Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Кафедра «Компьютерные технологии»

Б.Р. Яминов

Автоматизация верификации автоматных *UniMod*-моделей на основе инструментального средства *Bogor*

Бакалаврская работа

Руководитель – В.С. Гуров

Санкт-Петербург
2007

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Автоматные модели программ [1] широко используются для создания *реактивных* (или *реагирующих*) [2] и управляющих систем, поскольку автоматы позволяют просто и наглядно представлять логику работы системы. Реактивные системы часто используются в таких областях, где цена ошибки чрезвычайно велика [3]. По этой причине возникла необходимость автоматически проверять соблюдение автоматной моделью заданных свойств.

Одной из техник автоматической проверки моделей программ является *Model Checking* [4, 5]. *Model Checking* – это автоматизированная техника, которая для заданной модели поведения системы с конечным числом состояний и логического свойства (требования) проверяет, выполняется ли оно на модели. Проверяемые свойства могут формулироваться в *темпоральной логике* – определять не только мгновенное состояние системы, но также и историю развития системы во времени.

Идею метода *Model Checking* реализуют специальные программы – верификаторы. Такая программа получает на вход модель, описанную на входном языке верификатора, и свойство, которое проверяется (возможно, на другом языке). На выходе программа формирует либо сообщение о том, что указанное свойство на заданной модели выполняется, либо сценарий, приводящий к его нарушению. Заметим, что одно сообщение о невыполнении свойства является недостаточным, так как не дает возможности решить целевую задачу – исправить модель.

Существуют работы по *ручной* верификации автоматных моделей [5 – **Ошибка! Источник ссылки не найден.**]. В них модель и требования к ней вручную транслируются во входной язык выбранного верификатора, а затем полученный сценарий, приводящий к ошибке, вручную транслируется обратно в модель. Однако до сих пор не создано верификатора, который *автоматически*

верифицировал автоматные модели. Цель настоящей работы – **создать автоматический верификатор программ рассматриваемого класса**.

К верификатору предъявляются следующие требования.

- На вход верификатор получает следующие сущности:
  а. Автоматную модель. Определяются формат модели и способ ее подачи.
  б. Верифицируемое свойство модели. Определяются формат свойства и способ его подачи. Должна быть обеспечена возможность формулировать *темпоральные* свойства модели.
- Время работы программы может зависеть от размеров модели и сложности свойства, но должно быть конечным.
- В ходе работы на экран должны выводиться сообщения о работе верификатора.
- Работа верификатора завершается либо когда он прошел все истории работы автоматной модели и не нашел нарушений заданного свойства, либо при обнаружении ошибок. Должна иметься возможность задавать число ошибок, после нахождения которых верификатор остановит поиск.
- Верификатор должен корректно проверять выполнимость заданного свойства на модели.
- На выход верификатор должен выводить:
  а. В случае, если свойство выполняется, сообщение об удачном завершении верификации.
  б. Если свойство не выполняется, то должен выводиться подробный сценарий действий над исходной автоматной моделью, приводящих к нарушению свойства (их может быть несколько).
- Сценарий состоит из списка шагов автоматной модели. Шаг – это процесс обработки одного события. Для каждого шага должны указываться:
  а. Обрабатываемое событие.
  б. Список переходов, активированных в ходе выполнения шага.
  в. Состояние каждого автомата модели после выполнения шага.

- Сценарий отображает путь в автоматной модели, который приводит к доказательству нарушения свойства, в соответствии с алгоритмом верификатора. Задача интерпретации сценария возлагается на пользователя.

ГЛАВА 1. ТЕОРЕТИЧЕСКОЕ ВВЕДЕНИЕ И АЛГОРИТМ ВЕРИФИКАЦИИ

В данной главе будет введена необходимая терминология и описан один из алгоритмов верификации темпоральных формул на модели Крипке, которые используются в главе 2.

1.1. Модель Крипке

В *Model checking* верификация модели производится путем построения *модели Крипке* [4]. Модель Крипке представляет собой граф переходов из атомарных (элементарных) состояний модели. Приведем более формальное определение модели Крипке.

Рассмотрим множество атомарных высказываний, которое обозначим как *AP*. Модель Крипке *M* над множеством *AP* – это четверка S, S_0, R, L, где

1. S – конечное множество состояний;
2. $S_0 \subseteq S$ – множество начальных состояний;
3. $R \subseteq S \times S$ – тотальное отношение переходов. «Тотальное» означает, что из каждого состояния существует хотя бы один переход.
4. $L: S \to 2^{AP}$ – функция, которая для каждого состояния возвращает множество атомарных высказываний, выполняемых в этом состоянии.

Путем из состояния s в модели Крипке называется бесконечная последовательность $s_0 s_1 s_2 \ldots$ такая, что $s_0 = s$ и для любого $i \geq 0$ существует переход $R(s_i, s_{i+1})$.

На модели Крипке проверяются различные свойства. Простой пример такого свойства: «Ни в одном состоянии не выполняется утверждение *p*». Это предикат первого порядка, и проверить его можно перебором всех состояний модели Крипке. Более интересными являются *темпоральные* свойства модели.

1.2. Темпоральные логики

Темпоральные свойства модели выражаются в виде формул темпоральной логики. Такими формулами задаются свойства *пути* в модели Крипке. Так как пути бесконечны, для проверки темпоральных свойств требуются специальные алгоритмы.

Простой пример темпорального свойства: «Для любого пути предикат p выполняется бесконечное число раз». Такое утверждение выполняется, например, для пути следующего вида (в состояниях с p в верхнем индексе выполняется предикат p):

$$s_0 \to s_1 \to \ldots \to s_i^{\,P} \to s_{i+1} \to \ldots \to s_j \to s_{j+1}^{\,P} \to \ldots \to s_k^{\,P} \to \ldots$$

Существует несколько темпоральных логик, в настоящей работе будет использоваться темпоральная логика *LTL* (*Linear Temporal Logic*) [4]. Опишем ее операторы.

Формулы *LTL* строятся из множества пропозиционных формул p_i, стандартных логических операторов $\neg$ («не»), $\cup$ («или»), $\cap$ («и»), $\to$ («следует») и следующих темпоральных операторов:

- **X** (neXt) – «*X p*» выполняется для тех путей, во втором состоянии которых выполняется *p*;
- **G** (Globally) – «*G p*» выполняется для тех путей, в каждом состоянии которых выполняется *p*;
- **F** (Future) – «*F p*» выполняется для тех путей, в которых существует хотя бы одно состояние, в котором выполняется *p*;
- **U** (Until) – «*p U q*» выполняется для тех путей, в которых существует состояние, где выполняется q, и в каждом предыдущем состоянии (до первого q) выполняется *p*;
- **R** (Release) – «q R p» выполняется для тех путей, в которых p выполняется до тех пор, пока не станет выполняться q (включительно), или всегда, если q не выполняется никогда.

В работе [4] доказано, что операторы **F**, **G** и **R** можно выразить через другие операторы.

1.3. Автоматы Бюхи

Один из методов верификации темпоральных свойств на модели Крипке – построение по *LTL* формуле автомата Бюхи, и использование его для поиска контрпримера.

Автоматы Бюхи [4] – одни из простейших конечных автоматов над бесконечными словами (конечные автоматы над бесконечными словами – ω-автоматы). Путь в автомате Бюхи называется допускающим, если он бесконечное число раз проходит через хотя бы одно допускающее состояние автомата. Автомат Бюхи допускает слова, порождаемые допускающим путем. Например, автомат на рис.1 имеет допускающее состояние, обозначенное двойным кружком. Он допускает слова *a**, *(ab)**, но не допускает *aaab**.

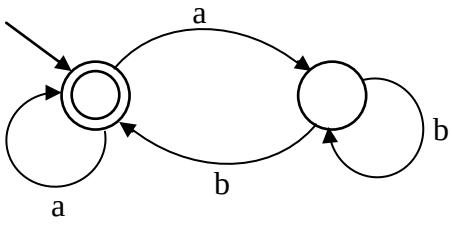

Рис. 1. Пример автомата Бюхи

Приведем более формальное определение. Автомат Бюхи – это пятерка **<Σ, Q, Δ, Q^0, F>**, где

- **Σ** – конечный алфавит;
- **Q** – конечное множество состояний;
- **Δ ⊆ Q x Σ x Q** – отношение переходов;
- **Q^0 ⊆ Q** – множество начальных состояний;
- **F ⊆ Q** – множество *допускающих* состояний.

