УДК 681.3.06

О ВЕРИФИКАЦИИ ПРОСТЫХ АВТОМАТНЫХ ПРОГРАММ НА ОСНОВЕ МЕТОДА *MODEL CHECKING*

С. Э. Вельдер,
магистрант
А. А. Шалыто,
доктор техн. наук, профессор
Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Излагается применительно к простым автоматным программам (их поведение описывается одним конечным автоматом) техника верификации, которая базируется на темпоральных логиках и называется Model Checking. Для автоматных программ удается автоматизировать процесс построения модели программы, подлежащей верификации.

Verification of simple automata-based programs (whose behavior can be described with a singe finite automaton) is considered. The applied verification technique is based on temporal logics and is known as Model Checking. For automata-based programs it is possible to automate the process of building program model subject to verification.

Введение

Model Checking — это автоматизированный подход, позволяющий для заданной модели поведения системы с конечным (возможно, очень большим) числом состояний и логического свойства (требования) проверить, выполняется ли это свойство в рассматриваемых состояниях данной модели.

Алгоритмы для *Model Checking* обычно базируются на полном переборе пространства состояний модели. При этом для каждого состояния проверяется, удовлетворяет ли оно сформулированным требованиям. Алгоритмы гарантированно завершаются, так как модель программы конечна. Принципиальная схема *Model Checking* приведена на рис. 1.

В проблематике верификации [1] сформировалось два направления: аксиоматическое и алгоритмическое. При использовании первого из них разрабатывается набор аксиом, с помощью которого может быть описана как сама система, так и ее свойства [2]. Основу второго направления составляет метод *Model Checking*.

Перечислим достоинства этого метода.

1. Эффективность. Программы для верификации моделей способны работать с большими пространствами состояний благодаря концепции упорядоченных двоичных разрешающих деревьев [3], которая также упоминается в данной работе.

2. Возможность генерации контрпримеров.

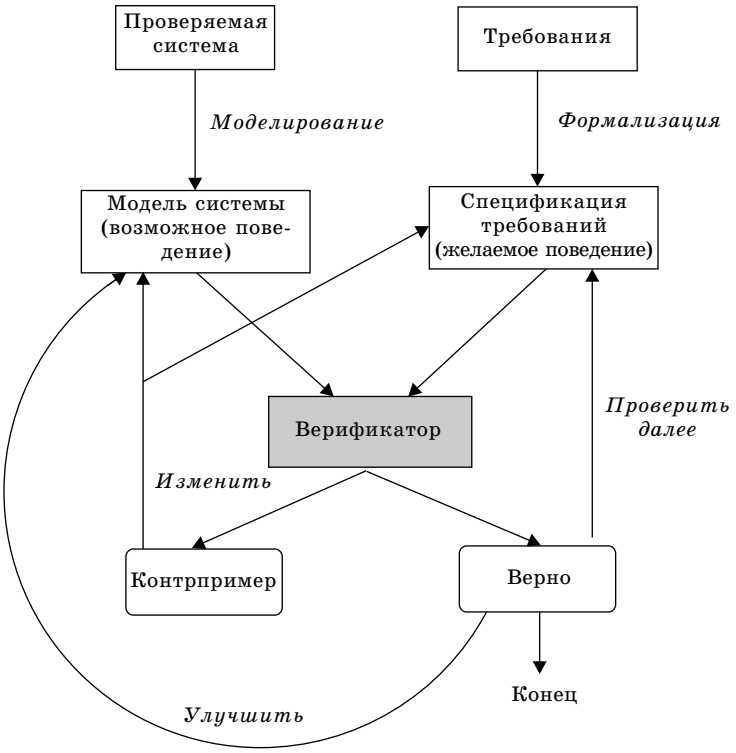
Перечислим ограничения рассматриваемого метода.

1. Поддержка только моделей с конечным числом состояний. Поэтому для большинства классов систем с бесконечным числом состояний необходимо выполнять формальную верификацию системы — математическое доказательство свойств самой программы, а не ее модели.

2. Ограниченность верификации. С использованием метода *Model Checking* проверяется модель системы вместо реальной системы. Таким образом, любое применение метода *Model Checking* настолько же качественно, как и сама модель системы. Кроме того, с помощью этого метода проверяются не любые свойства модели, а только темпоральные.

3. Для многопроцессорных систем размер пространства состояний в худшем случае пропорционален произведению размеров пространств состояний их индивидуальных компонент. Этот эффект называется проблемой показательного (экспоненциального) взрыва состояний (state-space explosion problem).

В рамках данной работы рассматривается автоматное программирование (программирование с явным выделением управляющих состояний) [4, 5], поэтому ограничения 1 и 3 в нашем случае несущественны.



■ ***Рис. 1.*** *Model Checking*

Отметим, что настоящая работа выполнялась «параллельно» с работой [6], которая появилась после знакомства ее авторов с автоматным подходом.

Технология верификации простых автоматных программ

Технология автоматного программирования использует такие модели [7], как автомат Мили [8], автомат Мура [9] и смешанный автомат, которые легко интерпретируются с помощью модели Крипке. Автоматную программу будем называть простой, если ее поведение описывается одним конечным автоматом.

Первым шагом в процессе верификации автоматной программы является преобразование графа переходов исходного автомата в модель Крипке, для которой удобно формулировать проверяемые свойства программ. В данной работе отдается предпочтение автомату Мили, который при необходимости всегда может быть преобразован в автомат Мура.

Исследования в данной области (моделирование автомата и конвертация его в модель Крипке) проводились в работах [10–13]. При этом конвертация была сопряжена со следующими проблемами:

• трудности с выполнением композиции автоматов;

• неоднозначность интерпретации формулы языка *Computational Tree Logic* (*CTL*) [14] в исходном автомате.

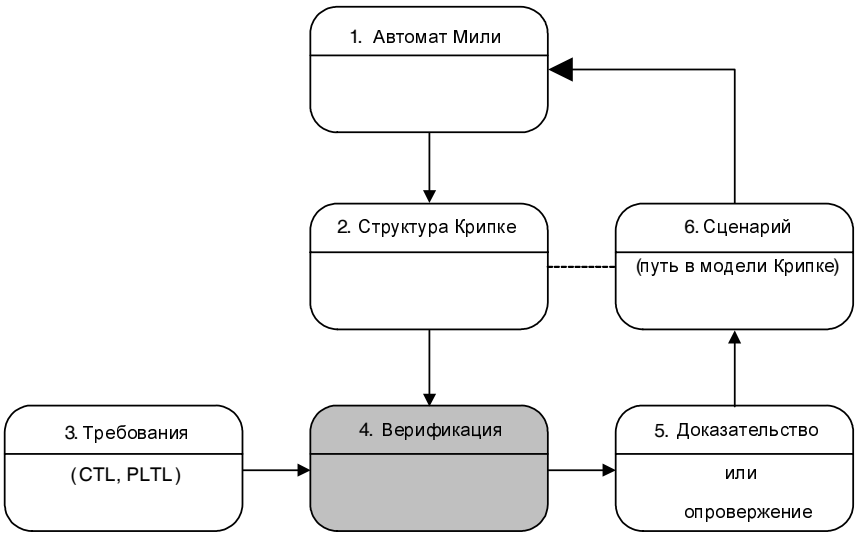
При решении первой проблемы, как правило, возникала вторая. Для ее решения применялась модификация языка *CTL*.

Методы моделирования, рассматриваемые в настоящей работе (в частности, «редуцированная» схема), проводят изменение семантики языка *CTL* для того, чтобы воспрепятствовать экспоненциальному росту числа состояний. При этом пути, построенные в качестве сценариев для *CTL*-формул, однозначно преобразуются из модели в автомат Мили. Это удобно, особенно если моделирование производится совместно с исполнением автомата, его визуализацией и отладкой [15, 16].

