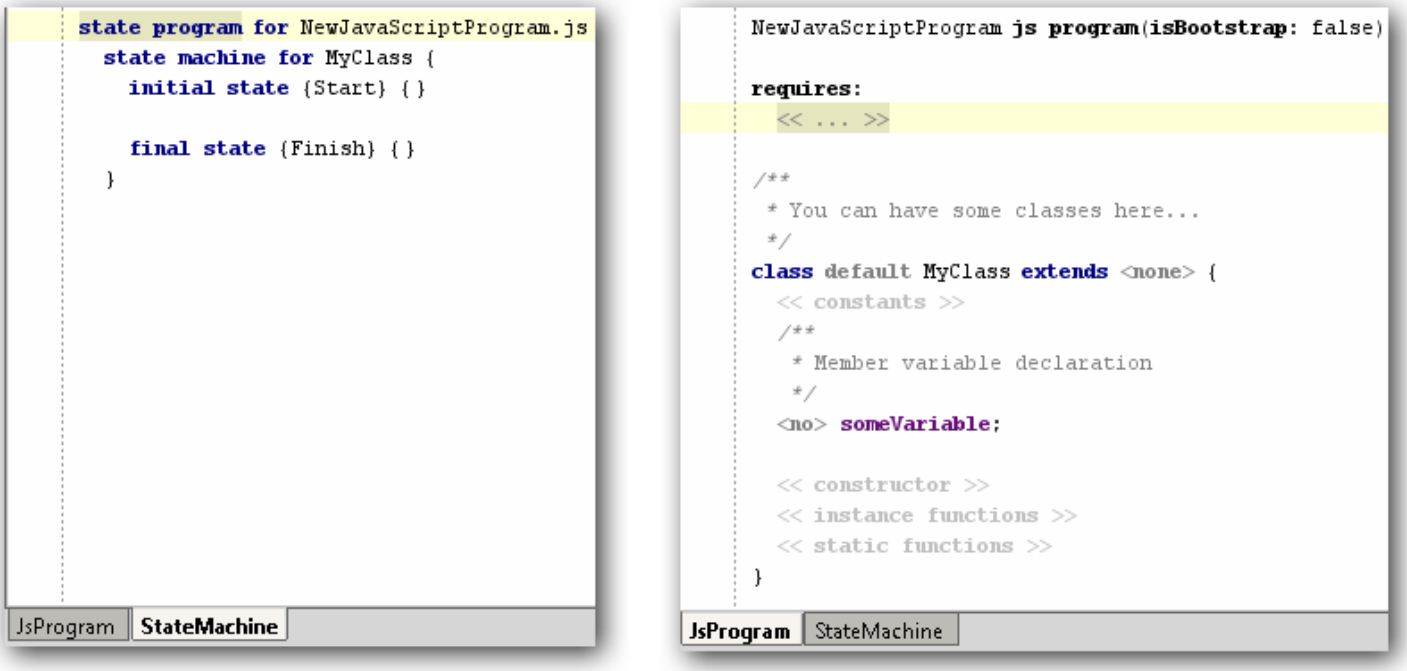
При генерации автоматов, работающих на сервере используется язык *Java*, в то время, как пользовательский интерфейс для работы через браузер написан на языке *JavaScript*. Это обстоятельство привело к тому, что язык *stateSpec* получил отдельную реализацию для работы с программами на *JavaScript*. Далее приводится подробное описание соответствующих модификаций.

Практическая реализация

Последовательно выполнив шаги, описанные в главе 4, получим язык для записи и проверки контрактов и темпоральных спецификаций для автоматных программ. До этого момента рассматривалась автоматная модель, соответствующая *Java*-классу, реализующему объекты со сложным поведением. Другими словами, для заданного класса описывалась автоматная модель, использующая его интерфейс (методы, события). По аналогии с предложенным в разд. 0 принципом совмещения императивного кода и описания автомата, будем хранить рядом с кодом *JavaScript*-

программы ее автоматную модель. Для этого добавим в язык *stateSpec* скриптовую версию языка *stateMachine*. В среде *MPS* для этого достаточно импортировать язык `jetbrains.mps.webr.javascript.stateMachine`, нажав *CTRL+L*, либо явно указать зависимость в свойствах создаваемого языка.

