Санкт-Петербургский государственный университет
информационных технологий, механики и оптики
Кафедра компьютерных технологий

С.Э. Вельдер, А.А. Шалыто

Введение в верификацию автоматных программ на основе метода *Model checking*

Санкт-Петербург
2006

Оглавление

Введение

Верификация моделей – это набор идей и методов для построения моделей работающих программ, математической формулировки требований к ним, отражающих правильность их работы (называемых также **спецификацией**), и создания алгоритмов для формальной проверки (доказательства или опровержения) этих требований (свойств).

В данной работе основное внимание концентрируется на особой технике (формальной) верификации, которая базируется на так называемых *темпоральных логиках* и позволяет уменьшить участие разработчика в верификационном процессе. Эта техника называется *Model checking* (проверка на моделях) [1–5].

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.

Основная идея *Model checking* состоит в **моделировании** – описании разработчиком поведенческой модели системы, подлежащей верификации, и **спецификации** – формулировке требований (желаемого поведения системы). Обратим внимание, что модель программы не всегда полно отражает ее поведение. Разработчик при построении модели, как правило, абстрагируется от несущественных ее свойств. Такая концепция дает возможность уменьшить размер самой модели и ускорить процесс ее проверки.

Если модель удовлетворяет указанным требованиям, то программа-верификатор сообщает об этом. Если же обнаруживается ошибка, то она предоставляет *контрпример*, который показывает, при каких условиях могло возникнуть данное несоответствие.

Контрпример представляет собой сценарий, в котором модель ведет себя нежелательным образом. Это означает, как правило, что модель ошибочна и подлежит пересмотру. Правда, в некоторых случаях это может означать, что формальные требования неверны, в том смысле, что средство верификации проверяет то, что разработчик не желал проверить.

Проверка модели позволяет разработчику обнаружить ошибку и исправить модель или требования. Если не найдено ни одной ошибки, разработчик может усовершенствовать описание модели (сделать модель более реалистичной, приняв во внимание больший набор свойств), как правило, увеличив ее размер, и перезапустить процесс верификации.

Основная трудность моделирования – не потерять важные детали программы, а трудность задания требований – сформулировать их корректно и исчерпывающе.

Алгоритмы для *Model checking* обычно базируются на полном просмотре пространства состояний модели: для каждого состояния проверяется, удовлетворяет ли оно сформулированным требованиям. Алгоритмы гарантированно завершаются, так как модель конечна.

Принципиальная схема *Model checking* приведена на рис. 1.

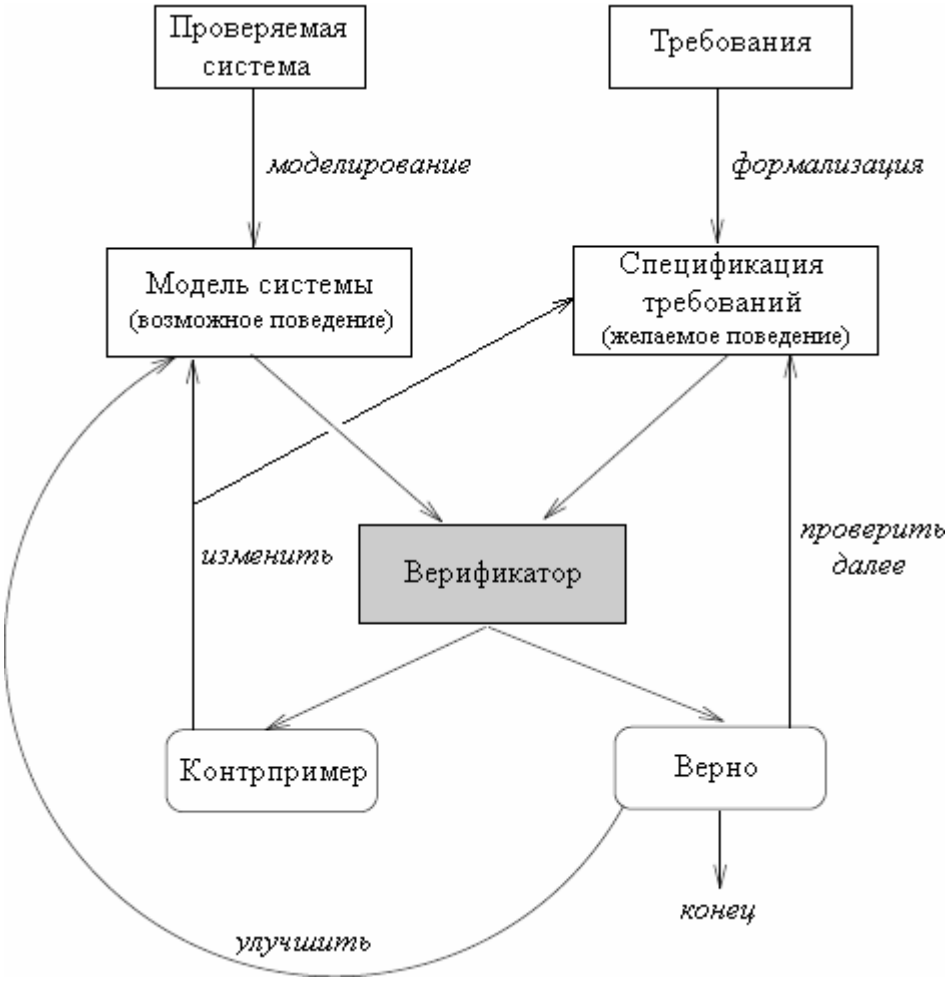


Рис. 1. Model checking

В проблематике верификации [6] сформировалось два направления: аксиоматическое и алгоритмическое. При первом из них разрабатывается набор аксиом, с помощью которого может быть описана как сама система, так и ее свойства [7]. Основу второго направления составляет *Model checking*.

Цель исследований в этой области – сформулировать ясную логическую основу для создания автоматических систем верификации программ.

Model checking берет начало с независимых работ двух пар авторов: J. Quielle и J. Sifakis [8] и E. M. Clarke и E. A. Emerson [9] (в этой же работе был введен термин *Model checking*).

Достоинства метода *Model checking*:

1. Эффективность. Программы для верификации моделей способны работать с достаточно большими пространствами состояний благодаря концепции упорядоченных двоичных разрешающих деревьев [1], которая также упоминается в данной работе. В литературе описан пример с числом состояний порядка 10^{130}.
2. Контрпримеры.

Ограничения *Model checking*:

1. Поддержка только конечных моделей. 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 *Model checking* проверяется *модель* системы вместо реальной системы. Таким образом, любое применение метода *Model checking* настолько же качественно, как и сама модель системы.
3. Для многопроцессорных систем размер пространства состояний в худшем случае пропорционален произведению размеров пространств состояний их индивидуальных компонент. Так как *Model checking* выполняется на моделях,

близких по структуре к конечному автомату, то для сложных многопроцессорных систем эта концепция перестает быть эффективной. Этот эффект называется проблемой показательного (экспоненциального) взрыва состояний (*state-space explosion problem*).

В рамках данной работы рассматривается программирование с явным выделением состояний [11–13], и поэтому ограничения 1 и 3 в нашем случае несущественны.

Основное отличие метода *Model checking* от классической формальной (или **хоаровской**) верификации состоит в том, что первый метод позволяет проверять динамические свойства программ – те, которые можно записать с помощью темпоральной (временно́й) логики, а второй метод проверяет, соответствует ли состояние переменных на выходе из программы условиям, накладываемым на их входное состояние [14].

Перечислим основные положения настоящей работы.

1. В автоматных программах [11, 12] поведение специфицируется с помощью конечных автоматов. В работе в основном применяются спецификации, состоящие из одного автомата *Мили*.

2. Использование подхода *Model checking* для таких программ связано с преобразованием автомата *Мили* в структуру *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*. Ввиду того, что среди языков темпоральной логики простейшим по семантике является язык *PLTL*, в главе 1 изложение начинается с него.

4. Собственно верификация выполняется по структуре *Крипке* и требованиям к ней.

5. Сценарий для структуры *Крипке* преобразуется в сценарий для автомата *Мили* (глава 4).

6. Все этапы изложенной технологии верификации рассматриваемого класса автоматных программ иллюстрируются на примере программы для «Универсального инфракрасного пульта для бытовой техники» [15].

Глава 1. Основные понятия

1.1. Синтаксис языка *PLTL*

Линейную темпоральную логику предложил A. Pnueli [10] для спецификации и верификации реактивных систем – систем, постоянно взаимодействующих с окружением и реагирующих на события.

Пусть символ *AP* обозначает множество, которое будем называть множеством атомарных предложений (позже будет показано, что для автоматной

технологии атомарными предложениями являются события, входные и выходные переменные и т. п.).