Кроме того, при использовании автоматного программирования число управляющих состояний относительно невелико. Это позволяет не применять в данной работе специальные техники для сжатия автоматов с большим числом состояний (упорядоченные двоичные разрешающие диаграммы), а использовать достаточно простые и наглядные алгоритмы, которые работают быстро при небольшом числе состояний, один из которых будет дополнен алгоритмом генерации сценариев.

Перечислим основные положения данной работы.

1. В автоматных программах [4, 5] поведение специфицируется с помощью конечных автоматов. В настоящей работе применяются спецификации, состоящие из одного автомата Мили. В общем случае модель может состоять из нескольких взаимодействующих автоматов. Для верификации таких



■ ***Рис. 2.*** *Этапы верификации автоматных программ*

систем применяется композиция исходных автоматов или моделей Крипке.

2. Использование подхода *Model Checking* для таких программ связано с преобразованием автомата Мили в структуру Крипке (модель Крипке), так как она, в отличие от автомата, приспособлена для верификации.

3. Использование структуры Крипке предполагает применение темпоральной логики для записи требований, которые должны быть проверены. В настоящей работе при написании программ верификации требования описываются на языке *CTL*.

4. Собственно верификация модели (см. рис. 1) выполняется по структуре Крипке, построенной по автомату Мили, и требованиям, записанным в виде формулы на языке темпоральной логики *CTL*.

Верификация осуществляется с использованием двух алгоритмов. Первый предназначен для определения набора состояний в структуре Крипке, в которых выполняется заданное формулой требование, а второй по заданному исходному состоянию и подформуле заданного требования с помощью построенного набора состояний формирует сценарий, который подтверждает или опровергает эту подформулу.

5. Сценарий для структуры Крипке преобразуется в сценарий для автомата Мили.

6. Все этапы изложенной технологии верификации рассматриваемого класса автоматных программ (рис. 2) иллюстрируются на примере программы для «Универсального инфракрасного пульта для бытовой техники» [17].

Преобразование автомата Мили в структуру Крипке и разработка требований

Построим несколько различных схем для генерации множества атомарных предложений автоматной программы, преобразования автомата Мили в модель Крипке и записи требований к программе.

Выделим и опишем три основные схемы такого преобразования:

1) установка состояний на событиях и выходных воздействиях (переменных);

2) создание полного графа переходов;

3) редукция полного графа переходов с внесением тесных отрицаний (термин поясняется ниже) внутрь атомарной формулы.

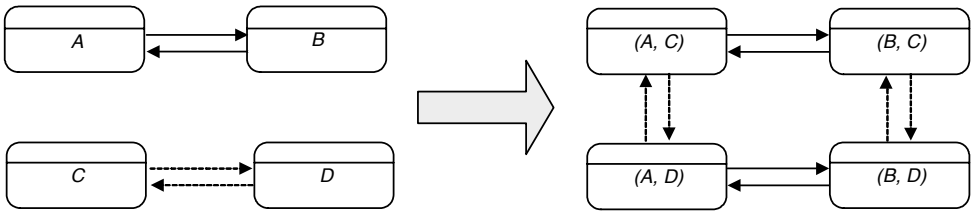
Учтем для любой схемы, что если конечная формула ее спецификации представима в виде конъюнкции нескольких подформул, то эту конъюнкцию целесообразно разбивать на операнды и рассматривать их по отдельности, так как при этом удобнее (и правильнее) исследовать, адекватны ли формальные требования к модели соображениям разработчика о них.

Автоматы, в которых состояния могут содержать внутри себя другие автоматы, можно исследовать тремя способами.

1. Для внешних и внутренних автоматов можно выполнять моделирование, спецификацию и верификацию независимо (конечно, этот способ влечет утрату определенных характеристик автомата при моделировании).

2. Можно «раскрыть» состояние S автомата A, внутри которого (состояния) находится другой автомат B, добавив для каждого перехода из состояния S в состояние T по одному эквивалентному переходу из каждого состояния автомата B в состояние T. Все переходы, которые ведут в состояние S, следует перенаправить в стартовое состояние автомата B. В результате, внутренний автомат B превращается в часть автомата A, и для него можно выполнять верификацию вместе с автоматом A.

3. Систему взаимодействующих автоматов можно привести к одному автомату с помощью



■ ***Рис. 3.*** *Композиция структур Крипке*

композиции (произведения) [10]. Также можно выполнять сначала моделирование каждого автомата, а после него — композицию моделей Крипке (рис. 3).

Выбор способа определяется соображениями эффективности и удобства. В примерах, описываемых далее, считается, что данный вопрос уже решен, и рассматриваются системы, описываемые одним автоматом без вложенных состояний.

Во всех трех схемах, которые будут построены, состояния исходного автомата изоморфно перейдут в состояния модели.

Для каждого перехода между состояниями S и T исходного автомата создадим не менее одного состояния в модели Крипке (назовем его *состоянием-событием*), атомарным предложением которого будет событие E, инициировавшее переход. При наличии выходных воздействий на переходе также создадим по одному состоянию на каждое воздействие Z, атомарным предложением которого (состояния) будет Z (такие состояния будем называть *состояниями-выходными воздействиями*). Добавим в модель переходы: между состоянием S и состоянием-событием E; между состоянием-событием E и первым состоянием-выходным воздействием; далее последовательно (в порядке выполнения) между соседними состояниями для выходных воздействий и, наконец, между последним таким состоянием-выходным воздействием и состоянием T (далее будет приведен пример такой конвертации).

Если выходное воздействие Z размещалось в состоянии T и выполнялось при входе в него, то при конвертации добавляется еще одно состояние, соответствующее воздействию Z. В это состояние должен вести каждый переход, который первоначально вел в состояние T. Кроме этого, добавляется переход из состояния, соответствующего Z, в состояние T. Само же это воздействие после генерации состояний уничтожается.

Для всех полученных состояний модели Крипке естественным образом устанавливаются атомарные предложения. Добавим также три «управляющих» атомарных предложения: *InState*, *InEvent*, *InAction* — для состояний модели, построенных соответственно из *состояний*, *событий*, *выходных воздействий* исходного автомата. Это сделано для того, чтобы при записи формулы в темпоральной логике можно было различать тип исследуемого состояния.

Таким образом, множество атомарных предложений во всех трех схемах содержит объединение множеств состояний, событий, выходных воздействий и трех описанных выше атомарных предложений. Далее будут рассмотрены индивидуальные особенности каждой из трех схем.

Схема «Состояния на событиях и выходных воздействиях» (ССВВ), как и две другие, наследует общую идеологию моделирования, описанную выше. От других схем ее отличает то, что кроме указанного общего принципа в ней больше ничего не содержится. Таким образом, применяя схему ССВВ для автоматной программы, можно полностью абстрагироваться от понятия входных переменных, оставляя только состояния (без них не обойдется ни одна базовая модель), события и выходные воздействия. Это самый простой подход.

Рассмотрим ***пример***. Пусть исходное автоматное приложение эмулирует (в довольно упрощенной форме) универсальный инфракрасный пульт для бытовой техники [17]. Эмулятор представлен с помощью одного автомата *ARemote* (рис. 4). Граф переходов этого автомата приведен на рис. 5.