Пусть задано некоторое слово (последовательность) v из Σ^* длины $|v|$. Тогда *проходом* автомата на слове v назовем всякое такое отображение $\rho: \{0, 1, ..., |v|\} \to Q$, что

- первое состояние является начальным. При этом $\rho(0) \in Q^0$;
- каждый переход осуществляется по соответствующему символу слова v. При этом $\forall i: 0 \le i \le |v|, (\rho(i), v(i), \rho(i+1)) \in \Delta$.

Обозначим через $inf(\rho)$ множество состояний, встречающихся бесконечно часто в ρ. Тогда проход называется допускающим, если $inf(\rho) \cap F \neq \varnothing$.

Автомат допускает слово v тогда и только тогда, когда существует хотя бы один допускающий проход на слове v.

Оказывается, любую формулу в логике *LTL* можно преобразовать в недетерминированный автомат Бюхи [4]. Это означает, что множество путей, удовлетворяющих формуле и допускаемых автоматом Бюхи, будут совпадать. Например, формула $\boldsymbol{FG} \neg p$ («Существует состояние, после которого p никогда не выполнится») преобразуется в автомат Бюхи, изображенный на рис. 2.

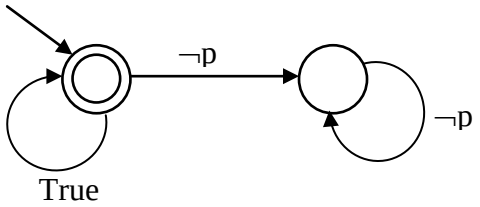

Рис. 2. Автомат Бюхи для формулы $G\neg p$

Из рассмотрения этого графа следует, что этот автомат является недетерминированным: он недетерминированно выбирает момент, начиная с которого p никогда не будет выполняться, и переходит в недопускающее состояние.

Модель Крипке достаточно просто можно преобразовать в ω-автомат, все состояния которого будут допускающими. Языком полученного автомата будут пути модели Крипке. Также известно, что можно построить ω-автомат,

допускающий пересечение языков двух ω-автоматов [4]. Тогда возникает следующая идея: построить два автомата Бюхи, из *отрицания* исходной *LTL*-формулы и из модели Крипке, и после этого их перемножить. Тогда любое слово, допускаемое полученным автоматом, будет одновременно путем в модели Крипке, и не будет удовлетворять исходной *LTL*-формуле. Следовательно, это и будет искомым контрпримером (путем в модели Крипке, нарушающим свойство). С другой стороны, если контрпример существует, то он будет допускаться в обоих автоматах Бюхи, и, следовательно, будет и их пересечением.

Приведенная идея используется во многих верификаторах *LTL*-формул. Повторим, что требуется для ее реализации:

1. Преобразовать *отрицание LTL*-формулы в автомат Бюхи.
2. Построить пересечение полученного автомата и модели Крипке (ее автоматного представления).
3. Найти допускающий путь в полученном пересечении или убедиться, что его не существует.

Для трансляции *LTL*-формулы в автомат Бюхи был предложен алгоритм [4], и в открытом доступе существуют программы, его реализующие (например, программа *ltl2ba*).

Построение пересечения (перемножение) автомата Бюхи и модели Крипке можно делать и неявно по правилам:

- Состояние пересечения – комбинация состояния модели Крипке и состояния автомата Бюхи.
- Каждый шаг в пересечении совершается в два действия: сначала происходит переход в модели Крипке, а затем в автомате Бюхи. Таким образом, они работают «параллельно».
- Если автомат Бюхи неполный (это бывает при использовании программы *ltl2ba*), то при отсутствии активных переходов текущее

состояние направляется в «сток» (недопускающее состояние с единственным переходом-петлей[*]).

- Если активных переходов несколько, то создается ветвление по активным переходам как в модели Крипке, так и в автомате Бюхи. Например, если в текущем состоянии системы активны два перехода в автомате Бюхи и три перехода в модели Крипке, то для системы-пересечения будет ветвление из текущего состояния в шесть переходов.
- Если автомат Бюхи оказался в допускающем состоянии, то и пересечение автоматов оказалось в допускающем состоянии.

Теперь опишем алгоритм проверки на пустоту языка полученного пересечения (напомним, что пересечение – тоже автомат Бюхи).

1.4. Алгоритм проверки на пустоту языка автомата Бюхи

Пусть p – допускающий проход автомата Бюхи. Так как множество состояний автомата конечно, то найдется суффикс p' прохода p, такой что всякое состояние из него встречается бесконечное число раз. Это означает, что любое состояние этого суффикса достижимо из любого другого состояния суффикса. Следовательно, состояния из p' входят в состав некоторой сильно связной компоненты графа, описывающего автомат Бюхи. Причем, так как одно из допускающих состояний также входит в p', то сильно связная компонента содержит допускающее состояние.

Обратно, если в графе автомата Бюхи существует достижимая сильно связная компонента, содержащая допускающее состояние, то легко построить допускающий путь. Он будет иметь структуру $\alpha\beta^*$, где α – префикс, ведущий к сильно связной компоненте, а β – цикл в этой компоненте, содержащий допускающее состояние. Структура допускающего пути изображена на рис. 3.

[*] В рамках настоящей работы в верификаторе *Bogor* была исправлена ошибка, связанная с невыполнением этого правила.

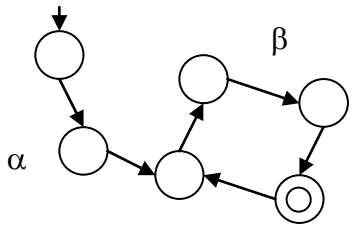

Рис. 3. *Структура допускающего пути*

Таким образом, проверка пустоты языка автомата Бюхи равносильна поиску сильно связной компоненты, достижимой из начального состояния и содержащей допускающее состояние. Для этого используется алгоритм *двойного DFS* [4] (*Depth-First Search*, поиск в глубину).

В этом алгоритме чередуются два поиска в глубину. Первый из них может запускать второй, а второй, в свою очередь, может либо завершить работу всего алгоритма, либо передать управление обратно в первый *DFS*. В этом случае первый поиск в глубину продолжает свою работу. Каждый *DFS* используют свой флаг для пометки посещенных состояний.

Первый *DFS* запускает второй в тот момент, когда он готов к откату из допускающего состояния. Если второй *DFS* в процессе обхода попадает в состояние, находящееся в стеке первого *DFS,* то допускающий путь получен. Если этого не происходит, то после завершения обхода второй *DFS* возвращает управление в первый.

Более формально алгоритм проверки на пустоту языка автомата Бюхи с помощью двойного *DFS* выглядит следующим образом (на псевдокоде, взято из [4]):

```
procedure emptiness
    for all q0 ∈ Q0 do
        dfs1(q0);
    terminate(False);
end procedure

procedure dfs1(q)
```

```
    local q';
    hash(q);
    for all последователей q' вершины q do
        if q' не содержится в хэш-таблице then dfs1(q');
    if accept(q) then dfs2(q);
end procedure

procedure dfs2(q)
    local q';
    flag(q);
    for all последователей q' вершины q do
        if q' в стеке dfs1 then terminate(True);
        else if q' не является помеченной then dfs2(q');
        end if;
end procedure
```

Алгоритм возвращает значение `True`, если был найден допускающий путь, и `False` – в противном случае. Если алгоритм вернул значение `True`, то можно восстановить допускающий путь: в стеке первого *DFS* хранится путь из начального состояния в некоторое допускающее состояние *q1*. Этот путь и будет искомым префиксом α. При этом в стеке второго *DFS* хранится путь из состояния *q1* в некоторое состояние *q2*, содержащееся в стеке первого. Тогда, достроив этот путь состояниями, находящимися в стеке первого *DFS* выше состояния *q2*, получим цикл $q1 \to q2 \to q1$, проходящий через допускающее состояние *q1*, а, следовательно, искомый суффикс β. Таким образом, будет получен допускающий путь $\alpha\beta^*$.

Доказательство корректности алгоритма подробно описано в [4].

Выводы по главе 1

В данной главе была подробно описана общая схема верификации моделей программ с помощью построения автомата Бюхи, а также описан алгоритм

двойного *DFS*, который используется во многих верификаторах для поиска контрпримера. Знание этого алгоритма понадобится для понимания следующей главы, в которой будет описан процесс создания верификатора автоматных моделей программ.

ГЛАВА 2. СОЗДАНИЕ ВЕРИФИКАТОРА АВТОМАТНЫХ МОДЕЛЕЙ

В этой главе будет подробно описан новый метод построения верификатора автоматных моделей программ и детали его реализации в настоящей работе.

2.1. Верификаторы

Так как верификация производится с помощью модели Крипке, по для решения задачи верификации любой программы необходимо решить три подзадачи.