Тогда множество всех формул, определяемых в нотации Бэкуса-Наура, как $\phi ::= p \in AP \mid \neg\phi \mid \phi \vee \phi \mid \mathbf{X}\phi \mid \phi \mathbf{U} \phi$, составляет синтаксис языка *PLTL* (*propositional linear temporal logic*).

Обратим внимание, что множество формул, построенных по первым трем правилам, дает все формулы пропозициональной логики – истинный подкласс языка *PLTL*. **X** (*neXt*) и **U** (*Until*) являются темпоральными операциями.

Для упрощения введем две темпоральные операции:

$$\mathbf{F}\phi \equiv \mathbf{1}\mathbf{U}\phi,$$
$$\mathbf{G}\phi \equiv \neg\mathbf{F}\neg\phi.$$

В дальнейшем будем писать упрощенные формулы, не обращаясь каждый раз к определению языка.

Таким образом, в языке *PLTL* операции **F** (*Future*) и **G** (*Globally*) выражены через базовую операцию **U**. Как будет показано в разд. 3.3, в языке *CTL* эти операции (точнее, их «разветвленные» варианты) сами могут быть базовыми, причем выбор базовых операций даст большую гибкость в разработке алгоритмов верификации.

1.2. Семантика языка *PLTL*

Пусть символ *S* обозначает множество, которое будем называть множеством состояний.

PLTL-моделью называется тройка $\boldsymbol{M} = (S;\ R: S \rightarrow S;\ Label \subseteq S \times AP)$, где R – тотальная (всюду определенная) функция на S, называемая отношением переходов.

Смысл формул логики определяется в терминах отношения семантической истинности (выполнения) $\models$ между моделью $\boldsymbol{M}$, одним из ее состояний s и формулой ϕ. Выражение $(\boldsymbol{M}, s, \phi) \in \models$ будем записывать в инфиксной нотации: $\boldsymbol{M}, s \models \phi$. Когда модель $\boldsymbol{M}$ ясна из контекста, будем исключать ее из формулы и писать $s \models \phi$.

Отношение выполнения $\models$ определяется в табл. 1.

Таблица 1. Семантика языка *PLTL*

$s \models p$	$\Leftrightarrow$	$(s, p) \in Label$
$s \models \neg\phi$	$\Leftrightarrow$	$\neg(s \models \phi)$
$s \models \phi \vee \psi$	$\Leftrightarrow$	$(s \models \phi) \vee (s \models \psi)$
$s \models \mathbf{X}\phi$	$\Leftrightarrow$	$R(s) \models \phi$
$s \models \phi \mathbf{U} \psi$	$\Leftrightarrow$	$\exists j \geq 0 \mid \left(R^j(s) \models \psi \wedge \left(\forall 0 \leq k < j : R^k(s) \models \varphi\right)\right)$

В заключение раздела отметим, что для семантики языка *PLTL* существует корректная и полная формальная система аксиом.

В настоящей работе этот язык в основном используется для упрощения понимания языка *CTL*.

1.3. Синтаксис языков *CTL* и *CTL**

Для множества атомарных формул *AP* набор *CTL*-формул[1] определяется следующим образом: $\phi ::= p \in AP \mid \neg\phi \mid \phi \vee \phi \mid \mathbf{EX}\phi \mid \mathbf{E}[\phi \mathbf{U} \phi] \mid \mathbf{A}[\phi \mathbf{U} \phi]$.

Введем также дополнительные темпоральные операции:

$$\mathbf{EF}\phi \equiv \mathbf{E}[\mathbf{1U}\phi], \qquad \mathbf{AF}\phi \equiv \mathbf{A}[\mathbf{1U}\phi];$$
$$\mathbf{EG}\phi \equiv \neg\mathbf{AF}\neg\phi, \qquad \mathbf{AG}\phi \equiv \neg\mathbf{EF}\neg\phi,$$
$$\mathbf{AX}\phi \equiv \neg\mathbf{EX}\neg\phi.$$

Таким образом, отличие языка *CTL* от *PLTL* состоит в том, что каждой введенной темпоральной операции соответствует префикс **E** или **A**, называемый *квантификатором пути*. Традиционные техники, используемые для эффективной проверки моделей линейной темпоральной логики, принципиально отличаются от тех, которые применяются для ветвящейся темпоральной логики. Синтаксис языка *CTL* требует, чтобы квантификаторы шли непосредственно перед темпоральными операциями **X**, **F**, **G** или **U**.

Если опустить это ограничение, то получится более выразительная ветвящаяся темпоральная логика *CTL** (Clarke и Emerson, 1981) [1]. Логика *CTL** позволяет квантификатору пути располагаться перед произвольной *PLTL*-формулой. Формально язык *CTL** имеет следующую нотацию:

$\phi ::= p \in AP \mid \neg\phi \mid \phi \vee \phi \mid \mathbf{E}\psi$ – формулы состояния

$\psi ::= \phi \mid \neg\psi \mid \psi \vee \psi \mid \mathbf{X}\psi \mid \psi \mathbf{U} \psi$ – формулы пути

[1] CTL – computational tree logic

Соотношения между выразительными свойствами языков *PLTL*, *CTL*, CTL^* приведены в разд. 1.5.

В настоящей работе подробно не рассматривается применение языка CTL^* для *Model checking*, так как формальная семантика его интуитивно ясна из интерпретации *CTL*, а задача проверки моделей для этой логики ***PSPACE***-полна по отношению к размеру спецификации [1]. В то же время для языка *CTL* существуют эффективные алгоритмы проверки моделей.

1.4. Семантика языка *CTL*

CTL-моделью для множества состояний *S* называется тройка

$\boldsymbol{M} = (S;\ R \subseteq S \times S;\ Label \subseteq S \times AP)$, где R – тотальное отношение на множестве *S*.

Эту тройку называют также *структурой Крипке* или *моделью Крипке*, так как Саул Крипке применял схожие структуры для построения семантик модальных логик – «ближайших соседей» темпоральной логики [16].

Путем будем называть бесконечную последовательность состояний $s_0 s_1 s_2 \ldots$ таких, что $(s_i, s_{i+1}) \in R$ для всех целых целых неотрицательных i.

Множество $P_{\boldsymbol{M}}(s)$ путей, которые начинаются в состоянии *s,* определяется следующим образом: $P_{\boldsymbol{M}}(s) = \{\sigma \in S^{\omega} \mid \sigma_0 = s\}$.

Отношение выполнения $\models$ строится аналогично такому же в языке *PLTL* и определяется согласно табл. 2.

Таблица 2. Семантика языка *CTL*

$s \models p$	$\Leftrightarrow$	$(s, p) \in Label$
$s \models \neg\phi$	$\Leftrightarrow$	$\neg(s \models \phi)$
$s \models \phi \vee \psi$	$\Leftrightarrow$	$(s \models \phi) \vee (s \models \psi)$
$s \models \mathbf{EX}\phi$	$\Leftrightarrow$	$\exists \sigma \in P_M(s) \mid \sigma_1 \models \phi$
$s \models \mathbf{E}[\phi \mathbf{U} \psi]$	$\Leftrightarrow$	$\exists \sigma \in P_M(s) \mid (\exists j \geq 0 \mid (\sigma_j \models \psi \wedge (\forall 0 \leq k < j : \sigma_k \models \varphi)))$
$s \models \mathbf{A}[\phi \mathbf{U} \psi]$	$\Leftrightarrow$	$\forall \sigma \in P_M(s) \mid (\exists j \geq 0 \mid (\sigma_j \models \psi \wedge (\forall 0 \leq k < j : \sigma_k \models \varphi)))$

1.5. Выразительные возможности языков *PLTL, CTL, CTL**

Для сравнения выразительности представим синтаксис языков *PLTL* и *CTL* в терминах языка CTL^*. Например, будем говорить, что формула ψ логики *PLTL* записана в терминах языка CTL^*, если $\psi = \mathbf{A}\phi$, где $\phi \in PLTL(AP)$ (простейший подход).

Для языка *CTL* перепишем синтаксис в следующем виде:

$\phi ::= p \in AP \mid \neg\phi \mid \phi \vee \phi \mid \mathbf{E}\psi$ – формулы состояния

$\psi ::= \neg\psi \mid \mathbf{X}\phi \mid \phi \mathbf{U} \phi$ – формулы пути

Данное определение синтаксиса языка *CTL* эквивалентно исходному его определению (разд. 1.3).

Из рассмотрения синтаксиса следует, что $PLTL, CTL \subseteq CTL^*$. Обратное неверно (синтаксически).

Будем называть формулы ϕ и ψ языка CTL^* эквивалентными, если

$$M, s \models \phi \Leftrightarrow M, s \models \psi$$

для любой модели ***M*** и ее состояния *s*.

Пусть *L* и *L′* – темпоральные логики. Будем говорить, что *L* не более выразительна, чем *L′*, если для любой формулы из *L* найдется эквивалентная ей формула из *L′*. Здесь предполагается, что $L, L' \subseteq CTL^*$.