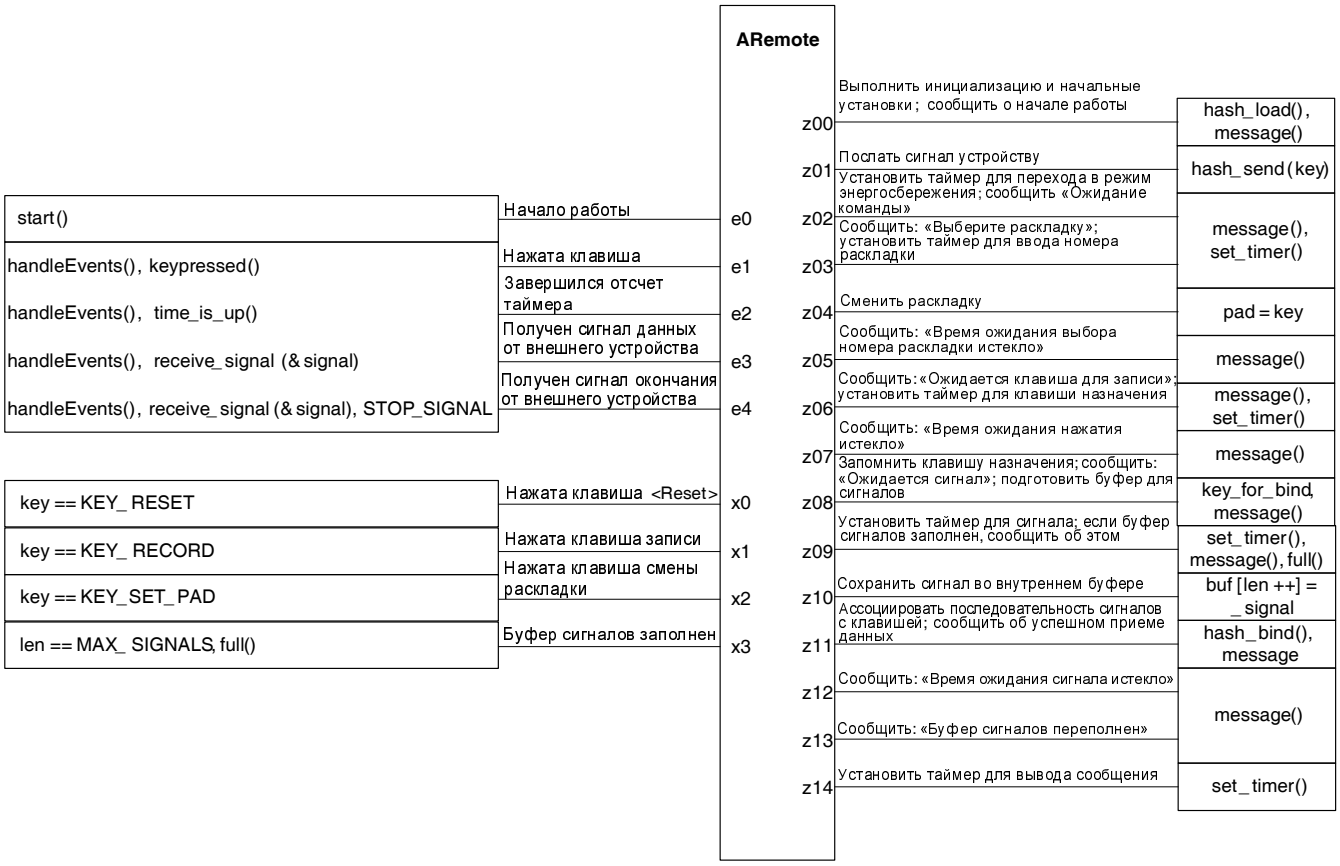
В рассматриваемом примере модель Крипке для автомата, построенная по схеме ССВВ, будет изоморфна графу на рис. 5.

В модели Крипке, изображенной на рис. 6, состояния-события и состояния-выходные воздействия указаны явно.

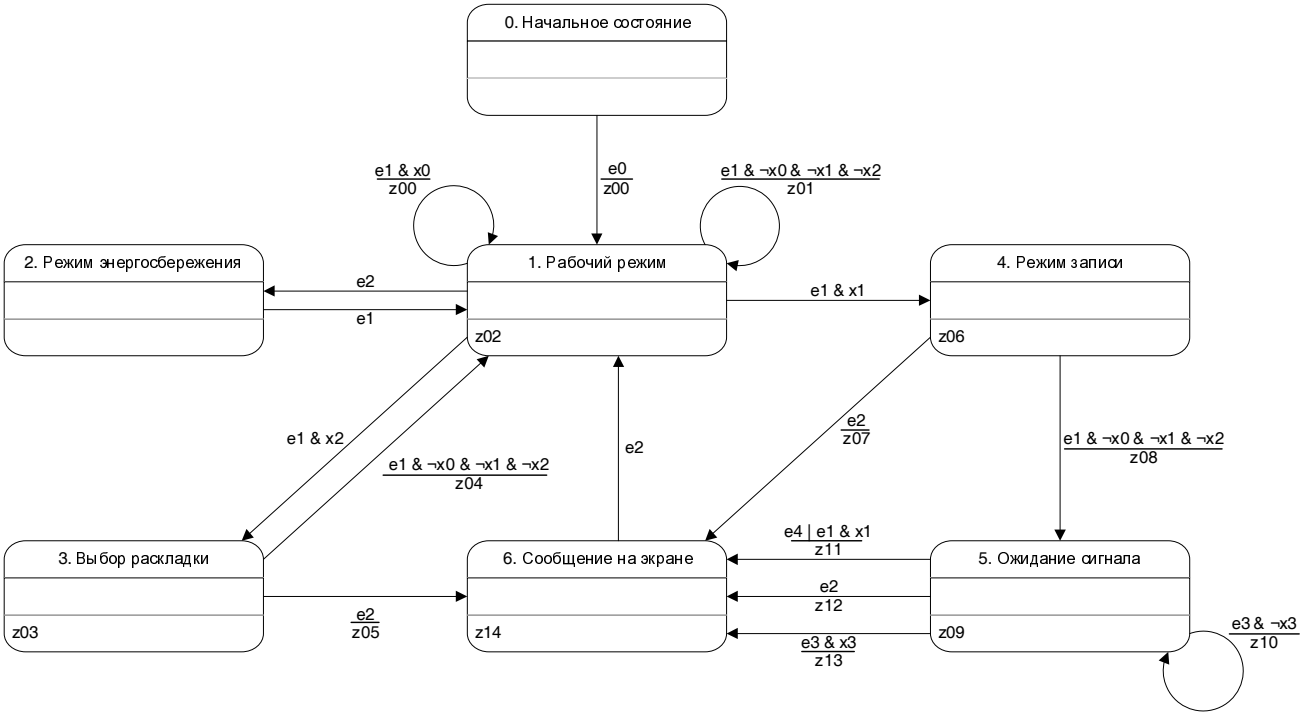
При интерактивном моделировании совместно с исполнением и визуализацией [14, 15] их целесообразно обозначать, как и в исходном автомате, в виде меток на дугах. Таким образом, модель построена.

Приведем теперь ***пример*** *CTL*-формулы, справедливость которой можно устанавливать верификацией: $\neg\mathbf{E}[\neg(Y=6)\mathbf{U}(Y=1)]$. Смысл этой формулы состоит в следующем: в состояние *1* нельзя попасть, минуя состояние *6* (нельзя попасть в рабочий режим, минуя сообщение на экране). Эта формула справедлива для состояний *4*, *5*, *6* исходного автомата и только для них (для модели Крипке таких состояний больше).