1. Транслировать исходную программу в модель Крипке.
2. Транслировать исходные требования к программе в требования к модели Крипке (транслировать темпоральную формулу).
3. Транслировать полученный в модели Крипке путь, приводящий к ошибке, обратно в программу.

Эти подзадачи и собственно задачу верификации модели Крипке решают специальные программы – *верификаторы*.

Проверяемая программа описывается на *входном языке* верификатора. Обычно у каждого верификатора свой входной язык, хотя, в общем, все они похожи. Часто входной язык пытаются сделать похожим на обычный язык программирования (например, у верификатора *SPIN* [8] *C*-подобный входной язык *Promela*). Это делается, для того чтобы пользователю было проще вручную перенести код программы во входной язык верификатора. Верификатор разбивает написанный код на атомарные действия и состояния и строит модель Крипке. Возможен также и другой подход, при котором входной язык верификатора примитивен, и задача выделения атомарных действий и состояний возлагается на пользователя. Такой подход полезен тем, что позволяет контролировать генерируемую модель Крипке. Поэтому возможно сократить число ее состояний по сравнению с первым подходом (когда программа моделировалась на языке

высокого уровня). Создавать модель программы на языке низкого уровня позволяет, например, верификатор *Bogor* [9].

Стоит заметить, что, хотя выше и употреблялся термин «построение модели Крипке», в реальных верификаторах она обычно явно не строится.

2.2. Сравнение способов решения задачи о верификации автоматной модели

Вернемся к поставленной задаче. Требуется верифицировать темпоральные свойства автоматной модели. Возможны три подхода.

1. Выполнять трансляцию автоматной модели и требований к ней во входной язык одного из верификаторов. Если верификатор находит ошибку, то транслировать полученный сценарий ошибки обратно в автоматную модель.
2. Написать свой верификатор, реализовав самостоятельно все необходимые аспекты верификации.
3. Комбинированный подход: использовать готовый верификатор, четко определив правила создания внутренней модели Крипке по автоматной модели.

Сравним эти три подхода.

Первый подход реализован в нескольких работах [6, 7]. В них использовался верификатор *SPIN*, однако решение всех трех подзадач трансляции производилось вручную. Заметим, что в этом подходе производится двойная трансляция: между автоматной моделью и входным языком верификатора, и между входным языком верификатора и внутренней моделью Крипке. В результате двойной трансляции в исходной модели может выделиться большое число «лишних» атомарных состояний, которые не влияют на результат верификации, однако, увеличивают размер модели Крипке. При этом отметим, что от размеров модели Крипке и от длины темпоральной формулы напрямую зависят необходимые для верификации время и память.

Второй подход (создание нового верификатора) лишен недостатков первого подхода и является, пожалуй, наиболее «чистым» решением поставленной задачи. Этот подход применяется в работе С. Э. Вельдера [5], однако в этой работе автоматическая верификация не выполнялась. К недостаткам этого подхода из работы [5] относится то, что пока все операции приходится реализовывать вручную, тогда, как уже существует достаточное число надежных и проверенных верификаторов.

Третий подход (смешанный) реализован в данной работе. Как было сказано выше, в этом подходе используется готовый верификатор, но для него определяются правила, по которым он будет строить модель Крипке для автоматной модели. Для реализации такого подхода требуется верификатор с достаточной степенью гибкости входного языка. Таким верификатором является *Bogor*.

2.3. Верификатор *Bogor*

Bogor [9, 10] – это верификатор с расширяемым входным языком и модульной структурой. Во входной язык этого верификатора (он называется *BIR – Bogor Input Representation*) можно добавлять новые типы и абстракции, а сам верификатор разделен на модули, реализующие различные аспекты верификации (такие как алгоритм обхода, кодирование состояний и т.д.). Эти модули достаточно просто можно заменять другими при необходимости изменить логику верификатора, не переписывая его. Таким образом, верификатор *Bogor* является весьма гибким.

Верификатор *Bogor* разработан в *Kansas State University* и был успешно применен при решении реальных задач верификации, таких как, например, верификация программного обеспечения для самолета фирмы *Boeing* [11]. Кроме консольной версии этого верификатора существует также плагин для среды разработки *Eclipse* с удобным интерфейсом.

Ядро верификатора *Bogor* не поддерживает верификацию темпоральных свойств, однако его разработчиками было создано несколько модулей, которые дают возможность проверять свойства, заданные в темпоральной логике *LTL*. Также создателями верификатора были разработаны примеры расширения входного языка верификатора.

Расширение входного языка – это создание в нем новых классов (типов). Новые классы используются для того, чтобы абстрагироваться от несущественных деталей реализации верифицируемой программы и сконцентрироваться лишь на той части логики программы, в которой ожидается возникновение ошибки.

Обычно во входных языках верификаторов поддерживаются лишь элементарные типы данных, такие как булевы, целочисленные, строки, массивы. Поэтому при моделировании программы, работающей с более сложными структурами данных (например, «множество»), возникает подзадача моделирования таких структур во входном языке верификатора. Это порождает следующие проблемы.

- Крайне затруднительно реализовать достаточно простые действия над структурой данных, поскольку, как правило, входные языки верификаторов имеют весьма ограниченную выразительность по сравнению с языками программирования, такими, как, например, *Java*. Даже организация простого цикла уже может стать нетривиальной задачей. Следовательно, создается громоздкий код, в котором будет сложно разобраться при анализе сценария ошибки.
- Основная проблема состоит в том, что построенная модель программы может оказаться слишком детализированной, в то время как для проверки требования к программе обычно неважны детали реализации операций над стандартными структурами данных (например, операция удаления элемента из множества). Из-за такой детализации повышается сложность верификации программы.

Эти проблемы показывают насколько важно иметь возможность создавать новые абстракции во входном языке верификатора.

2.4. Создание нового класса в *Bogor*

При создании нового класса в языке *BIR* требуется специфицировать некоторые его свойства [10], для того чтобы алгоритм обхода модели Крипке смог корректно работать с объектами нового класса. Опишем эти свойства.

Несмотря на то, что алгоритм двойного *DFS* работает с графом модели Крипке, в верификаторе граф явно не строится. Вместо этого состояние системы изменяется динамически, что соответствует переходам в модели Крипке. Поэтому в каждом состоянии системы должен быть определен ответ на следующие вопросы:

1. Как определить, посещалось ли уже данное глобальное состояние системы?
2. Сколько для данного состояния вариантов существует ветвей в модели Крипке?
3. Как шагнуть вперед по одной из ветвей?
4. Как шагнуть назад, чтобы откатиться из посещенного состояния?

Так как состояние системы в верификаторе *Bogor* складывается из состояний всех глобальных переменных и состояний каждого потока, объявленных в *BIR*-спецификации, для корректной работы алгоритма двойного *DFS* требуется уметь отвечать на перечисленные вопросы для каждого типа переменных и для каждых действий.

Для нового класса ответы на эти вопросы реализуются в *Java*-классах. Сначала специфицируется пункт 1. Для этого определяется метод, который возвращает `long[][]` – двумерный массив чисел, кодирующий состояние. Если два массива одинаковы, то состояния объектов, породивших их, принимаются в верификатор *Bogor* как одинаковые.

Затем для нового типа создаются методы. Они бывают двух типов:

- `expdef` – методы, которые возвращают значение и не изменяют внутреннего состояния объекта.
- `actiondef` – производят действия над объектом и изменяют его состояние, не возвращают значений.

Для методов обоих типов возможна спецификация пункта 2 (ветвление). Например, можно создать недетерминированный метод «`expdef` *select*» для класса «Множество», который будет возвращать случайный его элемент. Для методов `actiondef`, кроме того, требуется спецификация пунктов 3 и 4, поскольку они изменяют состояние объекта.

В настоящей работе был создан новый класс `AutomataModel`, объекты которого представляют собой автоматную модель. Подробно опишем его.

2.5. Класс *AutomataModel*

AutomataModel – это класс, объекты которого представляют собой *изолированную* автоматную модель. «Изолированная» означает, что от исходной реактивной системы отцепляются источники событий и объекты управления, и остается «чистая» автоматная модель. Опишем для нового класса реализацию четырех пунктов, упомянутых в разд. 2.4.

1. Состояние объекта составляется из состояний каждого автомата, входящего в автоматную модель. Таким образом, предполагается, что одинаковый набор состояний автоматов означает одинаковое поведение автоматной модели. Вообще говоря, данное утверждение может нарушаться, если часть логики работы реактивной системы содержится в объектах управления. Подробнее эта проблема будет рассмотрена в разд. 3.2.
2. Ветвление историй происходит в ходе выполнения действия `step` – шаг обработки автоматной моделью одного события.
    a. Сначала у автоматной модели запрашивается множество событий, которые обрабатываются в данном состоянии. Из

этого множества недетерминированно выбирается одно и подается на вход автоматной модели. Таким образом, возникает ветвление по входящему событию.