Теперь все готово для сравнения выразительных возможностей. Рис. 2 отражает соотношения между тремя рассматриваемыми логиками в терминах выразительности.

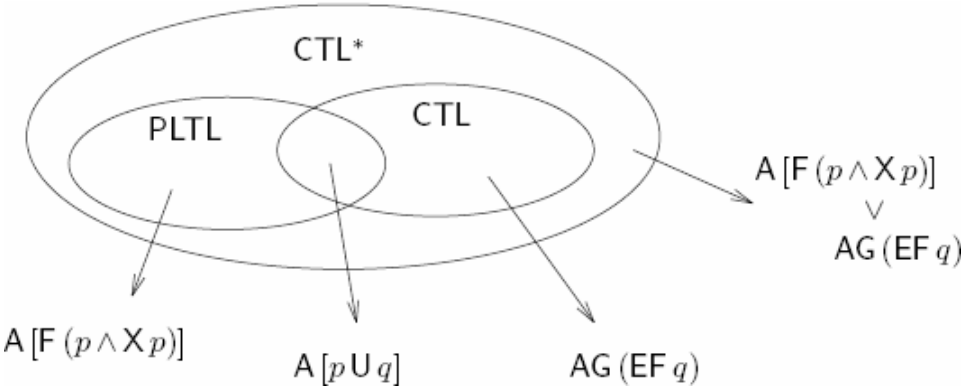


Рис. 2. Соотношения между *PLTL*, *CTL* и CTL^*

Более подробно о выразительной силе языков *PLTL*, *CTL*, CTL^* изложено в работах [1, 17].

1.6. Технология верификации одного класса автоматных программ

Технология программирования с явным выделением состояний [18] работает с такими объектами, как автомат *Мили* [19], автомат *Мура* [20] и смешанный автомат, которые легко интерпретируются с помощью моделей *Крипке*.

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 желании, всегда может быть преобразован в автомат *Мура*.

Исследования в данной области (моделирование автомата и конвертация его в структуру *Крипке*) проводились в работах [21–24]. При этом конвертация была сопряжена со следующими проблемами:

1. Трудности с выполнением композиции автоматов.
2. Неоднозначность интерпретации формулы языка *CTL* в исходном автомате.

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

Методы моделирования, рассматриваемые в настоящей работе (в частности, «редуцированная» схема из разд. 2.4), 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 [25, 26].

Кроме того, при использовании *SWITCH*-технологии число управляющих состояний относительно небольшое. 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 таких алгоритмов будет рассмотрен в главе 3.

Во введении были перечислены основные этапы технологии верификации рассматриваемого класса автоматных программ. Повторим определение этих этапов, прокомментировав более подробно некоторые из них.

1. В автоматных программах [11, 12] поведение специфицируется с помощью конечных автоматов. В настоящей работе в основном применяются спецификации, состоящие из одного автомата *Мили*. В общем случае модель может состоять из нескольких взаимодействующих автоматов. Для верификации таких систем применяется композиция исходных автоматов или моделей *Крипке* (изложено в разд. 2.1).

2. Использование *Model checking* для программ в данной работе связано с преобразованием автомата *Мили* в структуру *Крипке*.

3. Использование структуры *Крипке* предполагает применение темпоральной логики для записи требований. В настоящей работе требования описываются на языке *CTL*.

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

Верификация осуществляется с использованием двух алгоритмов. Первый из них предназначен для определения набора состояний в структуре *Крипке*, в которых выполняется заданное формулой требование (разд. 3.3), а второй – по заданному исходному состоянию и подформуле заданного требования с помощью построенного набора состояний формирует сценарий, который подтверждает или опровергает эту подформулу (разд. 3.4).

5. Сценарий для модели *Крипке* преобразуется в сценарий для автомата *Мили* (глава 4).

6. Все этапы изложенной технологии верификации рассматриваемого класса автоматных программ иллюстрируется на примере программы для «Универсального инфракрасного пульта для бытовой техники» [15].

Таким образом, схематично процесс, исследуемый в данной работе, можно представить так, как изображено на рис. 3.

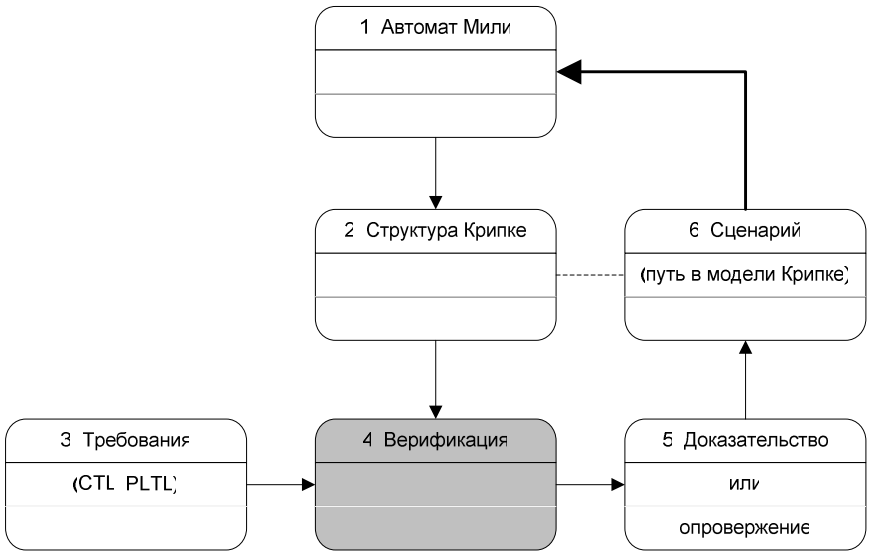


Рис. 3. Этапы верификации автоматных программ

Глава 2. Преобразование автомата *Мили* в структуру *Крипке* и разработка требований

2.1. Общие положения

В этой главе будут построены несколько различных схем для генерации множества атомарных предложений автоматной программы, преобразования автомата *Мили* в модель *Крипке* и записи требований к программе.

Выделим три основных 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 внутрь атомарной формулы.

Каждому из этих путей соответствует схема, описываемая в одном из разделов настоящей главы (разд. 2.2–2.4).

В разд. 2.5 рассматриваются способы упрощения графа переходов для того, чтобы к нему стали применимы специфические алгоритмы.

Отметим 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.

При этом отметим, что 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. Также 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 *A*.
3. Систему взаимодействующих автоматов можно привести к одному с помощью композиции (произведения) [21]. Также можно выполнять сначала моделирование каждого автомата, а после него – композицию моделей *Крипке* (рис. 4).

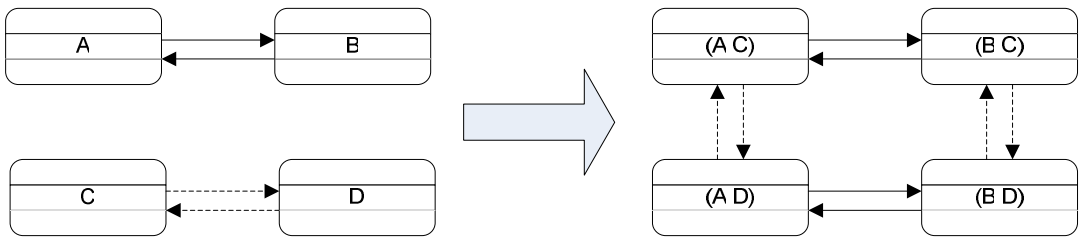


Рис. 4. Композиция структур *Крипке*

Выбор способа определяется соображениями эффективности и удобства. В примерах, исследуемых далее, будем считать, что данный вопрос уже решен, и рассматривать 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*, и 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.