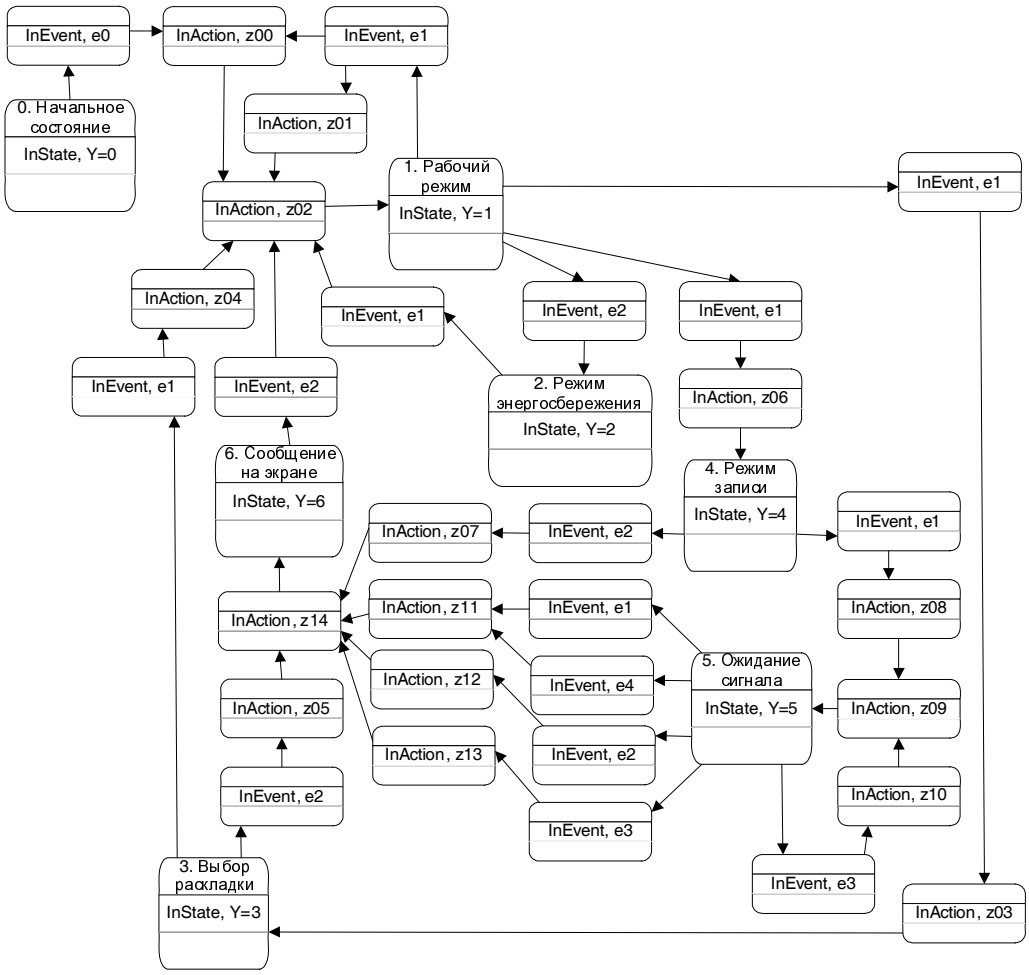
Схема «Полный автомат». Во второй схеме не будем абстрагироваться от входных переменных, а представим автомат моделью Крипке «со всей полнотой» относительно входных воздействий. В исходном автомате переходы могут быть заданы не полностью — могут существовать не указанные петли. Это означает, что для некоторого состояния (некоторых состояний) дизъюнкция формул,



■ ***Рис. 4.*** *Схема связей автомата ARemote*



■ ***Рис. 5.*** *Граф переходов автомата ARemote*



■ ***Рис. 6.*** *Модель Крипке, построенная по схеме CCBB*

составленных из входных переменных, которые помечают переходы из него по одному и тому же событию E, не является тавтологией.

Снабдим это состояние (состояния) петлевыми переходами по событию E, соответствующими дополнению к рассматриваемой дизъюнкции. Это, конечно же, не изменит семантику автомата, а лишь полностью опишет его поведение. В конечном счете в автомате из каждого состояния по каждому событию должно исходить 2^n переходов, где n — общее число входных переменных автомата. При этом каждому переходу соответствует набор значений всех переменных. После получения полного автомата преобразуем его в модель Крипке по общей схеме с одной модификацией: для каждого состояния-события добавим во множество его атомарных предложений набор входных переменных, истинных на том переходе, на котором находится рассматриваемое состояние-событие. Таким образом, во множество атомарных предложений по отношению к обобщенной схеме добавились еще и входные переменные. Достоинство такой схемы (несмотря на ее расточительность, освобождение от которой будет описано ниже) в том, что она и только она позволяет модели Крипке полностью отражать поведение исходного автомата.

«Редуцированная» схема. Основным недостатком предыдущей схемы было большое число генерируемых состояний для модели Крипке, а достоинством – ее полнота.

В «редуцированной» схеме семантика моделей будет изменена таким образом, чтобы число состояний в них можно было уменьшить, не потеряв при этом их выразительные возможности. Это можно сделать так, что размер модели изменится асимптотически *линейно* по отношению к размеру графа переходов исходного автомата и к числу переменных (*билинейно*), в отличие от предыдущей схемы, где размер модели увеличивался *экспоненциально* от числа входных переменных.

Множество атомарных предложений по отношению к предыдущей схеме также будет видоизменено.

Рассмотрим исходный автомат без дизъюнкций на переходах. Если такие переходы существуют, создадим эквивалентные переходы для каждого

дизъюнкта, а сами переходы с дизъюнкциями удалим. В качестве ***примера*** можно разбить переход "($e4 \mid e1\&x1$) / $z11$" графа *ARemote* (см. рис. 5) на два перехода: "$e4$ / $z11$" и "$e1\&x1$ / $z11$".

Добавим в автомат состояния, соответствующие событиям, входным и выходным переменным, так, как это было сделано в первой схеме (ССВВ), но с одним отличием: во множества атомарных предложений на состояниях-событиях добавим входные переменные в том виде, в котором они присутствуют на переходах (вместе с отрицаниями, если они есть). Таким образом, в состав множества всех атомарных формул модели входят следующие элементы, и только они: состояния; события; выходные воздействия; все литералы, составленные из входных переменных (сами переменные и их отрицания). Кроме того, в атомарные предложения каждого полученного состояния-события добавим все литералы, составленные из несущественных входных переменных для данного перехода (несущественными будем называть те переменные исходного автомата, которые не обозначены на рассматриваемом переходе). Таким образом, будем добавлять на одно и то же состояние-событие и несущественные переменные, и их отрицания. С точки зрения синтаксиса и семантики темпоральной логики, это допустимо: процесс обработки модели Крипке не предполагает совместность множества атомарных предложений состояния, так как интерпретирует эти предложения просто как строки. Причина такого обращения с несущественными переменными ясна: требуется обеспечить, чтобы любая ссылка на несущественную в данном состоянии-событии переменную, упомянутая в *CTL*-формуле, давала истинный результат.

Не обязательно хранить все литералы, составленные из несущественных переменных в состоянии в явном виде. Важно лишь то, что во время обработки модели существенные и несущественные входные переменные интерпретируются раздельно: первые – в том виде, в каком они записаны на переходах исходного автомата, а вторые – «в двух экземплярах» (в прямом и инверсном виде).

Результат конвертации графа переходов автомата ARemote, выполненного с применением данной схемы, изображен на рис. 7.

Размер модели на рис. 7 совпадает с размером модели, созданной по схеме ССВВ. Первая схема может рассматриваться по аналогии с третьей или второй, в которой полностью исключены входные воздействия. Аналогично, третью схему можно рассматривать как видоизменение второй, при котором отождествляются наборы значений несущественных переменных.

Теперь рассмотрим построение и интерпретацию *CTL*-формул для «редуцированных» моделей.

CTL-семантика в данной схеме будет немного отличаться от общепринятой: все отрицания, стоящие непосредственно перед атомарными предложениями в *CTL*-формуле (их также называют *местными отрицаниями*), следует внести внутрь атомарных предложений. При этом только результирующая формула в рассматриваемой схеме подлежит верификации методами, предназначенными для *CTL*-логики.

Рассмотрим ***пример*** для автомата *ARemote*. Пусть требуется проверить свойство: «существует способ провести инициализацию устройства, не нажимая кнопку *Reset*». В терминах языка *CTL* с исходной семантикой данное свойство может быть записано следующим образом: $\mathbf{E[}(InEvent \rightarrow \neg\, x0)\ \mathbf{U}\ z00\mathbf{]}$.