б. При выборе активных переходов в автоматах требуется вычислять условия на переходах. В выражениях условий участвуют значения предикатов объектов управления, но сами объекты управления не участвуют в верификации. Поэтому создается ветвление по возможным значениям каждого условия: `True` или `False`.

3. Шаг вперед – это реакция автоматной модели на одно событие. В один шаг могут быть включены несколько переходов в нескольких автоматах. Поэтому при обработке события в специальный контейнер `StepInfo` записывается следующая информация:
   - номер текущего шага в истории;
   - полученное событие;
   - осуществленные переходы (для всех автоматов);
   - значения вычисленных условий на переходах;
   - список вызванных действий объектов управления, в порядке вызова;
   - состояния автоматов после завершения шага.
4. Для шага назад из стека извлекается контейнер `StepInfo`, сохраненный на предыдущем шаге, и автоматы переводятся в предыдущие состояния.

При выборе значений условий на переходах используется анализатор, который исключает заведомо невозможные комбинации значений. Например, если из одного состояния два перехода помечены условиями `o1.x1` и `!o1.x1`, то они не могут одновременно быть вычислены как `False`.

Объявление класса `AutomataModel` в *BIR*-файле имеет следующим вид:

```
  extension AutomataModel for
      com.unimod.verifier.bogorextension.AutomataModelModule
  {
    /* Automata model type */
    typedef type;

    /* Creates new automata model. Model's filename is requested from
     * UNIMOD_MODEL_FILENAME property of bogor-configuration. */
    expdef AutomataModel.type create();

    /* Returns whether specified event was received during the last step */
    expdef boolean wasEvent(AutomataModel.type model, string event);

    /* Returns whether specified state machine was in specified state
     * before the last step */
    expdef boolean wasInState(AutomataModel.type model, string
stateMachine, string state);

    /* Returns whether specified state machine is in specified state after
     * the last step */
    expdef boolean isInState(AutomataModel.type model, string stateMachine,
string state);

    /* Returns whether specified state machine came to specified state
     * during the last step. If state machine was already in specified
     * state before the last step then returns false */
    expdef boolean cameToState(AutomataModel.type model, string
stateMachine, string state);

    /* Returns whether root state machine came to its final state during
     * the last step */
    expdef boolean cameToFinalState(AutomataModel.type model);

    /* Returns whether the action was executed during the last step */
    expdef boolean wasAction(AutomataModel.type model, string action);

    /* Returns whether action specified was the first to be executed during
     * the last step */
    expdef boolean wasFirstAction(AutomataModel.type model, string action);

    /* Returns whether action specified was the last to be executed during
     * the last step */
    expdef boolean wasLastAction(AutomataModel.type model, string action);

    /* Returns index of action in the list of actions executed during the
     * last step. This method is used to check if one action was executed
     * before another. If the action was not executed at all then method
     * returns -1 */
    expdef int getActionIndex(AutomataModel.type model, string action);

    /* Returns if specified guard was evaluated as true during last
     * transition */
    expdef boolean wasTrue(AutomataModel.type model, string guard);

    /* Returns if specified guard was evaluated as false during last
     * transition */
```

```
    expdef boolean wasFalse(AutomataModel.type model, string guard);

    /* Performs indeterministic step of the model. Chooses one of
     * possible and not yet chosen events and fires it into the model. */
    actiondef step(AutomataModel.type model);
  }
```

В объявлении класса перечисляются его методы. Отметим, что методы вызываются не у объекта, а у класса (аналог `static`-методов в *Java*). Объект, над которым производится действие, передается как параметр.

Опишем подробно элементы объявления класса `AutomataModel`.

extension AutomataModel **for** com.unimod.verifier.bogorextension.AutomataModelModule	Объявление нового типа и указание *Java*-класса, который его реализует.
typedef type;	Поле, обозначающее новый тип. Используется в *BIR*-файле при объявлении переменной в виде «`AutomataModel.type model`»;
expdef AutomataModel.type *create*();	Создает новый объект данного типа. Используется в виде `model := AutomataModel.create();` Путь к файлу, описывающему автоматную модель, передается в Bogor в специальном файле настроек, минуя *BIR*-спецификацию.
actiondef *step*(AutomataModel.type model);	Производит в передаваемой автоматной модели `model` один шаг выбора и обработки события.
expdef boolean *wasEvent*(AutomataModel.type model, string event);	Возвращает `True`, если в последнем шаге было выбрано для обработки событие с названием `event`, и `False` в противном случае.
expdef boolean *wasInState*(AutomataModel.type model, string stateMachine, string state);	Возвращает `True`, если перед последним шагом автомат, кодируемый *путем* `stateMachine`, находился в состоянии с

	названием `state`. Что такое путь автомата в модели пояснено ниже.
expdef boolean *isInState*(AutomataModel.type model, string stateMachine, string state);	Возвращает `True`, если после совершения последнего шага автомат, кодируемый *путем* `stateMachine`, находится в состоянии с названием `state`.
expdef boolean *cameToState*(AutomataModel.type model, string stateMachine, string state);	Возвращает `True`, если после совершения последнего шага автомат, кодируемый *путем* `stateMachine`, сменил свое состояние на `state`. То же, что `(isInState && !wasInState)`.
expdef boolean *cameToFinalState*(AutomataModel.type model);	Возвращает `True`, если после совершения шага корневой автомат модели вошел в свое конечное состояние. Это означает, что модель завершила работу.
expdef boolean *wasAction*(AutomataModel.type model, string action);	Возвращает `True`, если в ходе выполнения шага было вызвано выходное воздействие `action` (например, «`o1.x1`»).
expdef boolean *wasFirstAction*(AutomataModel.type model, string action);	Возвращает `True`, если в ходе выполнения шага первым вызванным действием было `action`.
expdef boolean *wasLastAction*(AutomataModel.type model, string action);	Возвращает `True`, если в ходе выполнения шага последним вызванным действием было `action`.
expdef int *getActionIndex*(AutomataModel.type model, string action);	Возвращает номер действия в списке действий, вызванных в ходе выполнения последнего шага. Если указанное действие не выполнялось, возвращается «`-1`». Данный

	метод предназначен для того, чтобы формулировать утверждения, задающие порядок выполнения действий в автоматной модели.
expdef boolean *wasTrue*(AutomataModel.type model, string guard);	Возвращает `True`, если в ходе выполнения последнего шага на одном из переходов было встречено условие `guard`, и его значение было определено как `True`.
expdef boolean *wasFalse*(AutomataModel.type model, string guard);	Возвращает `True`, если в ходе выполнения последнего шага на одном из переходов было встречено условие `guard`, и его значение было определено как `False`.

Поясним, что такое *путь* автомата в модели. Так как автоматная модель представляет собой иерархическую систему автоматов, одни могут быть вложены в состояния других. При этом одинаковые автоматы могут быть вложены в различные состояния, и одного имени становится недостаточно, чтобы их идентифицировать. Поэтому вводится понятие *пути автомата*, который строится по следующим правилам:

- путь корневого автомата имеет формат «/<*название корневого автомата*>»,
- путь вложенных автоматов имеет формат «<*путь родительского автомата*>:<*состояние родительского автомата*>/<*название вложенного автомата*>».

К примеру, если автомат `A1` вложен в состояние `s2` корневого автомата `A`, то путь автомата `A1` – это «`/A:s2/A1`».

За счет создания нового класса исполнимая часть *BIR*-спецификации сводится к минимуму: инициализации глобальной переменной типа

AutomataModel, а затем бесконечному циклу, вызывающему метод step у этой модели:

```
AutomataModel.type model;

main thread MAIN() {
  loc init:
    do invisible {
      model := AutomataModel.create();
    } goto loop;

  loc loop:
    do {
      AutomataModel.step(model);
    } goto loop;
}
```

Здесь описывается модель программы для верификатора *Bogor*. Программа состоит из двух состояний: состояния init, в котором инициализируется автоматная модель и происходит переход в состояние loop, и состояния loop, в котором происходит шаг автоматной модели и переход в себя. Модификатор invisible при вызове инициализации обеспечит, что это действие произойдет «незаметно» для верифицируемого свойства. Это требуется для того, чтобы при верификации свойство не проверялось на еще не созданной модели.