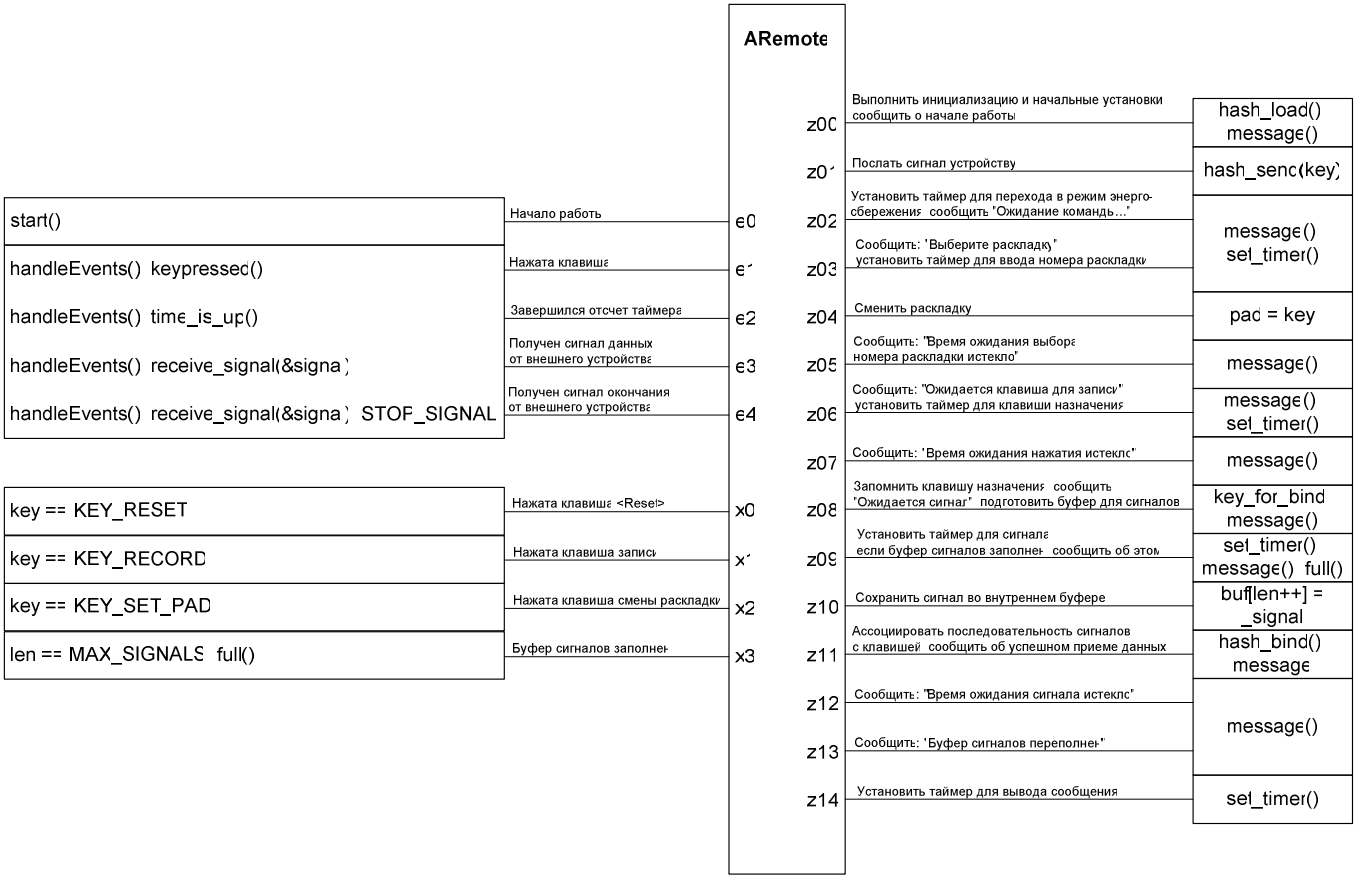
Далее будут рассмотрены индивидуальные особенности каждой из трех схем.

2.2. Схема «Состояния на событиях и выходных воздействиях»

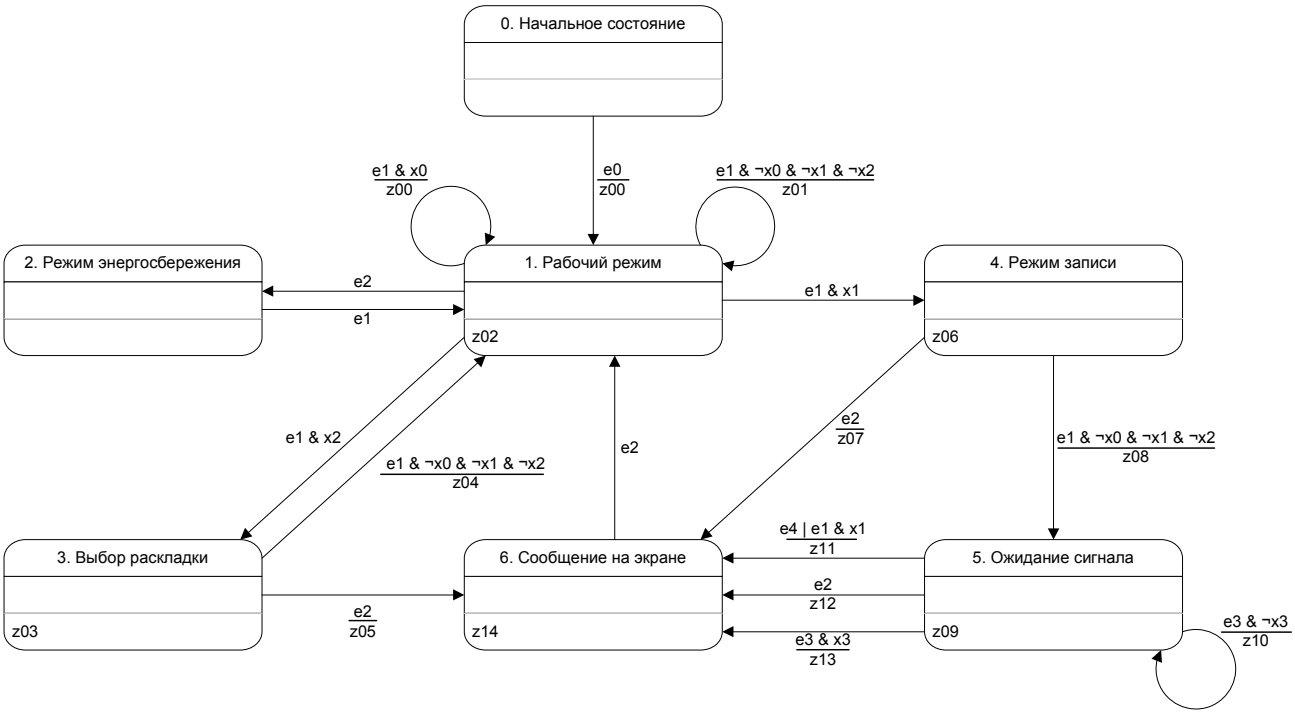
Схема «Состояния на событиях и выходных воздействиях» (*CCBB*), как и другие две, наследует общую идеологию моделирования, описанную выше. От других схем ее отличает то, что кроме указанного общего принципа в ней больше ничего не содержится. Таким образом, применяя схему *CCBB* для автоматной программы, 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 [15]. Эмулятор представлен с помощью одного автомата *ARemote*, схема связей и граф переходов которого изображены на рис. 5.



а



б

Рис. 5. Схема связей автомата *ARemote* (а). Граф переходов автомата *ARemote* (б)

В рассматриваемом примере модель *Крипке* для автомата, построенная по схеме *CCBB*, будет изоморфна графу на рис. 6.