Эта формула не выполняется в состоянии $Y = 0$ (см. рис. 7). На это, правда, и не стоило рассчитывать. Преобразуем формулу согласно третьей схеме: $\mathbf{E[}(InEvent \rightarrow !x0)\ \mathbf{U}\ z00\mathbf{]}$. Вместо отрицания в языке *CTL* в формулу было внесено другое атомарное предложение, являющееся отрицанием исходного. Преобразованная формула уже верна для состояния $Y = 0$.

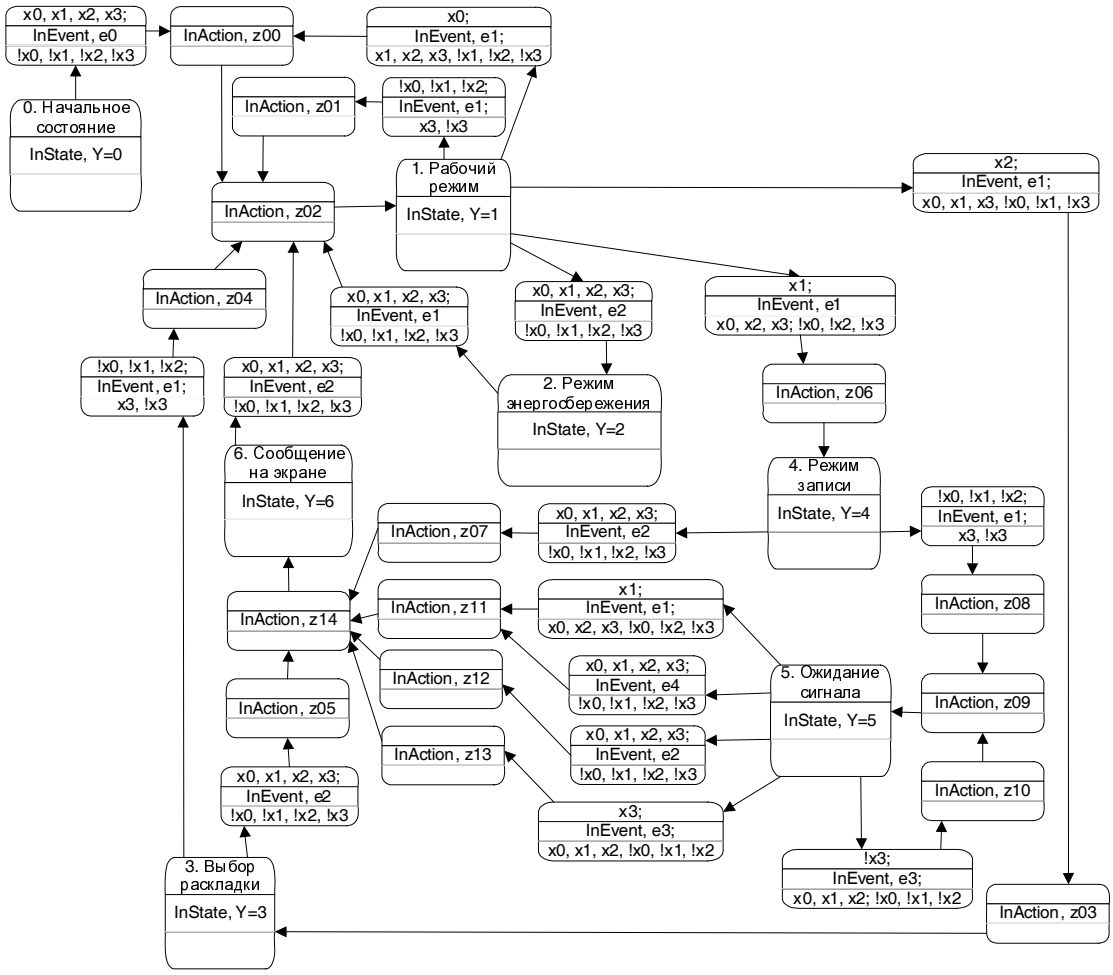
Подведем итог. Для уменьшения числа состояний и из соображений практичности была предложена схема моделирования автомата и изменена семантика языка *CTL*. Однако такое изменение семантики неудобно для верификации. В результате был предложен способ преобразования исходной формулы, соответствующей новой семантике, в новую формулу, для которой применима общепринятая семантика языка *CTL*.

Выполненные примеры показывают, что такой подход не существенно снижает выразительность модели по сравнению с предыдущим (схема «Полный автомат»). Опыт показывает, что для многих формул такая схема может быть использована.

Другие абстракции. Основным недостатком всех описанных выше схем моделирования автоматов является то, что при составлении требований к модели разработчику не всегда удобно различать, где состояния, которые перенесены из исходного графа, где состояния-события, а где состояния-выходные воздействия. Для различения состояний используются атомарные предложения *InState*, *InEvent* и *InAction*, но их применение может быть связано с дополнительными проверками. Для этого, а также для уменьшения числа состояний модели в принципе, можно при построении модели абстрагироваться от каких-либо других ее характеристик, помимо тех, которые были рассмотрены в описанных выше схемах.

Например, можно абстрагироваться не только от входных переменных, но и от событий, а также от выходных воздействий. Можно вообще преобразовать автомат в модель Крипке в один этап, например, с помощью исключения событий и выходных переменных на переходах. Для автомата *ARemote* результатом такого преобразования является модель на рис. 8.

Выбор альтернативного метода можно осуществлять, руководствуясь представлениями о производительности и результативности. Основное внимание необходимо уделять атомарности пере-



■ ***Рис. 7.*** *Редуцированная модель Крипке для автомата ARemote*

ходов. Если они слишком большие (по числу действий), то разработчик может пропустить ошибку, если же слишком маленькие — то размер модели может немотивированно увеличиться за счет появления несущественных свойств.

CTL-верификация автоматных программ

Опишем идею алгоритма *CES* (*Clarke*, *Emerson*, *Sistla*) [18], который основан на переформулировке синтаксиса языка *CTL*. Этот алгоритм дополнен таким образом, что позволяет строить подтверждающие сценарии для проверяемых формул. Применять этот алгоритм будем для изображенных явно моделей Крипке.

Под локальной задачей верификации обычно понимается вопрос: выяснить для данной модели и состояния в ней, выполняется ли в этом состоянии заданная формула.

При построении алгоритма формулируется глобальная задача верификации: для данной модели и проверяемой формулы построить множество всех состояний модели, в которых верна эта формула.

Когда число состояний невелико, как в случае автоматных программ, это множество можно строить в явном виде.

Запишем одну из форм определения синтаксиса и семантики языка *CTL* (в ней темпоральная часть будет целиком выражена через операции **EX**, **EU**, **EG**):

$$\mathbf{A}(f\ \mathbf{U}\ g) = \neg(\mathbf{E}[(\neg g)\ \mathbf{U} \neg(f \vee g)] \vee \mathbf{EG} \neg g)$$

$$\phi ::= p \in AP \mid \neg\phi \mid \phi \vee \phi \mid \mathbf{EX}\phi \mid \mathbf{E}[\phi \mathbf{U} \phi] \mid \mathbf{EG}\phi,$$