Опишем теперь, как формулируются свойства модели. Они записываются в *BIR*-файл с помощью расширения языка *BIR*, позволяющего формулировать утверждения в темпоральной логике *LTL*. К примеру, свойство «автомат A никогда не попадет в состояние Error» записывается в *BIR*-файле следующим образом:

```
LTL.temporalProperty(
  Property.createObservableDictionary(
    Property.createObservableKey(
        "is_Error", AutomataModel.isInState(model, "/A", "Error"))
  ),

  LTL.always(
    LTL.negation(LTL.prop("is_Error"))
  )
);
```

Сначала создается пропозиционная формула «is_Error», которой соответствует вызов соответствующего метода AutomataModel. Затем эта формула используется в *LTL*-формуле. Приведенная выше запись соответствует формуле ***G*¬**(is_Error) (всегда не ошибка). Приведем полный список темпоральных операторов в языке *BIR*.

expdef LTL.Formula *always*(LTL.Formula);	**G** (Globally, всегда)
expdef LTL.Formula *eventually*(LTL.Formula);	**F** (Future, когда-нибудь в будущем)
expdef LTL.Formula *negation*(LTL.Formula);	¬ (отрицание)
expdef LTL.Formula *next*(LTL.Formula);	**X** (neXt, в следующий момент времени). Сначала этот оператор отсутствовал в *LTL*-расширении языка *BIR*, однако он был туда добавлен в настоящей работе.
expdef LTL.Formula *until*(LTL.Formula, LTL.Formula);	**U** (Until, до тех пор, пока)
expdef LTL.Formula *weakUntil*(LTL.Formula, LTL.Formula);	**W** (Weak until). Этот оператор был добавлен в настоящей работе в *LTL*-расширение языка *BIR* для удобства. $p\ W\ q = (p\ U\ q) \cup G\ (p \cap \neg q)$, то есть это то же самое, что $p\ U\ q$, однако q не обязано когда-либо выполниться.
expdef LTL.Formula *release*(LTL.Formula, LTL.Formula);	**R** (Release, освобождение)
expdef LTL.Formula *equivalence*(LTL.Formula, LTL.Formula);	↔ (Эквивалентно). $p \leftrightarrow q = (p \rightarrow q) \wedge (q \rightarrow p)$
expdef LTL.Formula	→ (Следует)

implication(LTL.Formula, LTL.Formula);	
expdef LTL.Formula conjunction(LTL.Formula, LTL.Formula);	$\cap$ (И)
expdef LTL.Formula disjunction(LTL.Formula, LTL.Formula);	$\cup$ (Или)

С помощью перечисленных операторов можно задать любую *LTL*-формулу, описывающую требование к автоматной модели.

Как объяснялось в разд. 1.3, *LTL*-формула преобразуется в автомат Бюхи, который потом используется при верификации. В языке *BIR* имеется возможность сразу записывать требования в терминах автомата Бюхи. Например, приведенная выше формула $\boldsymbol{G}\neg$(is_Error) будет преобразована в следующий автомат:

```
function generated$FSA()
{
    loc T0_init:
        when true do
        {
        }
        goto T0_init;
        when AutomataModel.isInState(model, "/A", "Error") do
        {
        }
        goto bad$accept_all;
    loc bad$accept_all:
        when true do
        {
        }
        goto bad$accept_all;
}
```

Состояния сгенерированного автомата, названия которых начинаются на «bad$», являются допускающими.

2.6. Интерпретатор *UniMod*

Работа с автоматной моделью происходит с помощью инструментального средства *UniMod* [12, 13]. *UniMod* – это инструментальное средство,

предназначенное для создания и работы реактивных систем, оформленное как плагин к среде разработки *Eclipse*. Это инструментальное средство позволяет изображать автоматы в виде *UML*-диаграмм, в то время как источники событий и объекты управления реализуются в виде *Java*-классов.

В классе `AutomataModel` работа с автоматной моделью осуществляется напрямую через интерпретатор *UniMod*. Во-первых, такой подход избавляет от необходимости создавать собственный интерпретатор реактивных систем с иерархически связанными автоматами. При данной реализации верификатор может работать с любыми автоматными моделями, которые можно построить в *UniMod*. Во-вторых, возникает возможность получить единое средство для создания, интерпретации и верификации реактивных систем. Кроме того, использование одного интерпретатора, как для работы реактивной системы, так и для ее верификации, позволяет избежать ошибок несоответствия.

Выводы по главе 2

В этой главе был описан новый метод создания верификатора автоматных моделей программ и приведена реализация этого метода с помощью верификатора *Bogor* и инструментального средства *UniMod*. В следующей главе будет приведено сравнение результатов с другими работами, сделанными по данной теме, а также приведен пример, демонстрирующий работу созданного верификатора.

ГЛАВА 3. СРАВНЕНИЕ С ДРУГИМИ РАБОТАМИ

В этой главе выполнено сравнение построенного верификатора с другими работами, выполненными по теме верификации автоматных моделей программ. Сначала рассмотривается построенная модель Крипке, затем сформулирована проблема, связанная с логикой, хранящейся в объектах управления. В завершение главы будет продемонстрирована работа верификатора на простом примере.

3.1. Построенная модель Крипке

В работах [5, 6] модель Крипке строилась из автомата следующим образом. Каждый переход в автомате разбивался на атомарные действия, и создавалась цепочка переходов из одного состояния автомата в другое через цепь вспомогательных состояний.

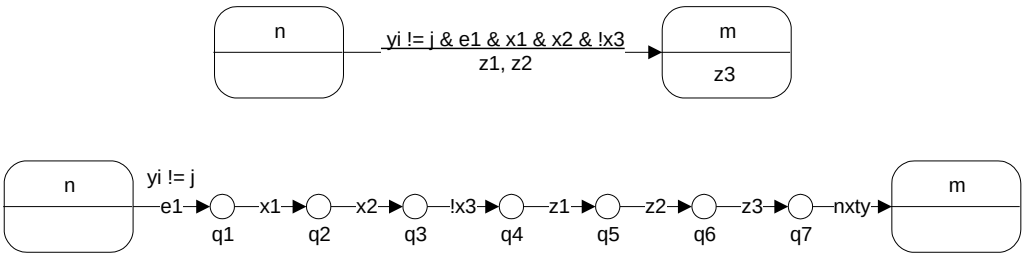

Рис. 4. *Выделение промежуточных состояний для перехода (из работы [6])*

Однако для каждого перехода автомата создается новая цепь, и поэтому у каждого промежуточного состояния лишь по одному входящему и исходящему переходу. В ходе обхода модели Крипке нельзя попасть в промежуточное состояние, не пройдя через глобальное состояние, из которого начинается цепь. Тогда можно утверждать, что промежуточное состояние посещено тогда и только тогда, когда посещено глобальное состояние, в которое ведет цепь промежуточных.

Отсюда следует, что в процессе обхода модели достаточно сохранять лишь множество посещенных *глобальных* состояний автоматной модели.

Промежуточные состояния необходимы лишь для проверки в них требований при обходе.

В данной работе промежуточные состояния явно не выделяются. Верификатор проходит сразу всю цепочку, а затем проверяет, не нарушено ли было проверяемое свойство в ходе перехода. Поэтому, если нарушение проверяемого свойства происходит в промежуточном состоянии, верификатор обнаружит нарушение с небольшим запозданием.

Обход модели Крипке в ходе верификации реализованным верификатором изображено на рис. 5.

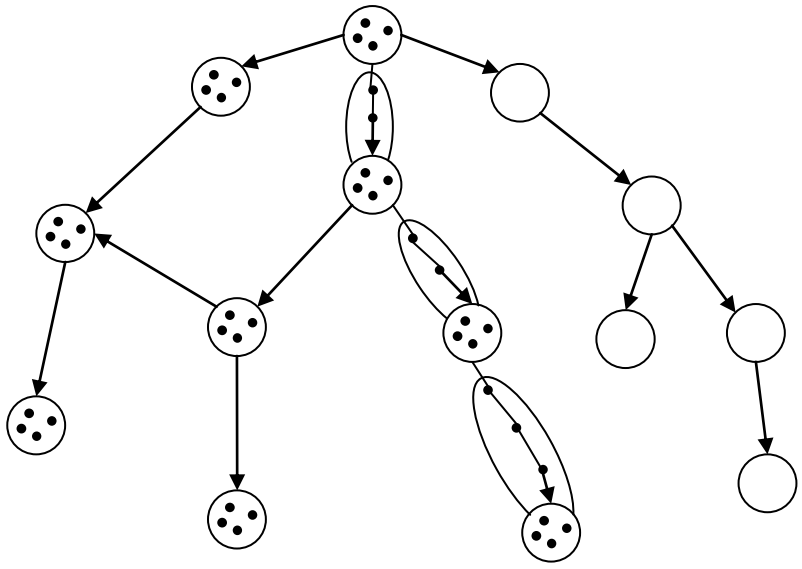

Рис. 5. *Схематичное изображение информации, хранимой в процессе верификации. Для обойденных состояний хранится набор состояний автоматов. Для состояний, находящихся в стеке, кроме того, хранится информация о пройденных промежуточных состояниях (*`StepInfo`*).*

На этом рисунке видно, что для пройденных состояний хранится только набор состояний автоматов (обозначено точками). Для состояний, находящихся в стеке хранится, кроме того, информация о промежуточных действиях, произведенных в ходе перехода в это состояние. Эта информация помещается в специальный контейнер `StepInfo`, о котором упоминалось в главе 2.