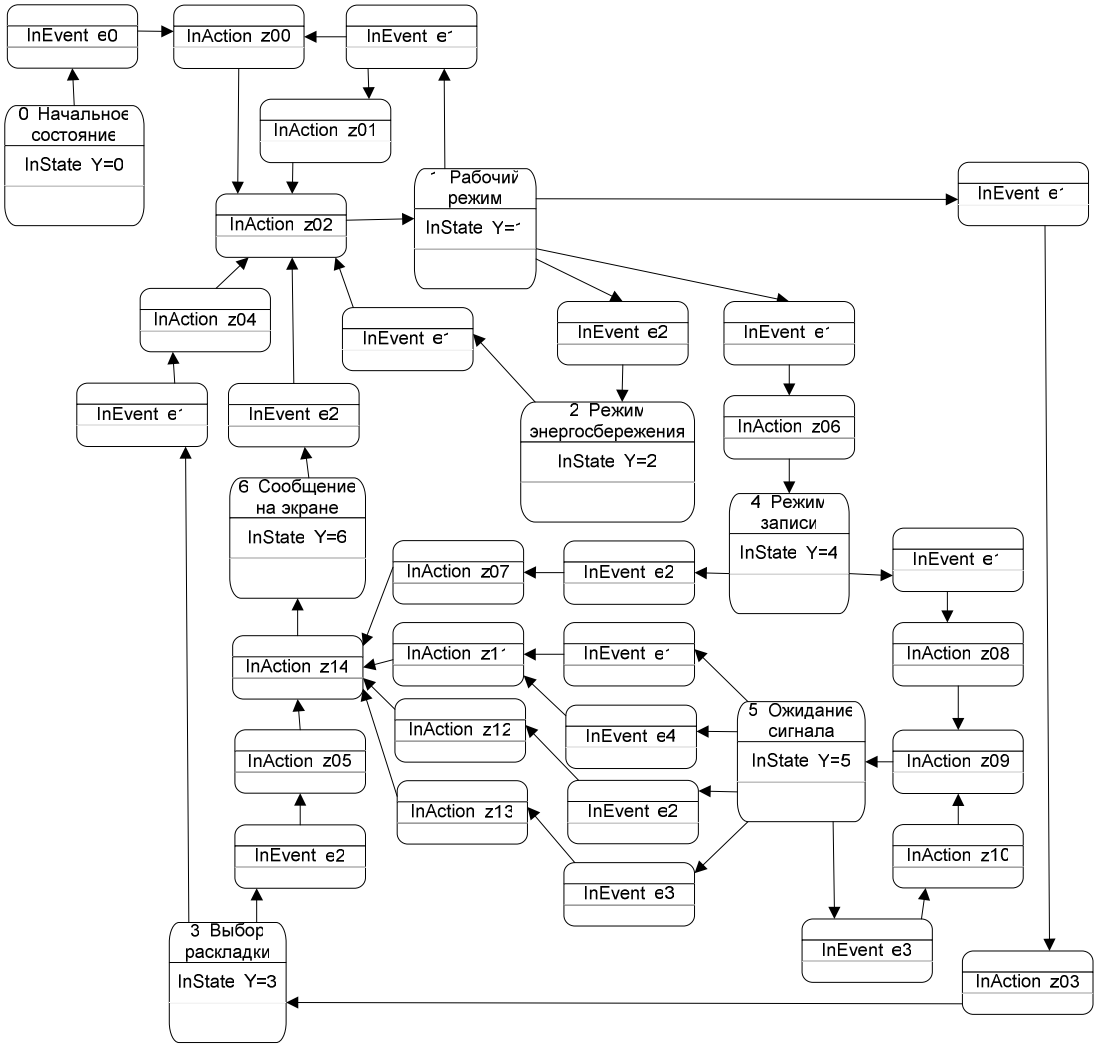


Рис. 6. Модель *Крипке*, построенная по схеме *CCBB*

В модели *Крипке*, изображенной на рис. 6, состояния-события и состояния-выходные воздействия указаны явно. При интерактивном моделировании совместно с исполнением и визуализацией [25, 26] их целесообразно обозначать, как и в исходном автомате, просто метками на дугах.

Итак, *CTL*-модель готова. 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 4–6 *исходного автомата* и только для них (для модели *Крипке* таких состояний больше).

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

В таком случае снабдим эти 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 (или множество истинных переменных). После получения полного автомата преобразуем его в модель *Крипке* по общей схеме (разд. 2.1) 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.

2.4. «Редуцированная» схема

Основным недостатком предыдущей схемы было большое число генерируемых состояний для модели *Крипке*, а достоинством – полнота.

В данном разделе семантика моделей будет изменена таким образом, чтобы число состояний в них можно было уменьшить, не потеряв при этом их выразительную силу. Это можно сделать так, что размер модели изменяется 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 изменим.

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 | e1&x1) / z11*” графа *ARemote*, изображенного на рис. 5,б, 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 (разд. 2.2), 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.

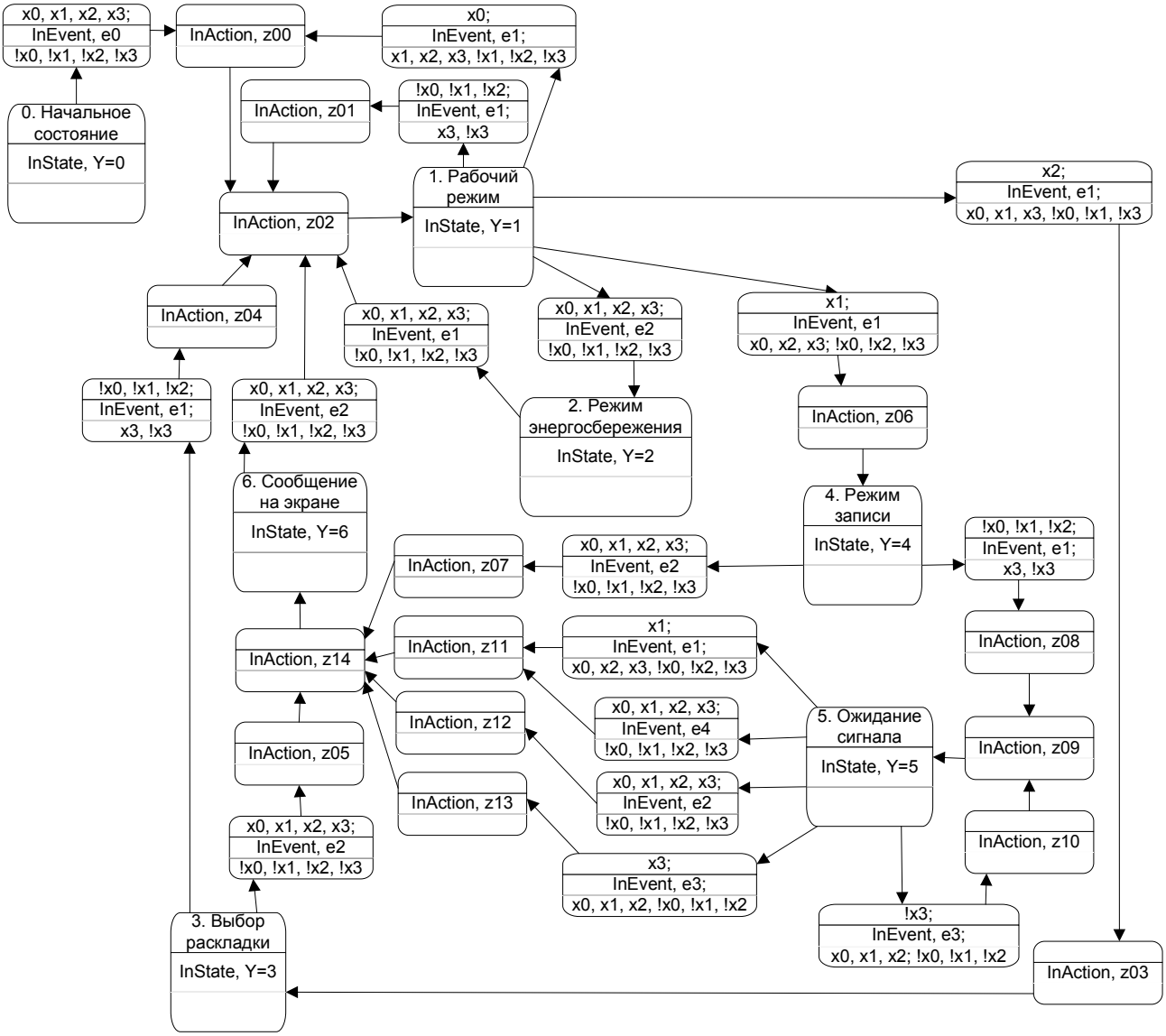


Рис. 7. Редуцированная модель *Крипке* для автомата *ARemote*

Размер модели на рис. 7 совпадает с размером модели, созданной по схеме «Состояния на событиях и выходных воздействиях». Вообще говоря, 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 *тесными 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: **E**[*(InEvent* → ¬ *x0)* **U** *z00*].

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

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 подойдет.

2.5. 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 разд. 2.1 – 2.4.

Приведем несколько альтернативных способов моделирования автоматов.

1. 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. Например, для автомата *ARemote* результатом будет модель на рис. 8.

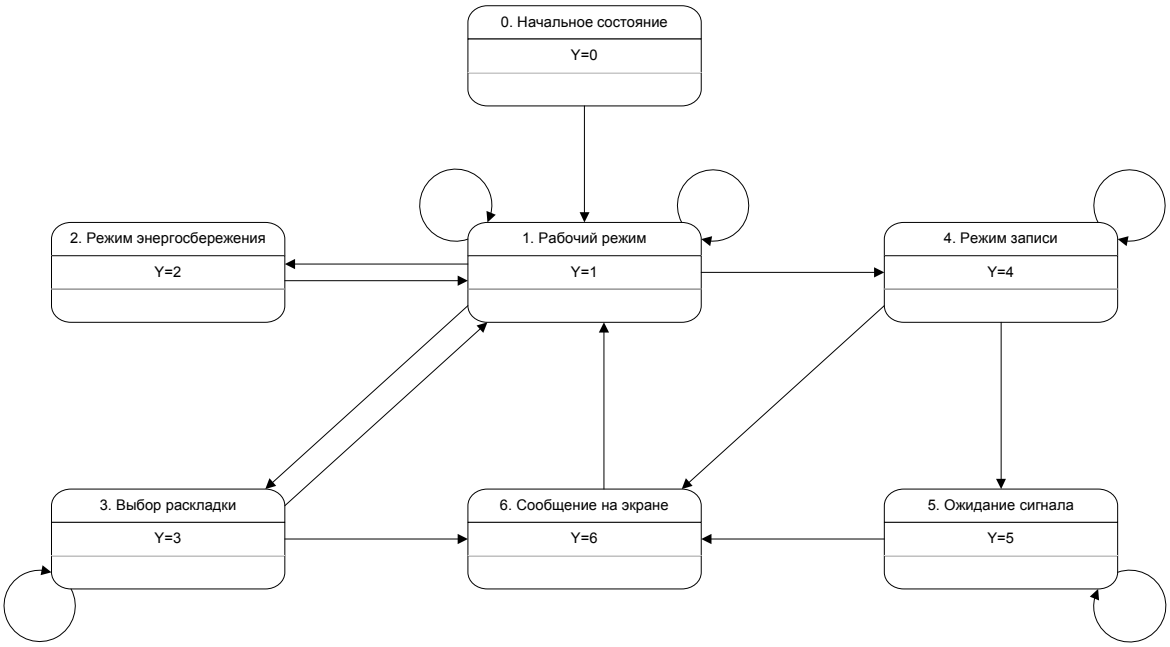


Рис. 8. Сокращенная модель *ARemote* (без событий и выходных переменных)

2. Для выделения существенных свойств автомата можно удалять части его графа переходов, оставляя только подграф, следя при этом за тем, чтобы не нарушалась качественная структура модели, обеспечивающая выполнение ее проверяемых свойств.
3. Заметим, что можно не только удалять подграфы, но и производить *гомеоморфную замену* одних участков графа другими, более простыми, опять же, с сохранением качественной структуры графа.