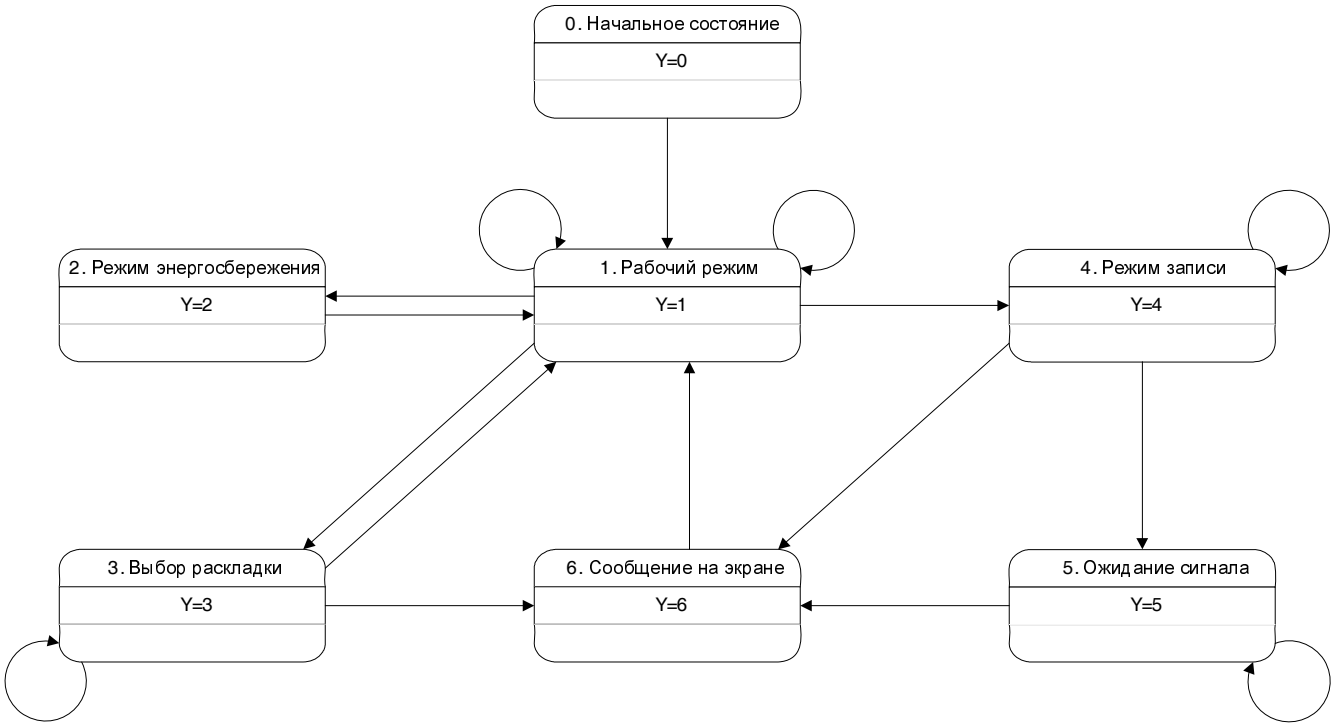
где AP — множество атомарных предложений.

CTL-моделью для множества состояний S называется тройка:

$$\boldsymbol{M} = (S;\ R \subseteq S \times S;\ Label \subseteq S \times AP),$$

здесь R — тотальное отношение на множестве S (отношение переходов между состояниями), а *Label* — отношение, определяющее атомарные предложения, соответствующие каждому состоянию.

Множество выполняющих состояний алгоритм строит для каждой подформулы входной формулы (для каждого состояния создается список вы-



■ ***Рис. 8.*** *Сокращенная модель ARemote (без событий и выходных переменных)*

полненных в нем подформул). Идея алгоритма отражена в псевдокоде (рис. 9).

Как следует из рассмотрения текста этой программы, множество состояний, выполняющих формулу ϕ, строится индукцией по построению ϕ.

Будем считать для удобства, что из исходного графа Крипке построен симметричный ему граф, в котором все переходы заменены на противоположные. В алгоритме *CES* нетривиальными являются последние два шага, которые могут быть реализованы с помощью построения деревьев «обратных путей» и определением компонент сильной связной связности у графа модели.

Теперь осталось только дополнить этот алгоритм методами *предоставления подтверждений* истинности формул в моделях. Иными словами, требуется построить способ генерации сценариев.

Алгоритм генерации сценария для *CTL*-формулы *f* в модели Крипке. Итак, требуется показать, что в данном состоянии *s* модели ***M*** выполняется (или не выполняется) формула *f*.

1. Если *f* — атомарное предложение, то предъявим описание состояния *s* в модели ***M*** — множество его атомарных предложений. В нем, в частности, содержится информация о выполняемости формулы *f* в данном состоянии *s*.

2. Доказательство $\neg f$ сводится к опровержению формулы *f* и наоборот.

3. Для доказательства формулы $f \vee g$ докажем одну из формул *f* или *g*, а для опровержения — опровергаем обе формулы *f* и *g*.

4. Для доказательства **EX** *f* предъявим вершину в модели Крипке, в которую из вершины *s* имеется переход и которая выполняет *f*. Такая вершина обязательно существует, иначе на этапе верификации не обнаружилось бы, что формула **EX** *f* верна. Опровержение **EX** *f* (доказательство **AX**$\neg f$) подтверждается весьма просто, так как *любой* переход, который ведет из вершины *s*, будет вести только в вершину, выполняющую $\neg f$. Таким образом, любой переход из этой вершины можно предъявить пользователю в качестве опровержения.

5. Доказательство формул **E**[*f* **U** *g*] и **EG** *f* выполняется рекуррентным способом с использова-

function $Sat(\phi: Formula)$: **set of** $State$;
begin

if	$\phi = 1$	$\rightarrow$ **return**	S
	$\phi = 0$	$\rightarrow$ **return**	$\varnothing$
	$\phi \in AP$	$\rightarrow$ **return**	$\{s \mid Label(s,\phi)\}$
	$\phi = \neg\phi_1$	$\rightarrow$ **return**	$S \setminus Sat(\phi_1)$
	$\phi = \phi_1 \vee \phi_2$	$\rightarrow$ **return**	$Sat(\phi_1) \cup Sat(\phi_2)$
	$\phi = \mathbf{EX}\phi_1$	$\rightarrow$ **return**	$\{s \in S \mid \exists (s,s') \in R \mid s' \in Sat(\phi_1)\}$
	$\phi = \mathbf{E}[\phi_1 \mathbf{U} \phi_2]$	$\rightarrow$ **return**	$Sat_{EU}(\phi_1, \phi_2)$
	$\phi = \mathbf{EG}\phi_1$	$\rightarrow$ **return**	$Sat_{EG}(\phi_1)$

end if
end

■ ***Рис. 9.*** *Индукция по построению формулы в алгоритме CES*

нием пп. 1–4. Выполним рекуррентное разложение для этих формул:

$$\mathbf{E}[f\ \mathbf{U}\ g] = g \vee f \wedge \mathbf{EX}\ \mathbf{E}[f\ \mathbf{U}\ g]$$

$$\mathbf{EG}\ f = f \wedge \mathbf{EX}\ \mathbf{EG}\ f.$$

Тогда для доказательства формулы $\mathbf{E}[f\ \mathbf{U}\ g]$ достаточно построить путь в графе, применяя шаг за шагом пп. 1–4 к рекуррентному разложению этой формулы до тех пор, пока не попадем в вершину, выполняющую g.

Для доказательства формулы $\mathbf{EG}\ f$ сделаем то же самое, пока не попадем в вершину, в которой уже были. Путь в этом случае, начиная с некоторого состояния, становится периодическим (рис. 10).

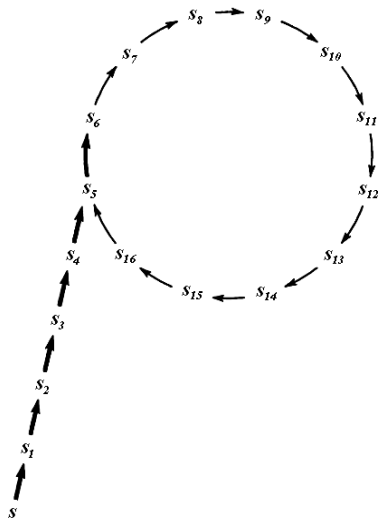
Опровержение формул $\mathbf{E}[f\ \mathbf{U}\ g]$ и $\mathbf{EG}\ f$ выполняется аналогично опровержению формулы $\mathbf{EX}\ f$. Любой (бесконечный) путь, который начинается в текущей вершине, можно предъявить пользователю для рассмотрения, так как путь не выполняет введенную формулу. Иначе говоря, вместо доказательства более выразительных CTL^*-формул $\mathbf{A}\neg[f\ \mathbf{U}\ g]$ и $\mathbf{A}\neg\mathbf{G}\ f$ следует доказывать формулы $\mathbf{E}\neg[f\ \mathbf{U}\ g]$ и $\mathbf{E}\neg\mathbf{G}\ f$. Проще всего предъявлять пути, замыкающиеся, начиная с некоторого состояния, в цикл, так как такие пути однозначно задаются конечным числом вершин.