Если сравнивать данную реализацию с работами [5, 6], в которых явно выделяются промежуточные состояния, то можно отметить следующие ее преимущества и недостатки. Преимущества:

- меньшее число состояний в модели Крипке;
- нет необходимости выделять промежуточные состояния и транслировать сценарий ошибки;
- возможность применять оператор `X (neXt)`, тогда как при реализации с выделением промежуточных состояний невозможно применять оператор `X` применительно к глобальным состояниям автоматов.

Недостатки:

- сложность формулировки условий, накладывающих ограничения в пределах одного перехода. Вся возможная информация о промежуточных действиях хранится в `StepInfo`, и следовательно, выразительные возможности данной реализации не должны уступать выразительным возможностям метода, описанного в работах [5, 6]. Однако, например, если при явном выделении промежуточных состояний легко сформулировать условие на порядок вызываемых выходных воздействий, то без явного выделения приходится делать это через вызовы к `AutomataModel.`*`getActionIndex`*.

3.2. Проблема логики в объектах управления

В данной работе верифицируется изолированная автоматная модель: источники событий и объекты управления отцепляются от реактивной системы. Вместо источников событий события автоматной модели поставляет сам верификатор. Что же касается объектов управления, то вызовы к их выходным воздействиям регистрируются, но не выполняются, а значения входных воздействий выбираются верификатором. При этом делается предпосылка о том, что изолированная модель работает так же, как и настоящая модель.

Данная предпосылка верна в том случае, если вся логика работы программы хранится в автоматной модели, и объекты управления не влияют на работу автоматов. Однако это свойство может нарушаться.

Приведем простой пример. Пусть несколько однотипных автоматов (рис. 6) используют один ресурс. И пусть этот ресурс представлен в виде объекта управления.

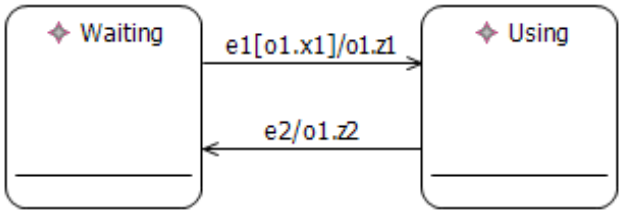

Рис. 6. *Простой автомат, использующий «ресурс».*

Объект управления `o1` имеет внутреннюю булеву переменную, которая символизирует, является ли ресурс в данный момент свободным. Значение ее возвращается входным воздействием `o1.x1`. Также определены действия: «занять ресурс» (`o1.z1`) и «освободить ресурс» (`o1.z2`), которые изменяют внутреннюю переменную. Ясно, что если один из автоматов занял ресурс, вызвав `o1.z1`, то ни один другой автомат не перейдет в состояние `Using`, поскольку не будет выполнено условие `o1.x1`.

Однако при отцеплении объектов управления эта логика теряется. Верификатор не может знать о том, что если было выполнено `o1.z1`, то `o1.x1` возвращает `False`. В результате при верификации несколько автоматов могут одновременно попасть в состояние `Using`, что невозможно в исходной модели. Таким образом, возможно получение *наведенных* ошибок верификации.

Кроме такого простого примера с внутренней булевой переменной в объекте управление, сложнее дело обстоит, если объект управления имеет входное переменные типа `int`. В этом случае число внутренних состояний объекта управления становится практически необозримым.

Проблема объектов управления с внутренней логикой, влияющей на переходы автомата, состоит не только в появлении фиктивных историй при их отцеплении. Важнее то, что в этом случае при верификации больше не работает предпосылка о том, что одинаковые состояния автоматов порождают одинаковое поведение реактивной системы. Действительно, система из одного автомата с одним состоянием и объектом управления «Счетчик» уже может находиться в бесконечном числе состояний (хотя не все эти состояния будут интересны, а лишь те классы состояний, которые приводят к разному поведению системы).

Для решения описанной проблемы можно применять несколько следующих идей.

- Если смоделировать тот же «Ресурс» в виде автомата, проблема решается сама собой. Логика объекта управления добавляется в верифицируемую модель в виде нового автомата из двух состояний.
- Ограничения на истории развития можно добавлять в верифицируемую формулу, в следующем виде:

Ограничение1 ∩ Ограничение2 ∩ … → Требование

  Таким образом, фактически, логика объектов управления переносится в автомат Бюхи, и проблема решается (как упоминалось в разд. 1.3, состояние модели при верификации складывается из состояния модели и состояния автомата Бюхи). Однако сформулировать такое ограничение в терминах темпоральной логики бывает достаточно сложно. Даже для такой простой логики, как «Ресурс», вывод темпоральной формулы нетривиален:

  ```
  o1.x1 W o1.z1 ∩
  G (o1.z2 → (o1.x1 W o1.z1) ∩
     o1.z1 → (¬o1.x1 W o1.z2))
  ```

- Можно не отцеплять объекты управления, влияющие на логику работы автоматной модели, и добавить их состояния в общее состояние модели. Это, пожалуй, самое полное, но в то же время

самое неэффективное решение. Во-первых, число состояний в итоговой модели Крипке может возрасти гораздо больше, чем требуется. Во-вторых, объекты управления могут производить долгие действия или вообще быть внешними устройствами. В таком случае их можно заменить «заглушками» («mock»).

В любом случае решение проблемы логики, хранящейся в объектах управления, требует большого вмешательства пользователя в процесс верификации и хорошее знание верифицируемой системы.

3.3. Пример использования

Разработанный верификатор был апробирован на нескольких примерах и успешно доказывал верные утверждения о модели и опровергал неверные утверждения. Приведем простой и наглядный пример использования разработанного верификатора.

На рис. 7 изображена простая модель работы дверей лифта. Модель состоит из единственного автомата.

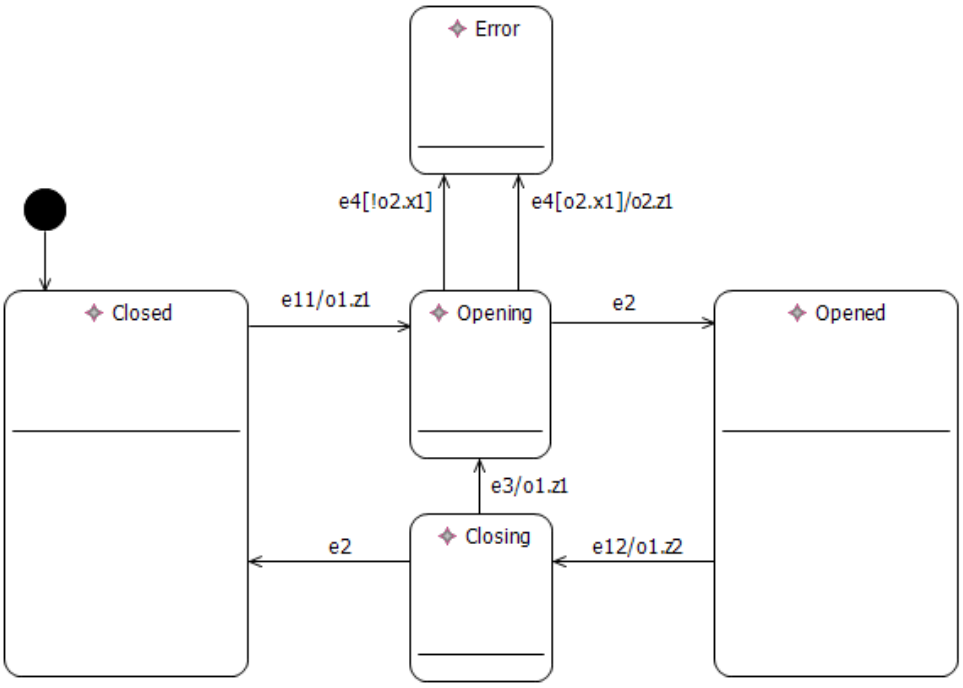

Рис. 7. *Автомат, управляющий дверьми лифта.*

Работа начинается при закрытых дверях в состоянии `Closed` («Закрыты»). При нажатии кнопки «Открыть» (событие `e11`), запускается механизм открытия дверей (`o1.z1`) и автомат переходит в состояние `Opening` («Открываются»). При получении сообщения об успешном завершении открытия или закрытия (`e2`), автомат переходит в состояние `Opened` («Открыты»). Процесс закрытия происходит аналогичным образом, где `e12` – нажатие кнопки «закрыть», `o1.z2` – запуск закрытия дверей. Если какое-либо препятствие мешает дверям закрыться, то происходит событие `e3`, и двери снова открываются, и автомат переходит в состояние `Opening`. Также при открытии дверей возможно возникновение ошибки (событие `e4`), что приводит автомат в состояние `Error` (Ошибка). При этом если включен механизм оповещения (`o2.x1`), то происходит звонок в аварийную службу (`o2.z1`). Состояние `Error` создано для демонстрации возможностей верификатора, и поэтому, для простоты, в него можно попасть только из состояния `Opening`.