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

Отметим, что при построении в явном виде автоматной модели программы, написанной вне рамок *SWITCH*-технологии, может возникнуть проблема экспоненциального роста.

Примером задачи, иллюстрирующей эту проблему, является так называемая задача *ABRO*, применяемая в синхронном программировании [27–31]: сгенерировать сигнал *O* (*Output*), как только одновременно возникнут два сигнала *A* и *B*. Сигнал *R* (*Reset*) инициализирует систему заново. На рис. 9 приведен автомат *Мили*, формализующий решение задачи *ABRO*.

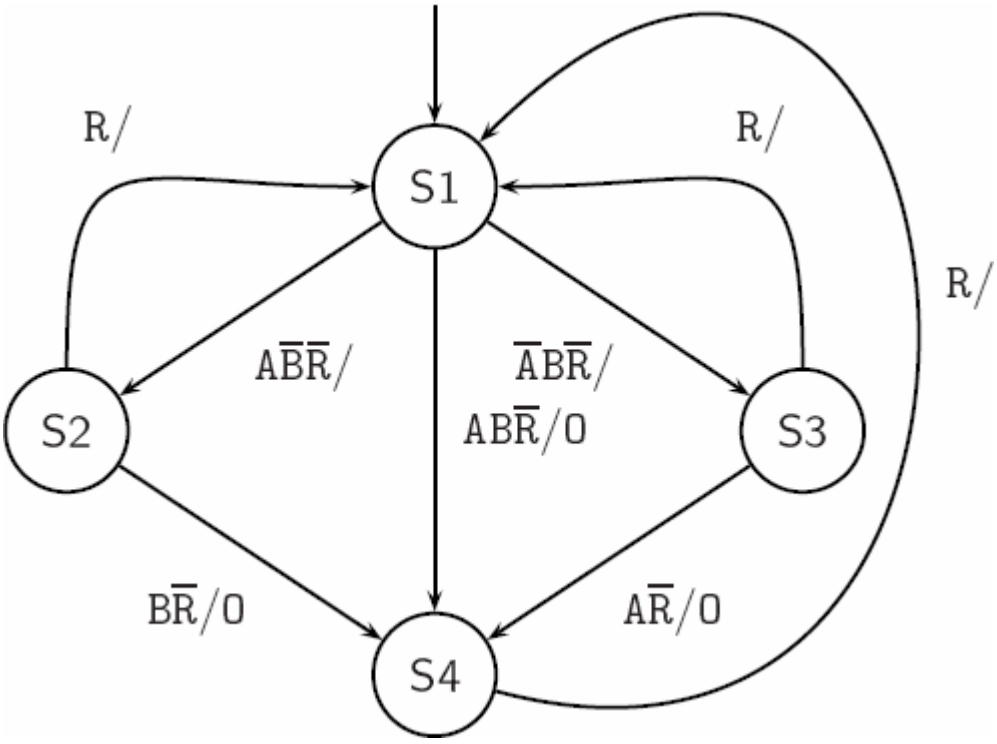


Рис. 9. Автомат *Мили* для решения задачи *ABRO*.
Два управляющих сигнала и *Reset* ведут к четырем состояниям

Если решать аналогичную задачу с n управляющими сигналами («$A_1A_2...A_nRO$»), то для автомата *Мили* потребуется 2^n состояний. Это означает, что

размер автоматной модели растет экспоненциально по отношению к размерности задачи.

Итак, на этом обзор методов моделирования и спецификации завершен. Описанные выше особенности, преимущества и недостатки этих методов позволяют утверждать, что в этой области еще есть пути для совершенствования.

Глава 3. *CTL*-верификация автоматных программ

3.1. Общие положения

Широко известные эффективные алгоритмы верификации *CTL* базируются на применении так называемой *теории неподвижной точки* [1, 33] совместно с существенным уменьшением размера модели. Такое уменьшение, как правило, происходит за счет применения технологии упорядоченных двоичных разрешающих деревьев. С другой стороны, пока используется программирование с явным выделением состояний, существенное уменьшение модели не всегда целесообразно, так как бывает связано с расходами на реализацию эффективных алгоритмов, в то время как их выигрыш во времени незаметен.

Работа *Bernholtz, Vardi, Wolper* (1994) привела к значительному совершенствованию использования «автоматного» подхода к ветвящейся темпоральной логике. Временная сложность их подхода линейна по отношению к длине *CTL*-формулы и размеру спецификации системы. Интересный аспект этого подхода заключается в том, что пространственная сложность их алгоритма принадлежит классу алгоритмов ***NLOGSPACE*** по отношению к размеру модели ***M***.

Это означает, что применение метода *Model checking* может потребовать память полиномиального размера по отношению к размеру спецификации системы.

Несмотря на то, что данные исследования интересны и многообещающи, существует эффективный и твердо устоявшийся метод (*Clarke, Emerson*, 1981), основанный на другой парадигме (теории неподвижных точек), которая является концептуально более простой. Из этого метода рассматривается только «каркас», так как ниже будет описан другой, еще более простой, алгоритм *CES* (*Clarke*, *Emerson, Sistla*, 1986), который основан на переформулировке синтаксиса языка *CTL.* В этом (более простом) методе алгоритм дополнен таким образом, что позволяет строить подтверждающие сценарии для проверяемых формул. Применять этот алгоритм будем для изображенных явно моделей *Крипке*.

3.2. Идея Кларка и Эмерсона

Под локальной задачей верификации обычно понимается вопрос: выяснить для данных $\boldsymbol{M}$, s, ϕ, справедливо ли $\boldsymbol{M}, s \models \phi$.

При построении алгоритма Кларк и Эмерсон формулируют глобальную задачу верификации: для данных $\boldsymbol{M}$ и ϕ построить множество всех s, для которых $\boldsymbol{M}, s \models \phi$.

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

Опишем идею их алгоритма на псевдокоде (рис. 10).

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**	$Sat_{EU}(\phi_1, \phi_2)$
	$\phi = \mathbf{A}[\phi_1 \mathbf{U} \phi_2]$	$\rightarrow$	**return**	$Sat_{AU}(\phi_1, \phi_2)$

end if
end

Рис. 10. Алгоритм «индукция по подформулам»

Как следует из рассмотрения текста программы, множество состояний, выполняющих формулу ϕ, строится индукцией по построению ϕ. Для подформул вида $\mathbf{E}[\phi_1 \mathbf{U} \phi_2]$ и $\mathbf{A}[\phi_1 \mathbf{U} \phi_2]$ множество выполняющих состояний получается с использованием методов неподвижной точки. При этом модель *Крипке* кодируется в упорядоченную бинарную разрешающую диаграмму.

Упорядоченные бинарные разрешающие диаграммы (*OBDD – Ordered Binary Decision Diagram*) – это мощная техника сжатия обрабатываемых данных, которая позволяет выполнять манипуляции над объектами прямо в сжатом виде, минуя распаковку, преобразование и обратную упаковку. Она заключается в том, что обрабатываемый объект неявно кодируется в непрерывный битовый поток из 2^n бит, который потом интерпретируется как булева функция n переменных. Именно для этой функции и строится корневой направленный ациклический граф, представляющий собой разрешающую диаграмму. Изначально этот граф может представлять собой полное двоичное дерево высоты n, уровень i которого

соответствует *i*-й переменной, а каждый путь от корня к листу в нем соответствует определенному двоичному набору и значению функции на этом наборе (*основное свойство*). Очевидно, что для многих битовых строк, применяемых на практике, такое представление сильно избыточно, так как в полном дереве только число вершин уже равно 2^n – число, равное длине битового образа. Однако, преобразуя (редуцируя) этот граф и не утрачивая при этом его основного свойства, можно добиться очень эффективного представления (сжатых) данных. Над разрешающими диаграммами можно выполнять логические операции.