На рис. 15 приведен пример записи кода программы на языке *JavaScript* и соответствующей автоматной модели. Для того чтобы хранить темпоральные спецификации и контракты непосредственно в автоматной модели необходимо повторить действия, аналогичные описанным в разд. 0: создать сущности *StateContract* и *Specification*, но хранить в них ссылку не на *Java*-класс, а на *JavaScript*-программу. При необходимости проверки сложных сценариев исполнения можно воспользоваться модульным тестированием – программа *YouTrack* имеет встроенную поддержку *JUnit* и *Selenium*.



А б

Рис. 15. Описание автоматной модели программы на *JavaScript* с помощью конечного автомата (а), императивное (б)

В качестве примера, иллюстрирующего контроль качества программ, разрабатываемых в рамках системы учета дефектов *YouTrack*, рассмотрим автомат, отвечающий за работу списка дефектов.

АВТОМАТ УПРАВЛЕНИЕМ СПИСКОМ ДЕФЕКТОВ *ISSUELIST.JS*

Список дефектов – один из наиболее важных элементов интерфейса программы *YouTrack,* так как в нем выводится информация об имеющихся в базе дефектах с учетом различных фильтров поиска. Для навигации по списку используются клавиши управления кареткой. Пример работы списка дефектов изображен на рис. 16.