Разработанному верификатору подавался на вход *UniMod*-файл, содержащий описанную модель, *BIR*-файл, содержащий служебную информацию, а также название утверждения, сформулированного в *BIR*-файле, которое требуется проверить. Далее перечисляются утверждения и результаты их верификации.

«Автомат никогда не попадает в состояние `Error`». В *BIR*-файле это утверждение формулируется следующим образом:

```
fun NeverError_Failing() returns boolean =
  LTL.temporalProperty(
    Property.createObservableDictionary(
      Property.createObservableKey("is_Error",
                AutomataModel.isInState(model, "/A", "Error"))
    ),

    LTL.always(
      LTL.negation(LTL.prop("is_Error"))
    )
  );
```

что соответствует формуле темпоральной логики «**G¬**(Error)» («A» – название корневого и единственного автомата в данном примере). Результат работы верификатора выглядит следующим образом:

```
D:\>verifier NeverError_Failing

Transitions: 1, States: 1, Matched States: 0, Max Depth: 1, Errors found:
0, Used Memory: 2MB
Transitions: 9, States: 7, Matched States: 3, Max Depth: 6, Errors found:
1, Used Memory: 1MB
Transitions: 10, States: 7, Matched States: 4, Max Depth: 6, Errors found:
2, Used Memory: 1MB
Total memory before search: 702a880 bytes (0,67 Mb)
Total memory after search: 1a109a016 bytes (1,06 Mb)
Total search time: 1156 ms (0:0:1)
States count: 7
Matched states count: 4
Max depth: 6

Generating error trace 0...

Generating error trace 1...

Done!


2 traces were found.
Replaying the trace with least states (#0).

Replaying trace by key: 1
Stack of transitions leading to the error:
Model [ step [0] event [null] guards [null] transitions [null] actions
       [null] states [null] ] fsaState [T0_init]
Model [ step [0] event [] guards [] transitions [] actions [] states [(/A)
       - (Top)] ] fsaState [T0_init]
Model [ step [1] event [*] guards [] transitions [s1#Closed#*#true] actions
       [] states [(/A) - (Closed)] ] fsaState [T0_init]
Model [ step [2] event [e11] guards [true->true] transitions
       [Closed#Opening#e11#true] actions [o1.z1] states [(/A) - Opening)] ]
       fsaState [T0_init]
Model [ step [3] event [e4] guards [o2.x1->true] transitions
       [Opening#Error#e4#o2.x1] actions [o2.z1] states [(/A) - (Error)] ]
       fsaState [bad$accept_all]

Done!
```

Верификатор нашел два сценария нарушения приведенного требования и вывел ошибку для наиболее короткого из них. Поясним формат сценария.

В сценарии каждый шаг имеет следующий формат:

```
step [<номер шага модели>]
event [<полученное событие>]
guards [<вычисленные условия на переходах с их значениями>]
transitions [<осуществленные переходы в автоматах>]
actions [<список вызванных воздействий в объектах управления>]
states [<состояние каждого автомата после обработки события>]
fsaState [<состояние автомата Бюхи после обработки события>]
```

При выводе сценария выводится избыточная информация, и для понимания произведенных переходов достаточно только полученного события и вычисленных условий на переходах. Первые два шага – инициализация модели, и они не несут смысловой нагрузки. С третьей строчки сценария осуществляется получение события «*». Это событие происходит тогда, когда не определено ни одного другого события для исходящих из текущего состояния переходов. Автомат перешел из начального состояния в состояние `Closed`. Далее было получено событие `e11`, и автомат перешел в состояние `Opening`. Затем произошло событие `e4`, условие `o2.x1` было вычислено как `True`, и автомат оказался в состоянии `Error`.

Допускающий путь имеет структуру αβ*, и в сценарий выводится только часть αβ. Выделить цикл β достаточно просто: требуется лишь найти шаг сценария с теми же состояниями автоматов модели и автомата Бюхи, как в последнем шаге. Заметим, что в алгоритме двойного *DFS* получаемый суффикс β всегда начинается с допускающего состояния. Названия допускающих состояний в сгенерированном автомате Бюхи в верификаторе *Bogor* начинаются с «`bad$`». В данном сценарии цикл β состоит из единственного шага в состоянии `Error`.

Сгенерированный автомат Бюхи можно увидеть следующим образом. При нахождении ошибок верификации *Bogor* генерирует zip-архив с расширением `.bogor-trails`, в котором хранится также исходный *BIR*-файл. При этом в *BIR*-файле исходная функция, формулирующая *LTL*-формулу для

верификации (в данном примере это была NeverError_Failing), заменена на автомат Бюхи:

```
function generated$FSA()
    {
        loc T0_init:
            when true do
            {
            }
            goto T0_init;
            when AutomataModel.isInState(model, "/A", "Error") do
            {
            }
            goto bad$accept_all;
        loc bad$accept_all:
            when true do
            {
            }
            goto bad$accept_all;
    }
```

Итак, верифицируемое свойство было нарушено и представлена история, в которой автомат попадает в состояние Error. Изменим свойство, добавив условие, что никогда не происходит события e4:

$$G(\neg \texttt{wasEvent("e4")}) \rightarrow G(\neg \texttt{isInState("/A", "Error")})$$

Результат работы верификатора:

```
D:\>verifier NeverError_Successive
Transitions: 1, States: 1, Matched States: 0, Max Depth: 1, Errors
found: 0, Used Memory: 2MB
Transitions: 9, States: 6, Matched States: 3, Max Depth: 6, Errors
found: 0, Used Memory: 1MB
Total memory before search: 706a320 bytes (0,67 Mb)
Total memory after search: 1a116a888 bytes (1,07 Mb)
Total search time: 625 ms (0:0:0)
States count: 6
Matched states count: 3
Max depth: 6
Done!
Verification successful!
```

Данное свойство выполняется на модели.

Проверим теперь следующее свойство: если в состоянии `Opening` произошло событие `e4`, то следующее состояние будет `Error`.

```
G ( wasInState("/A", "Opening") ∩ wasEvent("e4") →
isInState("/A", "Error") )
```

Результат – свойство выполняется. Данный пример предназначен для демонстрации работы анализатора условий на переходах. До того, как был добавлен анализатор, для данного свойства генерировался контрпример, в котором присутствовал следующий невозможный набор значений условий:

```
guards [o2.x1->false, !o2.x1->false]
```

Теперь проверим следующее свойство: если не происходит ошибки, то для любой истории автомат попадает в состояние `Opened` бесконечно часто.

```
G¬ wasEvent("e4") → G F isInState("/A", "Opened")
```

Результат верификации – условие выполняется. Отметим, что при верификации рассматриваются только *справедливые* [4] истории. Это означает, что любое ожидаемое событие когда-либо произойдет, и автомат не остановится навсегда в состоянии, из которого есть переход по некоторому событию. Применительно к данному примеру, это означает, например, что если дверь стала открываться, то событие `e2` обязательно произойдет. Рассматривать несправедливые истории, пожалуй, не имеет смысла, поскольку тогда для любого свойства найдется контрпример, когда автомат просто не получает никаких событий.

Последнее свойство, которое будет проверено на данном примере – такое же, как предыдущее, только для состояния `Closed`: при условии отсутствия ошибки, автомат бесконечно часто проходит через состояние `Closed`.

$$G\neg\, \texttt{wasEvent("e4")} \rightarrow G\, F\, \texttt{isInState("/A", "Closed")}$$

Результат верификации – свойство нарушается. Выводимый сценарий ошибки:

```
[1] event [*] ... states [(/A) - (Closed)] ] fsaState [T0_init]
[2] event [e11] ... states [(/A) - (Opening)] ] fsaState [bad$accept_S2]
[3] event [e2] ... states [(/A) - (Opened)] ] fsaState [bad$accept_S2]
[4] event [e12] ... states [(/A) - (Closing)] ] fsaState [bad$accept_S2]
[5] event [e3] ... states [(/A) - (Opening)] ] fsaState [bad$accept_S2]
```

Как видно, был получен цикл, состоящий из шагов 2 – 5, в котором автомат не попадает в состояние `Closed`. Полученный контрпример соответствует ситуации, когда при каждой попытке закрыться дверь встречает препятствие (`e3`) и снова открывается. Графически он представлен на рис. 8.