Таким образом, многие задачи могут решаться прямо на *ROBDD* (*Reduced OBDD*), без составления таблиц истинности и без представления графов в явном виде. В частности, к таким задачам относится задача верификации (*Model checking*), для которой технология *ROBDD* успешно применяется, так как она позволяет работать с весьма обширными пространствами состояний.

3.3. Верификация модели в языке *CTL* (алгоритм *CES*)

В разд. 1.3 дано определение синтаксиса языка *CTL* и введены несколько дополнительных темпоральных операций в нем. Сейчас одну из этих операций преобразуем в базовую – выразим темпоральную часть синтаксиса языка *CTL* через операции **EX**, **EU**, **EG**:

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

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

Алгоритм *CES* [1, 32], который описывается далее, наследует идею Кларка и Эмерсона: множество выполняющих состояний строится для каждой подформулы входной формулы (для каждого состояния создается список выполненных в нем подформул). Ввиду изменения набора базовых правил языка *CTL*, псевдокод, записанный на рис. 10, превращается в текст на рис. 11.

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. 11. Индукция по построению формулы в алгоритме *CES*

Опишем кратко алгоритм *CES*. Будем считать для удобства, что из исходного графа *Крипке* построен симметричный ему граф – граф, в котором все переходы заменены на противоположные.

1. Можно считать, что выполняющие множества для атомарных предложений содержатся в исходных данных (информация об этих множествах указана в самой модели *Крипке*).
2. Если известно выполняющее множество для формулы f, то можно построить его и для формулы $\neg f$.

3. Аналогично, если известны выполняющие множества для f и g, то построим их и для $f \vee g$.
4. Сделаем «один шаг назад» от выполняющего множества для f – получим выполняющее множество для **EX** f.
5. Для построения выполняющего множества формулы **E**[f **U** g] вначале отмечаем в качестве выполняющих все вершины, в которых соблюдается формула g, а потом построим от них деревья «обратных путей» **вдоль тех вершин, в которых выполнена формула *f***. Для этого пригодится симметричный граф.
6. Осталось только научиться строить выполняющее множество для **EG** f. Это построение выполняется в три этапа:
   - исключаем из графа вершины, которые не выполняют формулу f;
   - у нового графа находим компоненты сильной связности и отмечаем все вершины в них как выполняющие. Учитываем при этом, что если компонента состоит лишь из одной вершины, то она не считается компонентой сильной связности (за исключением случая, когда на данной вершине есть петля) – отношение сильной связности будем считать в общем случае иррефлексивным;
   - строим обратные деревья, начиная от этих компонент, попутно помечая все посещенные вершины как выполняющие формулу f.

Трудоемкость итогового алгоритма составляет $O(|f| (|S| + |R|))$ – растет билинейно относительно длины формулы и размеров модели.

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

3.4. Построение сценариев в модели *Крипке*

Итак, требуется *показать*, что в данном состоянии s модели $\boldsymbol{M}$ выполняется (или не выполняется) формула f.

Алгоритм генерации сценария для *CTL*-формулы f

1. Если f – атомарное предложение, то просто предъявим описание состояния s в модели $\boldsymbol{M}$ – 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 $\mathbf{EX}f$ предъявим вершину в модели *Крипке*, в которую из вершины s есть переход и которая выполняет f. Такая вершина обязательно существует, иначе на этапе верификации не обнаружилось бы, что формула $\mathbf{EX}f$ верна. Опровержение $\mathbf{EX}f$ (доказательство $\mathbf{AX}\neg f$) 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 $\mathbf{E}[f\,\mathbf{U}\,g]$ и $\mathbf{EG}\,f$ выполняется рекуррентным способом с использованием методов 1–4 рассматриваемой схемы. 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 $\mathbf{E}[f\,\mathbf{U}\,g]$ достаточно построить путь в графе, применяя шаг за шагом пункты 1–4 к рекуррентному разложению этой формулы, до тех пор, пока не попадем в вершину, выполняющую g.

Для доказательства $\mathbf{EG}\,f$ сделаем то же самое, пока не попадем в вершину, в которой уже были. Путь в этом случае, начиная с некоторого состояния, становится периодическим (рис. 12).

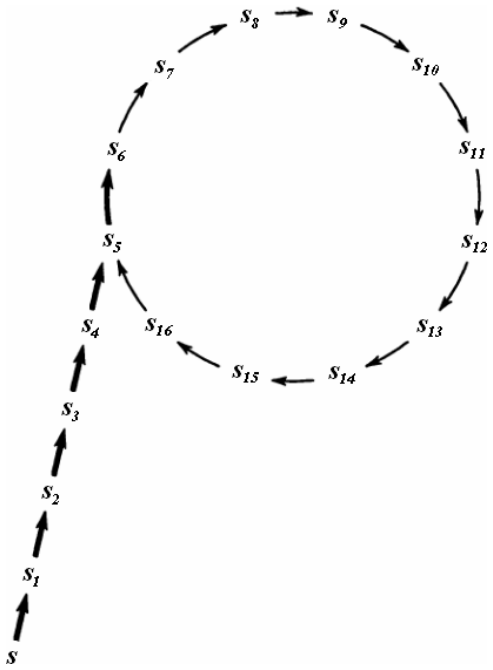


Рис. 12. Бесконечный ρ-путь

Опровержение формул $\mathbf{E}[f\,\mathbf{U}\,g]$ и $\mathbf{EG}\,f$ выполняется аналогично опровержению $\mathbf{EX}\,f$. **Любой** (бесконечный) путь, который начинается в текущей вершине, можно предъявить пользователю для рассмотрения, так как он (путь) не выполняет введенную формулу. Иначе говоря, вместо доказательства 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 (рис. 12).*

►Доказательство является конструктивным и целиком опирается на применение описанного алгоритма. Достаточно только заметить, что алгоритм всегда завершается, выдавая сценарий, который необходимо обладает свойством периодичности.◄

На этом завершим рассмотрение методов верификации автоматных программ. В этой главе были описаны исследования, проводившиеся в данной области, в том числе была упомянута концепция упорядоченных двоичных разрешающих диаграмм. Также был приведен алгоритм верификации модели, описанной в логике *CTL*, который был дополнен «модулем» для генерации сценариев к формулам.

Отметим, что в работе не рассматривались алгоритмы верификации линейных моделей (*PLTL*), основанные на применении ω-автоматов, называемых также автоматами *Бюхи* (Büchi). Подробнее об автоматах *Бюхи* и верификации моделей, описанных в логике *PLTL*, изложено в работах [1, 33–40].

Глава 4. Преобразование сценария для модели *Крипке* в сценарий для автомата *Мили*

Переходим к последней фазе процесса верификации в автоматном программировании (рис. 3). Ниже будет описано, как представлять путь (последовательность вершин) модели *Крипке* в виде пути исходного автомата *Мили*. Отметим сразу, что на протяжении этой главы предполагается, что модель *Крипке* была сгенерирована по «редуцированной» схеме, изложенной в разд. 2.4. Остальные схемы, для которых было дано описание, позволяют применять к себе аналогичный интуитивно ясный способ перехода от модели к автомату. Исключение составляют лишь специфические элементы разд. 2.5,

в котором описывались нетрадиционные методы моделирования, применяемые для случаев слишком больших автоматов. При использовании этих методов интерпретировать результаты разработчику приходится самому – ему придется проводить анализ путей прямо на модели *Крипке*, которую он сам (вручную) и построил.