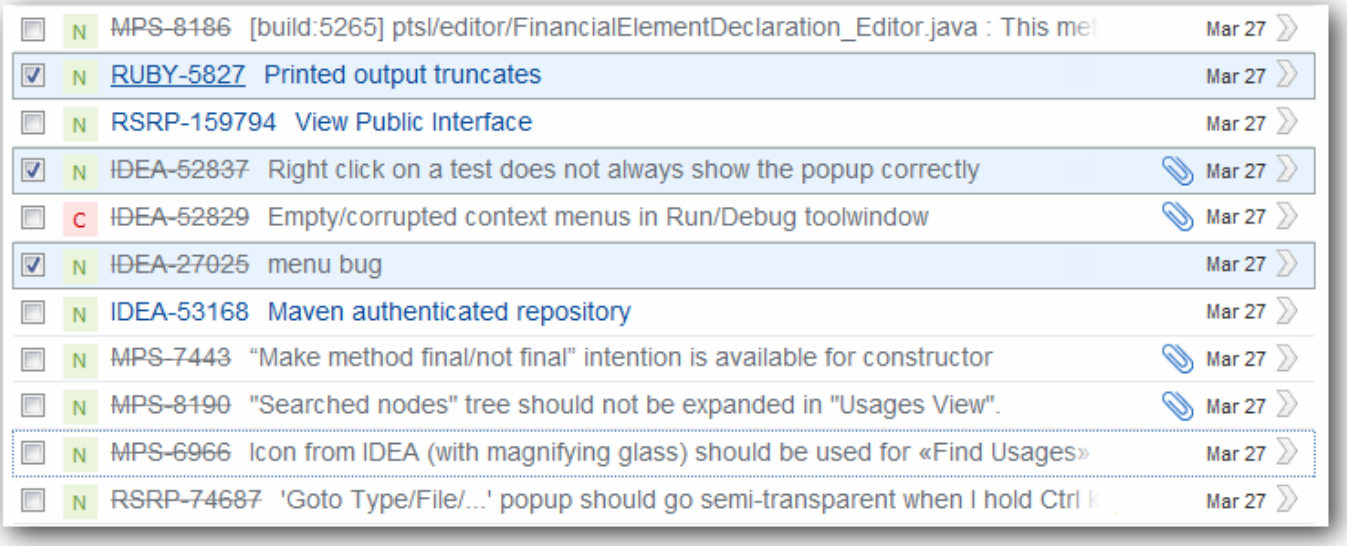


Рис. 16. Пример работы списка дефектов

Выбранные дефекты (выделены синим фоном на рисунке) а также тот дефект, который в данный момент принимает команды пользователя (обведен пунктирной рамкой) являются состоянием списка дефектов. Логика его работы реализована в классе `IssueList.js`. Поведение этого класса задается автоматом, реализованным на языке *stateMachine*, текст которого приведен ниже:

```
state program for IssueList.js
  state machine for charisma.smartui.IssueList {
    initial state {NOT_SHIFTED} {
      exit do {this.ch =
      !(this.getSelected().checkbox.checked);}
      on shiftDown() do {<< statements >>} transit
      to SHIFTED
    }
```

```
state {SHIFTED} {
  initial state {INITIAL} {
    on kdown() do {<< statements >>}
      transit to MOVE_DOWN
    on kup() do {<< statements >>}
      transit to MOVE_UP
    on shiftUp() do {<< statements >>}
      transit to NOT_SHIFTED
  }
state {MOVE_DOWN} {
  enter do
  {this.getSelected().setChecked(this.ch);}
  on kdown() do
  {this.getSelected().setChecked(this.ch);}
  on kup() do {this.ch = !this.ch;}
    transit to MOVE_UP
  on shiftUp() do {<< statements >>}
    transit to NOT_SHIFTED
}
state {MOVE_UP} {
  enter do
  {this.getSelected().setChecked(this.ch);}
  on kup() do
  {this.getSelected().setChecked(this.ch);}
  on kdown() do {this.ch = !this.ch;}
    transit to MOVE_DOWN
  on shiftUp() do {<< statements >>}
    transit to NOT_SHIFTED
}}}
```

Данный автомат обрабатывает четыре события:

- `kdown` (нажата клавиша «Вниз»);
- `kup` (нажата клавиша «Вверх»);
- `shiftDown` (нажата клавиша «Shift»);
- `shiftUp` (отпущена клавиша «Shift»).

Внутренним вычислительным состоянием является значение флага `ch`, указывающее на то, добавляется ли выбранный элемент в выделенный набор или исключается из него.

Выходными воздействиями автомата являются инвертирование флага `ch` («`this.ch = !this.ch`»), включение/исключение выбранного элемента из набора выделенных элементов `this.getSelected().setChecked(this.ch)`, а также инициализация флага `ch`:

```
this.ch = !(this.getSelected().checkbox.checked).
```

Автомат содержит два состояния верхнего уровня: `NOT_SHIFTED` («Клавиша *Shift* не нажата») и `SHIFTED` («Клавиша *Shift* нажата»). Второе состояние включает в себя три вложенных состояния: `INITIAL` («Начальное»), `MOVE_UP` («Перемещение фокуса вверх») и `MOVE_DOWN` («Перемещение фокуса вниз»).

Требования к автоматизированному объекту управления `IssueList.js` формулируются следующим образом:

- в состояние `MOVE_UP` автомат переходит только после нажатия клавиши «Вверх»;
- в состояние `MOVE_DOWN` автомат переходит только после нажатия клавиши «Вниз»;
- при движении курсором в одном направлении не должно производиться инвертирование флага `ch`.

Два первых пункта требований были формализованы в виде **контрактов-предусловий** на состояния `MOVE_UP` и `MOVE_DOWN` с формулами `kup` и `kdown` соответственно (рис. 17).