На этом изложение алгоритма завершено.

Анализ построенного алгоритма формирования сценариев, а также семантики языка *CTL* позволяет сформулировать следующее утверждение.

Утверждение. Если в модели Крипке существует бесконечный путь, выполняющий заданную *CTL*-формулу или являющийся контрпримером к ней, то существует и путь «в ρ-форме» (аналогично, выполняющий или опровергающий ее), представимый в виде объединения «предциклической» и «циклической» частей (см. рис. 10).

Доказательство этого утверждения является конструктивным и целиком опирается на применение описанного алгоритма. Достаточно только заметить, что алгоритм всегда завершается, выдавая сценарий, который должен обладать свойством периодичности.



■ ***Рис. 10.*** *Бесконечный ρ-путь*

Преобразование сценария для модели Крипке в сценарий для автомата Мили

Переходим к последней фазе процесса верификации в автоматном программировании (см. рис. 2). Ниже будет описано, как представлять путь (последовательность вершин) модели Крипке в виде пути исходного автомата Мили. При этом будем предполагать, что модель Крипке была сгенерирована по «редуцированной» схеме. Остальные схемы, для которых было дано описание, позволяют применять к себе аналогичный интуитивно ясный способ перехода от модели к автомату.

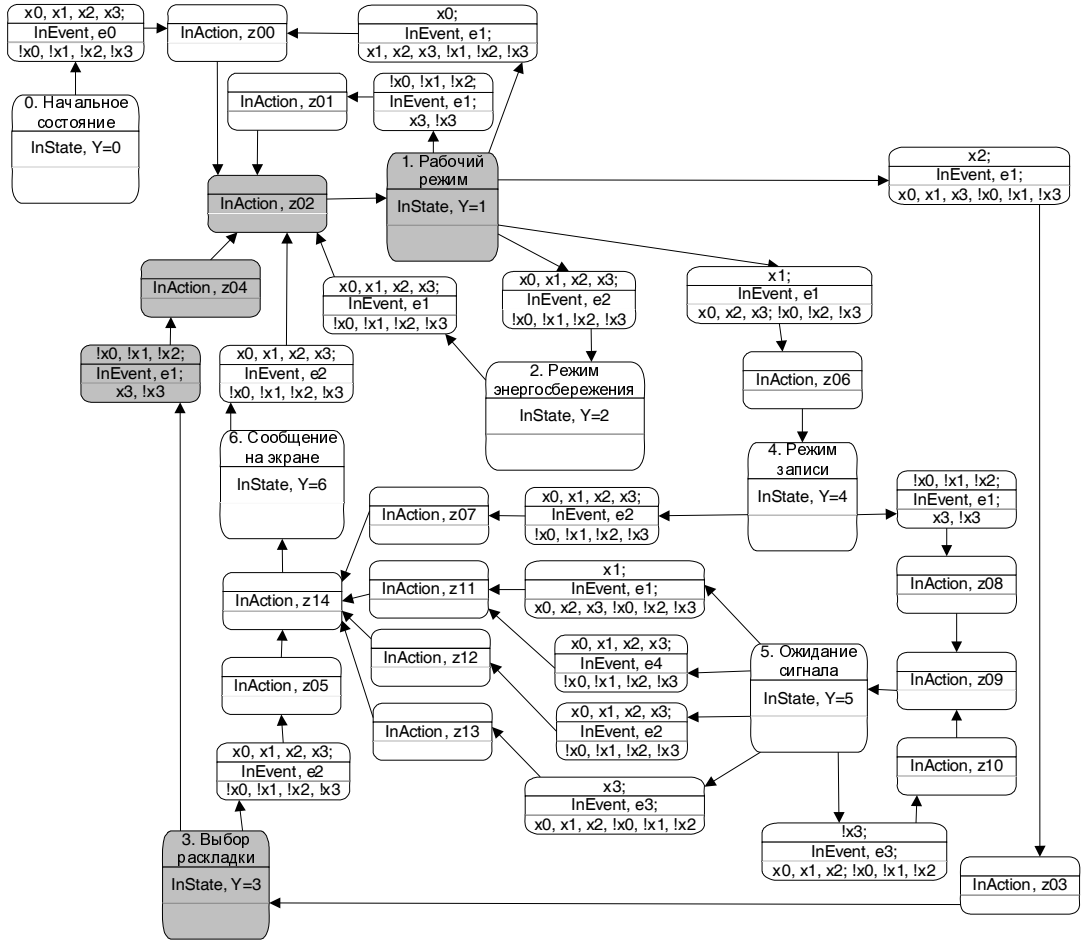
При использовании нестандартных методов моделирования интерпретировать результаты разработчику приходится самому — ему придется проводить анализ путей прямо на модели Крипке, которую он сам (вручную) и построил.

После того, как отработала программа-верификатор, необходимо определить выполнимость формул спецификации на определенных участках автомата. Среди этих участков могут быть состояния, события, выходные воздействия. Сценарий для любой подформулы спецификации представляет собой бесконечный путь в модели Крипке, иллюстрирующий справедливость или ошибочность данной подформулы. Требуется, чтобы сценарий, предъявленный программой, был представлен в исходном автомате. Изображать этот путь следует конечным (вспомним утверждение из предыдущего раздела).