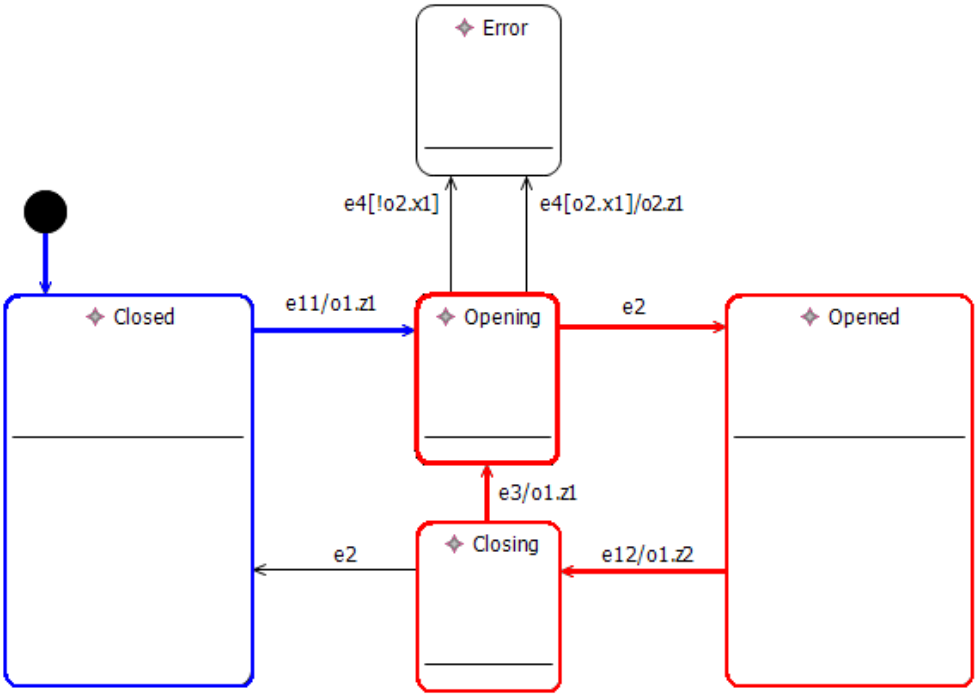

Рис. 8. *Графическое представление контрпримера для утверждения* $G\neg$ `wasEvent("e4")` $\rightarrow G\,F$ `isInState("/A", "Closed")`

Также для сравнения результатов с предыдущими работами, был промоделирован пример из работы [5] – создана модель «Универсального инфракрасного пульта для бытовой техники». Результаты работы [5] и настоящей работы на этом примере совпали.

Выводы по главе 3

Сравнение с другими работами показало, что предлагаемый метод верификации не уступает в функциональности другим методам. В то же время разработанный верификатор позволяет *автоматически* верифицировать те модели, которые вручную верифицировались в других работах. Результаты верификации совпадают.

В этой главе также была сформулирована проблема возможной потери части логики работы системы при отключении объектов управления от автоматов, и были намечены некоторые идеи для решения этой проблемы.

ЗАКЛЮЧЕНИЕ

В данной работе был разработан автоматический верификатор автоматных моделей программ. Он успешно был использован на нескольких примерах, в том числе взятых из других работ, и показал корректные результаты. Верификатор был разработан как дополнение к инструментальному средству *UniMod*, что позволяет получить универсальное средство для создания, запуска и верификации реактивных систем.

Проверим, насколько разработанный верификатор удовлетворяет требованиям, предъявленным к нему во введении к настоящей работе.

- На вход верификатор получает следующие сущности.
  а. Автоматную модель в виде *UniMod*-файлов.
  б. Верифицируемое свойство модели в виде темпоральной *LTL*-формулы, записываемой в *BIR*-файл.
- Время работы верификатора конечно, поскольку суммарное число состояний, в которых может оказаться система в процессе верификации равняется произведению числа состояний автоматов на число состояний сгенерированного автомата Бюхи, что является конечным числом.
- В ходе работы верификатора в консоль выводятся сообщения о прогрессе.
- Работа верификатора завершается, когда он обошел все возможные пути в модели. Можно также задать число ошибок, после нахождения которых верификатор завершит работу. Это число задается в файле настроек верификатора *Bogor* (файл с расширением `.bogor-conf`).
- То, что разработанный верификатор правильно работал на примерах, позволяет надеяться, что он всегда работает корректно.
- На выход верификатор выводит следующее.
  а. В случае если свойство выполняется, выводится сообщение об удачном завершении верификации.

б. Если свойство не выполняется, выводится число обнаруженных сценариев его нарушения. Затем выводится самый короткий из них.

- Выводимый сценарий состоит из списка шагов обработки событий автоматной моделью. Для каждого шага указываются:
  а. номер шага;
  б. обрабатываемое событие;
  в. список переходов, активированных в ходе выполнения шага;
  г. значения условий, вычисленных на переходах;
  д. выполненные в ходе шага действия объектов управления;
  е. состояние каждого автомата модели после выполнения шага;
  ж. состояние автомата Бюхи после выполнения шага.

В дальнейшем планируется создание удобного интерфейса для разработанного верификатора, который позволял бы пользователю прямо на диаграмме переходов автоматов наблюдать сценарий ошибки. Также планируется создать удобный способ задания верифицируемых темпоральных свойств модели.

СПИСОК ЛИТЕРАТУРЫ

1. *Шалыто А. А.* Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/switch/l/
2. *Шалыто А. А., Туккель Н. И.* SWITCH-технология – автоматный подход к созданию программного обеспечения «реактивных» систем // Программирование. 2001. № 5. http://is.ifmo.ru/works/switch/l/
3. Статья «NASA: миссия надежна». http://www.osp.ru/os/2004/03/184060/
4. *Clarke E., Grumberg O., Peled D.*. Model checking. The MIT Press. 2000.
5. *Вельдер С. Э., Шалыто А. А.* Введение в верификацию автоматных программ на основе метода Model checking. http://is.ifmo.ru/download/modelchecking.pdf
6. *Кузьмин Е. В., Соколов В. А.* О верификации «автоматных» программ /Актуальные проблемы математики и информатики. Сборник статей к 20-летию факультета ИВТ ЯрГУ им. П.Г. Демидова. Ярославль. ЯрГУ. 2006, с. 27 – 32. http://is.ifmo.ru/verification/_verautpr.pdf
7. *Белешко Д. С.* Верификация автоматных моделей программ. Бакалаврская работа. Санкт-Петербургский государственный университет информационных технологий, механики и оптики, 2006.
8. *Holzman G. J.* The Model Checker Spin. http://spinroot.com/spin/Doc/ieee97.pdf
9. *Robby, Dwyer M., Hatcliff J.* Bogor: A Flexible Framework for Creating Software Model Checkers /TAIC PART 2006: pp. 3 – 22. http://projects.cis.ksu.edu/docman/view.php/8/125/SAnToS-TR2006-2.pdf
10. *Robby, Dwyer M., Hatcliff J.* Bogor: An Extensible and Highly-Modular Model Checking Framework /In the Proceedings of the Fourth Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003). http://projects.cis.ksu.edu/docman/view.php/8/19/SAnToS-TR2003-3.pdf
11. *Deng W., Dwyer M., Hatcliff J., Jung G., Robby, Singh G.* Model-checking Middleware-based Event-driven Real-time Embedded Software /In the Proceedings

of the First International Symposium on Formal Methods for Components and Objects (FMCO 2002). http://projects.cis.ksu.edu/docman/view.php/8/20/SAnToS-TR2003-2.pdf

12. *Гуров В. С., Мазин М. А., Шалыто А. А.* UniMod – Инструментальное средство для автоматного программирования //Научно-технический вестник СПбГУ ИТМО. Вып. 30. Фундаментальные и прикладные исследования информационных систем и технологий». 2006, с. 32 – 44. http://is.ifmo.ru/works/_instrsr.pdf

13. *Гуров В. С., Мазин М. А., Нарвский А. С., Шалыто А. А.* UML. SWITCH-технология. Eclipse //Информационно-управляющие системы. 2004. № 6. http://is.ifmo.ru/works/uml-switch-eclipse/