а б

Рис. 17. Контракты-предусловия на состояния
MOVE_UP (а), MOVE_DOWN (б)

Третий пункт был представлен в виде двух темпоральных формул (рис. 18):

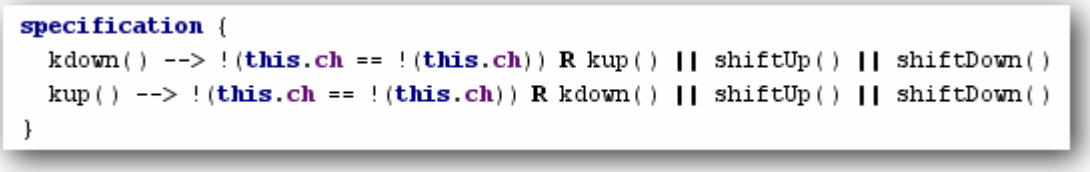


Рис. 18. Темпоральные формулы, описывающие движение курсора в одном направлении

Верификация автомата *IssueList.js* по этим формулам успешно проведена.

Выводы по главе 5

1. Разработана версия языка *stateSpec*, использующая в качестве языка реализации автоматизированных объектов управления *JavaScript*. Возможность такой реализации является свидетельством высокой гибкости предлагаемого метода.
2. В программе учета дефектов *YouTrack* использован язык *stateSpec* для формализации требований к автоматам, реализующим логику поведения компонентов пользовательского интерфейса. Для формализации применялись формулы темпоральной логики и контракты.
3. Встроенная в среду *MPS* поддержка инструментов модульного тестирования (*JUnit*, *Selenium*) позволяет разработчикам проверять сложные сценарии исполнения программ в тех случаях, когда попытка формализации требований с помощью контрактов и темпоральных спецификаций приводит к слишком громоздким формулам.

6. ДЕМОНСТРАЦИОННЫЙ ПРИМЕР

Рассмотрим работу описанного выше инструментального средства проверки контрактов и темпоральных спецификаций на конкретном примере. Пусть задана автоматная модель кофеварки, схематично изображенная на рис. 19.

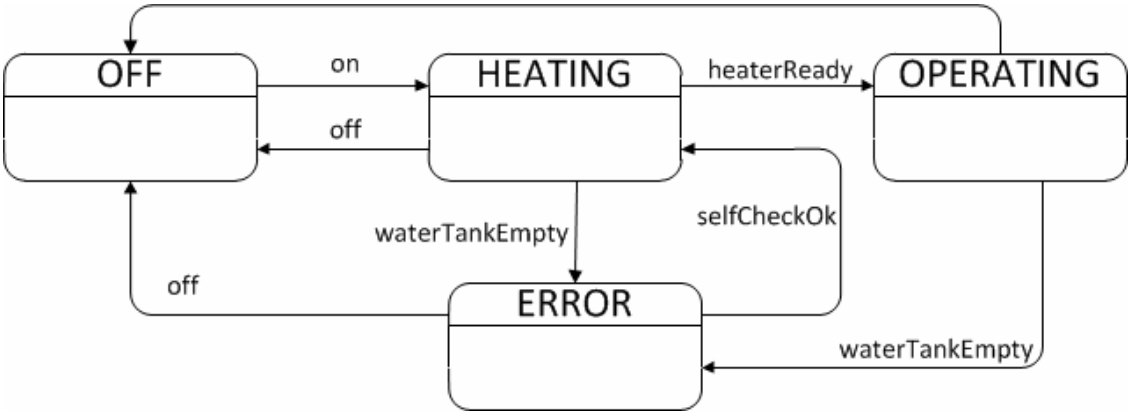


Рис. 19. Автомат, отвечающий за работу кофеварки

Условимся считать стартовым состояние `OFF` (когда кофеварка выключена). Не вдаваясь в подробности поведения данной модели в каждом из выше указанных состояний, потребуем лишь соблюдения очередности переходов, указанной на рис. 19 (соответствующие события подписаны над стрелками).

Для реализации программы потребуется подключить язык *stateMachine* и созданный в рамках данной работы язык *stateSpec*. Контроль качества разрабатываемой программы будем осуществлять на основе некоторого набора требований, предъявляемой к модели кофеварки.