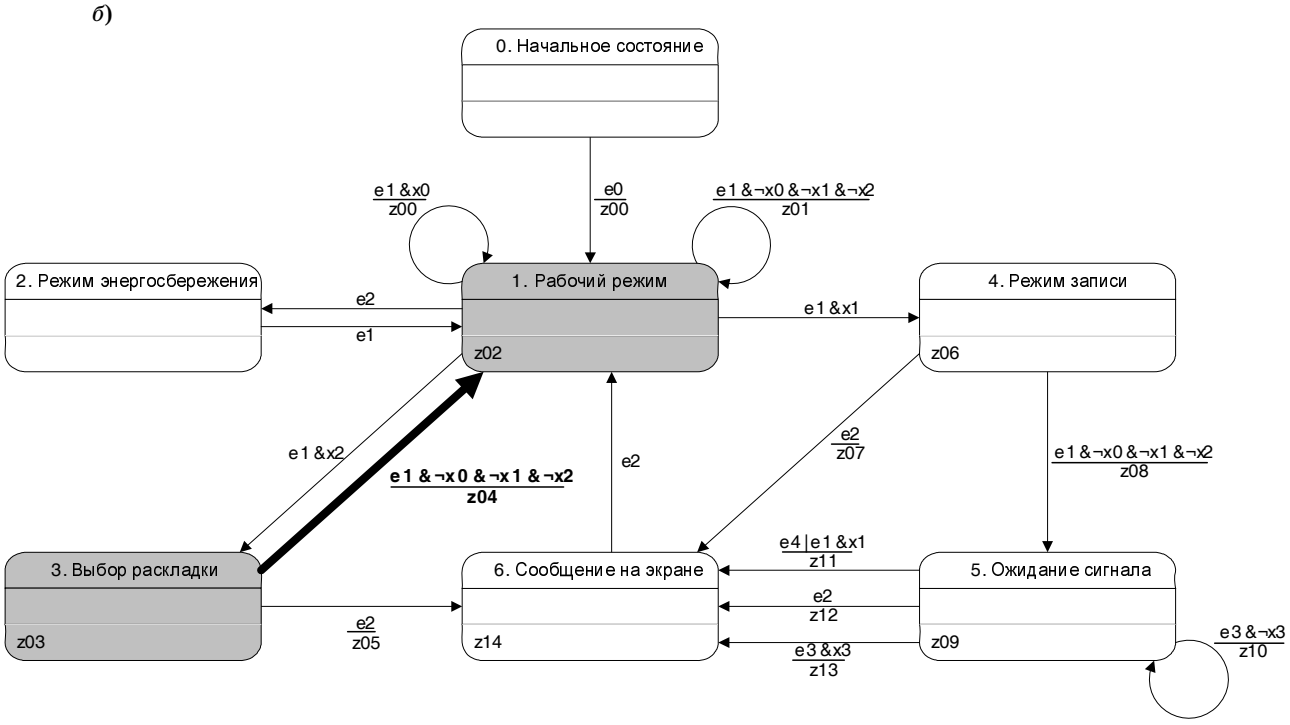
Что касается «переноса» пути из модели Крипке в автомат, то данная операция (скажем, для «редуцированной» схемы) выполняется однозначно. Действительно, состояния модели, содержащие атомарное предложение *InState*, однозначно преобразуются в соответствующие им состояния автомата. Путь же между любыми двумя соседними состояниями проходит ровно через одно состояние-событие, из атомарных предложений которого можно узнать, какое событие ведет по данному пути из исходного состояния, а также значения существенных и список несущественных входных переменных в момент, когда произошло это событие. Эта информация однозначно определяет направление, вдоль которого строится путь в исходном автомате Мили. Если же путь (или его участок) начинается не в состоянии *InState*, то обратная трассировка пути позволяет узнать состояние *InState*, предшествующее текущему, и всю необходимую информацию относительно того, как попасть в текущее состояние.

Рассмотрим ***пример*** для автомата *ARemote*. Пусть для состояния *3* выполняется верификация формулы $\neg\mathbf{E}[\neg(Y = 6)\mathbf{U}(Y = 1)]$ (в состояние *1* нельзя попасть, минуя состояние *6*). Эта формула

а)



б)



■ ***Рис. 11.*** *Контрпример: а — путь в модели Крипке; б — путь в автомате Мили*

в состоянии *3* не выполняется. Верификатор сгенерировал (кратчайший и единственный в данном случае) контрпример, который на рис. 11, *а* выделен серым цветом. Это конечный путь, любое продолжение которого удовлетворяет формуле $\neg\mathbf{E}[\neg(Y = 6)\mathbf{U}(Y = 1)]$.

Этот же путь, но представленный в исходном автомате, приведен на рис. 11, *б*.

В случае, когда при моделировании выполнялась композиция автоматов/моделей Крипке, независимая нумерация их состояний позволит для каждого перехода в пути, представленном в окончательной модели, однозначно решить вопрос, в какой именно индивидуальной компоненте системы взаимодействующих автоматов произошел переход. Это, опять же, дает возможность отобразить путь на модели в путь на исходном автомате.

Заключение

В работе были предложены методы для моделирования автомата Мили структурами Крипке. Был разработан алгоритм для построения сценариев и их интерпретации в исходном автомате. В связи с созданием этого алгоритма было сформулировано утверждение, позволяющее привести все сценарии к общему виду.

Составление сценариев (в том числе, контрпримеров) с помощью верифицирующих инструментов позволяет проводить исследования в области автоматической или интерактивной коррекции модели или автомата с целью удовлетворить предъявляемым условиям. Например, если программа-верификатор предъявила путь, опровергающий некоторое желательное свойство для системы, она может предложить разработчику исказить/ликвидировать этот путь, например, за счет удаления какого-либо перехода. При этом, разумеется, не гарантируется, что в модели тогда не возникнет других противоречий со спецификацией, хотя не исключается возможность и более интеллектуальной коррекции.

Исходя из изложенного можно кратко сформулировать основные достоинства автоматных программ в части их верификации [19].

1. Класс автоматных программ является наиболее удобным для верификации методом *Model Checking*, так как в этом случае модель программы может быть автоматически построена по спецификации ее поведения, задаваемой в общем случае системой взаимодействующих конечных автоматов, в то время как для программ других классов модель приходится строить вручную.

2. Структура автоматных программ, в которых функции входных и выходных воздействий почти полностью отделены от логики программ, делает практичным верификацию этих функций на основе формальных доказательств с использованием пред- и постусловий [20, 21].

Литература

1. **Джексон Д.** Программы проверяют программы // В мире науки. 2006. № 10. С. 52–57.
2. **Вудкок Дж.** Первые шаги к решению проблемы верификации программ //Открытые системы. 2006. № 8. С. 36–57.
3. **Katoen J.-P.** Concepts, Algorithms, and Tools for Model Checking. Lehrstuhl für Informatik VII, Friedrich-Alexander Universität Erlangen-Nürnberg. Lecture Notes of the Course (Mechanised Validation of Parallel Systems) (course number 10359). Semester 1998/1999.
4. **Шалыто А. А.** SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/switch/1
5. **Шалыто А. А.** Логическое управление. Методы аппаратной и программной реализации. СПб.: Наука, 2000. http://is.ifmo.ru/books/log_upr/1
6. **Кузьмин Е. В., Соколов В. А.** Верификация автоматных программ с использованием LTL // Моделирование и анализ информационных систем / ЯрГУ. Ярославль. 2007. № 1. С. 3–14. http://is.ifmo.ru, раздел «Верификация».
7. **Finite state machine.** http://en.wikipedia.org/wiki/Finite_state_machine
8. **Mealy machine.** http://en.wikipedia.org/wiki/Mealy_machine
9. **Moore machine.** http://en.wikipedia.org/wiki/Moore_machine
10. **Sebastiani R.** Introduction to Formal Methods, 2005–2006. http://dit.unitn.it/~rseba/DIDATTICA/fm2005/02_transition_systems.pdf
11. **Margaria T.** Model Structures. Service Engineering – SS 06. https://www.cs.uni-potsdam.de/sse/teaching/ss06/sveg/ps/2-ServEng-Model-Structures.pdf
12. **Roux C., Encrenaz E.** CTL May Be Ambiguous when Model Checking Moore Machines. UPMC LIP6 ASIM. CHARME, 2003. http://sed.free.fr/cr/charme2003-presentation.pdf
13. **Hull R.** Web Services Composition: A Story of Models, Automata and Logics. Bell Labs, Lucent Technologies, 2004. http://edbtss04.dia.uniroma3.it/Hull.pdf
14. **Миронов А. М.** Математическая теория программных систем. http://intsys.msu.ru/study/mironov/mthprogsys.pdf
15. **Сайт проекта UniMod.** http://unimod.sf.net
16. **Сайт eVelopers Corporation.** http://www.evelopers.com
17. **Вельдер С. Э., Бедный Ю. Д.** Универсальный инфракрасный пульт для бытовой техники: Курсовая работа / СПбГУ ИТМО, 2005. http://is.ifmo.ru/projects/irrc/
18. **Clarce E. M., Emerson E. A., Sistla A. P.** Automatic Verification of Finite-State Concurrent Using Temporal Logic Specifications //ACM Transactions on Programming Languages and Systems (TOPLAS). 1986. Vol. 8. N 2. P. 244–263.
19. **Switch-technology.** http://en.wikipedia.org/wiki/Switch-technology
20. **Дейкстра Э.** Заметки по структурному программированию // Дал У., Дейкстра Э., Хоор К. Структурное программирование. М.: Мир, 1975.
21. **Мейер Б.** Объектно-ориентированное конструирование программных систем. М.: Русская редакция, 2005.