ТРЕБОВАНИЯ К МОДЕЛИ КОФЕВАРКИ

Допустим, что проверке подлежат следующие требования:

1. Кофеварка, находящаяся в состояние нагревания (`HEATING`) или приготовления кофе (`OPERATING`) при опустошении водяного резервуара перейдет в аварийное состояние (`ERROR`).
2. Если на выключенной кофеварке (`OFF`) нажать кнопку `включить`, то кофеварка активирует нагревательный элемент.

3. Резервуар с водой остаётся пустым пока нагревательный элемент не придёт в состояние готовности.
4. Кофеварка может перейти в состояние приготовления кофе (`OPERATING`) лишь из состояния нагрева (HEATING) и только в случае готовности нагревательного элемента.
5. Во время приготовления кофе нагревательный элемент остается включенным.

ФОРМАЛИЗАЦИЯ ТРЕБОВАНИЙ

На основании рекомендаций, описанных в разд. 0, представим требования 4 и 5 как предусловия на состояние приготовления кофе. Соответствующие контракты запишем в `require`-секцию состояния *OPERATING*. Получаем описание автоматной модели, в которое интегрированы последние два требования из вышеуказанного списка (рис. 20).

```
state program for CoffeeMachine.js
  state machine for CoffeeMachine {
    initial state {OFF} {
      on on() do {
        this.turnHeaterOn;
        this.makeCoffee();
      } transit to HEATING
    }

    state {HEATING} {
      on off() do {<< statements >>} transit to OFF

      on heaterReady() do {<< statements >>} transit to OPERATING

      on waterTankEmpty() do {<< statements >>} transit to ERROR
    }

    state {OPERATING} {
      require {
        heaterReady() && HEATING
        this.turnHeaterOn U makeCoffee()
      }

      on off() do {<< statements >>} transit to OFF

      on waterTankEmpty() do {<< statements >>} transit to ERROR

      on makeCoffee() do {this.makeCoffee();}
    }

    state {ERROR} {
      on off() do {<< statements >>} transit to OFF

      on selfCheckOk() do {<< statements >>} transit to HEATING
    }
```

Рис. 20. Автоматная модель кофеварки

Требования 1–3 удобно записать в качестве темпоральных спецификаций. Полученные формулы довольно компактны и просты в понимании. Содержащая их секция представлена на рис. 21.

```
specification {
  G HEATING || OPERATING && waterTankEmpty() --> F ERROR
  waterTankEmpty() U heaterReady()
  OFF && on() --> F this.turnHeaterOn
}
```

Рис. 21. Темпоральные формулы, описывающие требования к кофеварке

Запуск проверки контрактов

Процедура запуска проверки контрактов достаточно проста: в контекстном меню редактора автоматной модели *MPS* пользователю необходимо выбрать команду «Check the contracts». Все необходимые преобразования (разд. 0) произойдут автоматически, а все контракты будут преобразованы в темпоральные спецификации по формулам:

- Инвариант «всегда верно P» записывается как $\boldsymbol{G}\ P$.
- Предусловие «перед выполнением P должно соблюдаться Q» записывается как $!(\ !Q \rightarrow \boldsymbol{X}P)$.
- Постусловие «после выполнения P должно быть соблюдено Q» записывается как $P \rightarrow \boldsymbol{X}Q$.

После этого будет запущен верификатор *NuSMV*, о результатах работы которого пользователю будет сообщено в специальном диалоговом окне (рис. 22).

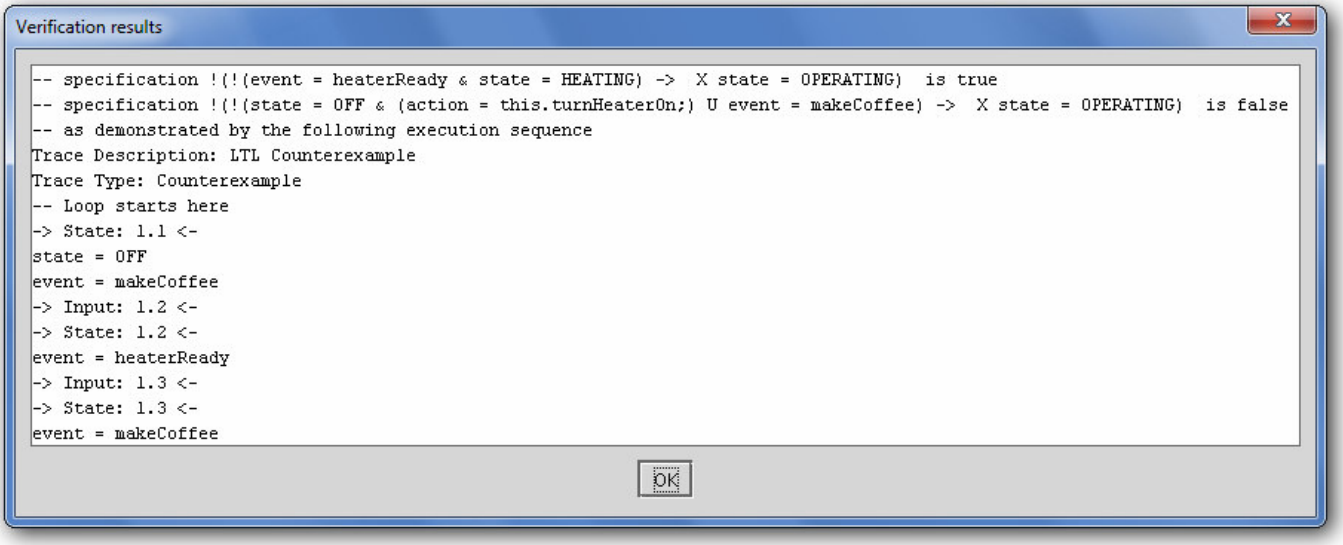


Рис. 22. Результат проверки контрактов

В данном случае, контракт, соответствовавший требованию 4, не выполняется. В состояние *OPERATING*, достижимое из стартового состояния *OFF*, кофеварка может перейти с нарушением предусловия `this.turnHeaterOn` **`U`** `makeCoffee()`. Соответствующий контрпример приводится в диалоговом окне на рис. 22.

Запуск проверки спецификаций

Для того чтобы проверить темпоральные спецификации, сформулированные для данной модели (рис. 21), пользователю необходимо вызвать контекстное меню «Check specification» в окне редактора среды *MPS*. После автоматических преобразований, описанных в главе 4, результаты работы верификатора *NuSMV* будут выведены в диалоговом окне (рис. 23).

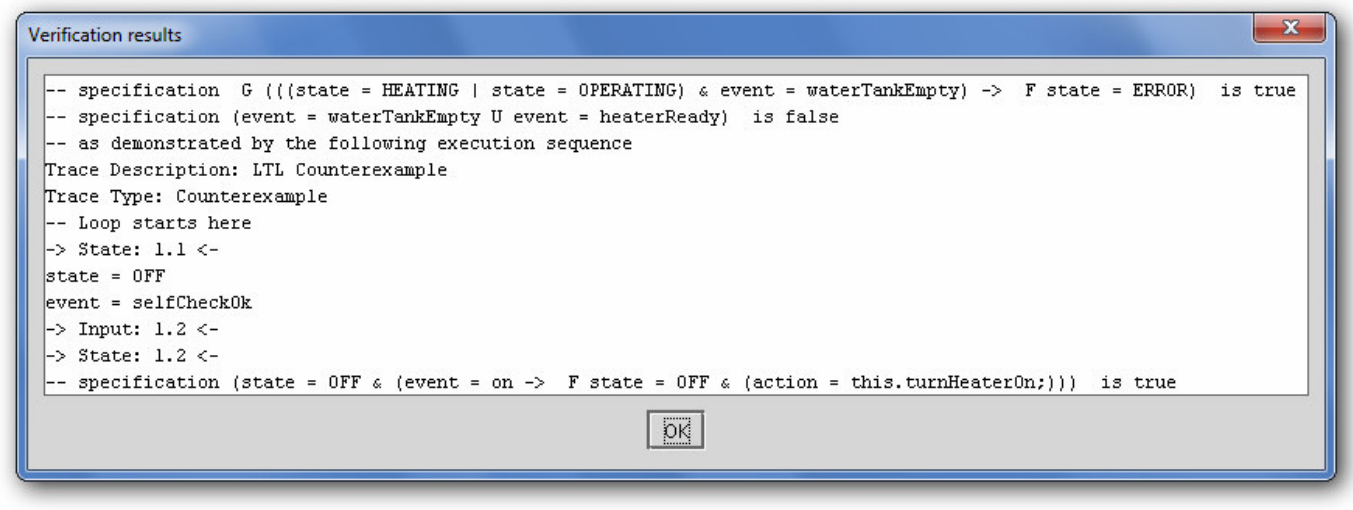


Рис. 23. Результат проверки спецификаций

Согласно результатам верификации, требования 1 и 2 выполняются. Требование 3 («Резервуар с водой остается пустым пока нагревательный элемент не придет в состояние готовности») выполнено не всегда. В качестве контрпримера приводится бесконечный цикл: кофеварка находится в выключенном состоянии (*OFF*), а других событий, кроме `selfCheckOk`, не происходит. Нагревательный элемент никогда не придет в состояние готовности. Следовательно, темпоральная формула `waterTankEmpty` **`U`** `heaterReady` ложна.

Выводы по главе 6

1. Поддержка темпоральных спецификаций и контрактов позволяет разработчикам автоматных программ в среде *MPS* самостоятельно выбирать способ формализации требований, добиваясь при этом упрощения получаемых формул.
2. Отсутствие необходимости ручного преобразования автоматной модели к форме, пригодной для верификации позволяет избежать привнесенных ошибок и повышает качество разрабатываемых программ.
3. Интеграция в среду разработки позволяет исправлять найденные ошибки сразу же в тексте программы.
4. Хранение исполнимой спецификации облегчает контроль качества разрабатываемых программ, позволяет избежать опечаток при составлении требований, предъявляемых к программе.

ЗАКЛЮЧЕНИЕ

В современных проектах вся сопутствующая информация должна обновляться в процессе разработки программы. Это относится и к спецификации. Хранить ее «рядом» с кодом недостаточно. Необходимо сделать ее исполнимой – внедрить контроль автоматический контроль качества программы в процесс ее разработки. Спецификация, которая не отвечает данному требованию, не является эффективной еще и потому, что она никак не реагирует на изменение модели программы и быстро устаревает.

В целях создания среды разработки надежных программ, реализующих системы со сложным поведением, в работе предложено совместить сразу несколько подходов к проверке качества ПО. Тестирование и верификация на модели были внедрены в процесс разработки автоматных программ в среде *MPS*. Также была изучена роль контрактов в преобразовании требований к программе, сформулированных словесно, к форме, пригодной для автоматической проверки.

В качестве направлений дальнейших исследований можно выделить внедрение проверки контрактов во время выполнения программы в предложенной среде контроля качества ПО, изучение ее свойств, а также более глубокое описание контрактного программирования с явным выделением состояний. Отдельной областью дальнейших исследований является итеративная верификация, позволяющая существенно ускорить получение результата, опираясь на данных предыдущих процессов верификации.

СПИСОК ЛИТЕРАТУРЫ

1. *Поликарпова Н. И., Шалыто А. А.* Автоматное программирование. СПб: Питер, 2009. http://is.ifmo.ru/books/_book.pdf
2. *Риган П., Хемилтон С.* NASA: миссия надежна // Открытые системы. 2004. № 3, с. 12–17. http://www.osp.ru/text/302/184060.html
3. Отчет по государственному контракту о верификации автоматных программ. Второй этап. СПбГУ ИТМО, 2007. http://is.ifmo.ru/verification/_2007_02_report-verification.pdf
4. *Вельдер С. Э., Шалыто А. А*. Введение в верификацию автоматных программ на основе метода *Model checking*. 2006. http://is.ifmo.ru/download/modelchecking.pdf
5. *Корнеев Г. А., Парфенов В. Г., Шалыто А. А.* Верификация автоматных программ /Тезисы докладов Международной научной конференции, посвященной памяти профессора А. М. Богомолова «Компьютерные науки и технологии». Саратов: СГУ. 2007, с. 66–69. http://is.ifmo.ru/verification/_KNIT-2007.pdf
6. *Поликарпова Н. И.* Объектно-ориентированный подход к моделированию и спецификации сущностей со сложным поведением. Магистерская диссертация. СПбГУ ИТМО, 2006. http://is.ifmo.ru/papers/oosuch
7. *Курбацкий Е. А., Шалыто А. А.* Верификация программ, построенных при помощи автоматного подхода /Материалы международной научно-методической конференции «Высокие интеллектуальные технологии и инновации в образовании и науке». СПбГПУ. 2008, с. 293–296. http://is.ifmo.ru/download/2008-02-25_politech_verification_kurb.pdf
8. *Кузьмин Е. В., Соколов В. А.* Моделирование, спецификация и верификация «автоматных» программ // Программирование. 2008. № 1, с. 38–60. http://is.ifmo.ru/download/2008-03-12_verification.pdf
9. *Мейер Б.* Объектно-ориентированное конструирование программных систем. М.: Интернет-университет информационных технологий, 2005.

10. *Кларк Э. М., Грамберг О., Пелед Д.* Верификация моделей программ. Model Checking. М.: МЦНМО, 2002.
11. *Кулямин В.В.* Методы верификации программного обеспечения. Институт системного программирования РАН, 2008. http://www.ict.edu.ru/ft/005645/62322e1-st09.pdf
12. *Веденеев В. В.* Автоматизация тестирования использования программных интерфейсов приложений на основе моделирования конечными автоматами. Магистерская диссертация. СПбГУ ИТМО, 2007. http://is.ifmo.ru/testing/vedeneev/
13. *Винниченко И. В.* Автоматизация процессов тестирования. СПб: Питер, 2005.
14. *Мейер Б.* Семь принципов тестирования программ // Открытые системы. 2008. № 7, с. 13–29. http://www.osp.ru/os/2008/07/5478839/
15. *Polikarpova N., Ciupa I., Meyer B. A* comparative study of programmer-written and automatically inferred contracts / Proceedings of ISSTA 2009: International Symposium on Software Testing and Analysis Chair of Software Engineering. Chicago, IL, USA. 2009.
16. *Бромберг И.* Автоматизация тестирования // Открытые системы. 2002. № 5, с.45–47. http://www.osp.ru/os/2002/05/181457
17. *Гуров В. С., Мазин М. А., Шалыто А. А.* Текстовый язык автоматного программирования / Тезисы докладов Международной научной конференции, посвященной памяти профессора А. М. Богомолова «Компьютерные науки и технологии». Саратов: СГУ. 2007, с. 66–69. http://is.ifmo.ru/works/_2007_10_05_mps_textual_language.pdf
18. *Гуров В. С., Шалыто А. А., Яминов Б. Р.* Технология верификации автоматных программ без их трансформации во входной язык верификатора / Международная научно-техническая мультиконференция «Проблемы информационно-компьютерных технологий и мехатроники». Материалы международной научно-технической конференции «Многопроцессорные вычислительные и управляющие системы (МВУС'2007)». Таганрог: НИИ МВС. 2007. Т.1, с. 198–203. http://is.ifmo.ru/verification/_jaminov.pdf