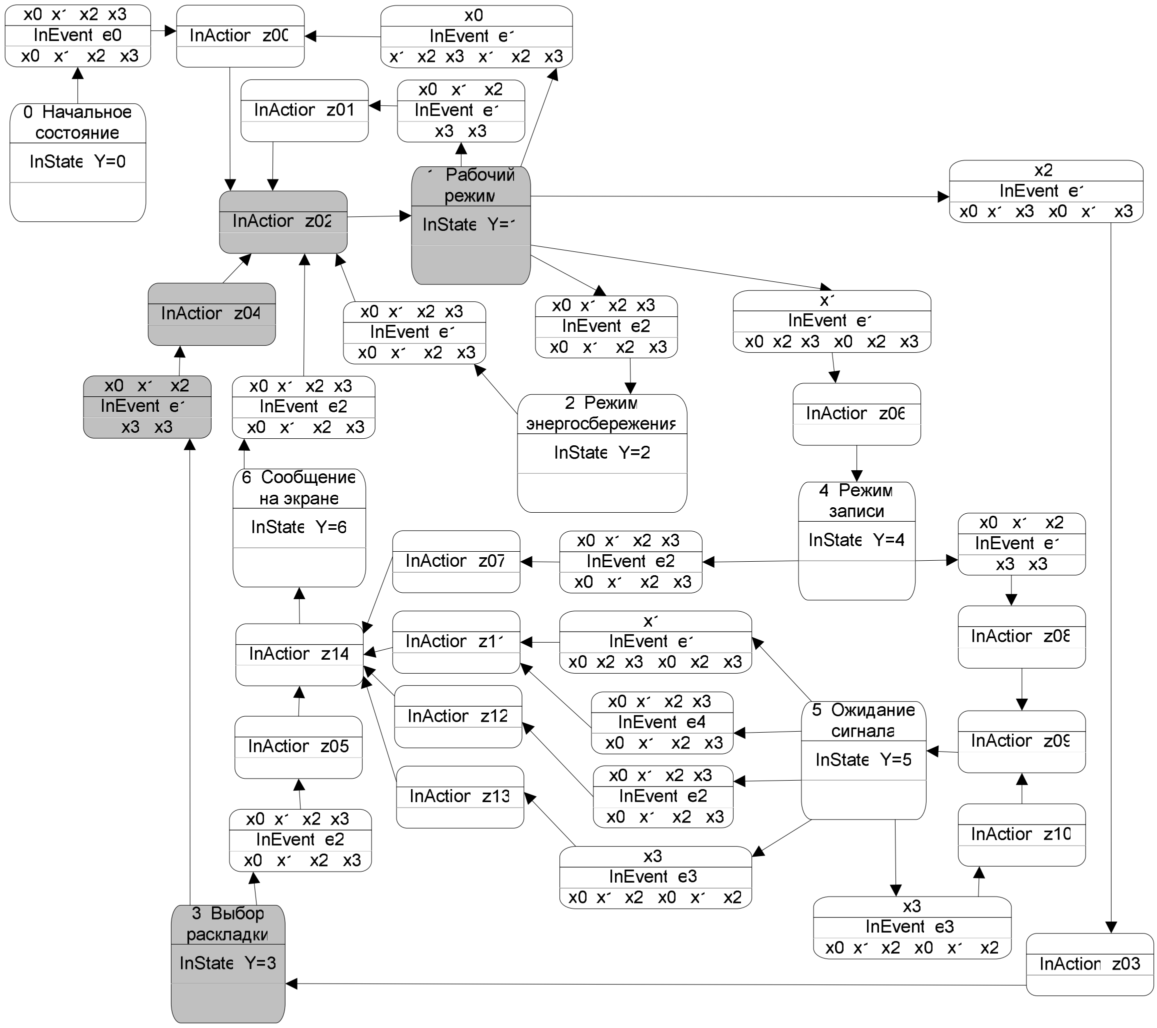
После того, как отработала программа-верификатор, необходимо определить выполнимость формул спецификации на определенных участках автомата. Среди этих участков могут быть состояния, события, выходные воздействия. Будем считать, что недостаточно лишь знать ответ, верна или неверна формула на некотором участке. Требуется, чтобы сценарий, предъявленный программой, был представлен в исходном автомате. Сценарий для любой подформулы спецификации представляет собой бесконечный путь в модели *Крипке*, иллюстрирующий справедливость или ошибочность данной подформулы. Изображать этот путь следует конечным – вспомним утверждение из разд. 3.4.

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

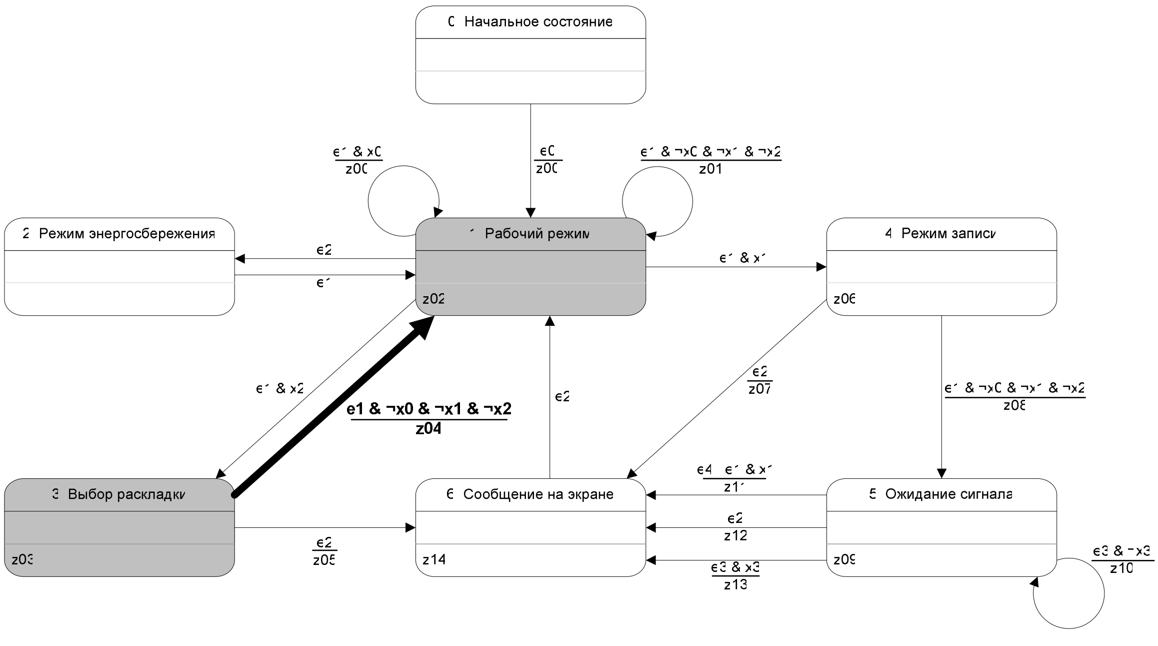
событие ведет по данному пути из исходного состояния, а также значения существенных и список несущественных входных переменных в момент, когда произошло это событие. Эта информация однозначно определяет направление, вдоль которого строится путь в исходном автомате *Мили*. Если же путь (или его участок) 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 $\neg\mathbf{E}[\neg(Y=6)\mathbf{U}(Y=1)]$, которая, как описано в разд. 2.2, трактуется следующим образом: в состояние *1* нельзя попасть, минуя состояние *6*. Эта формула в состоянии *3* не выполняется. Верификатор сгенерировал (кратчайший и единственный в данном случае) контрпример, который на рис. 13,а выделен серым цветом. Это конечный путь, любое продолжение которого удовлетворяет формуле $\mathbf{E}[\neg(Y=6)\mathbf{U}(Y=1)]$.

Этот же путь, но представленный в исходном автомате, можно увидеть на рис. 13,б.



а



б

Рис. 13. Контрпример. Путь в модели *Крипке* (а). Путь в автомате *Мили* (б)

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

На этом процесс верификации автоматных программ завершен.

Заключение

В данной работе были предложены методы для моделирования автомата *Мили* структурами *Крипке.* Также были описаны существующие алгоритмы верификации ветвящейся темпоральной логики применительно к программам с явным выделением состояний. Был разработан алгоритм для построения сценариев и их интерпретации в исходном автомате. В связи с созданием этого алгоритма было доказано утверждение, позволяющее привести все сценарии к общему виду.

Составление сценариев (в том числе, контрпримеров) верифицирующими инструментами 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, скажем, за счет

удаления какого-либо перехода. При этом, разумеется, не гарантируется, что в модели при этом не возникнет других противоречий со спецификацией, хотя, не исключается возможность и более интеллектуальной коррекции.

Исходя из изложенного выше, можно кратко сформулировать основные достоинства автоматных программ в части их верификации [41]:

- 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;
- 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 [42, 43].

С математическим аппаратом, используемым в настоящей работе, на русском языке можно ознакомиться в работе [44].

Источники

1. *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.
2. *Clarke E. M., Grumberg O., Peled D. A.* Model Checking. http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=3730
3. *Clarke E. M, Jr.* Lectures on Model Checking. Computer Science Department, Carnegie Mellon University, 2005.
4. *Соколов В. А., Чалый Д. Ю.* Методы исследования поведения транспортных протоколов в условиях интенсивного сетевого трафика. Ярославль: Ярославский государственный университет, 2004. http://www.teletraffic.ru/public/pdf/Sokolov_Chalyi_2004.pdf
5. *Покозий Е. А.* Методы спецификации и верификации параллельных моделей с непрерывным временем. Автореферат диссертации на соискание ученой степени кандидата физико-математических наук. Институт систем информатики Сибирского отделения РАН. Новосибирск, 1999.
6. *Джексон Д.* Программы проверяют программы //В мире науки. 2006. №10, с.52–57.
7. *Вудкок Дж.* Первые шаги к решению проблемы верификации программ //Открытые системы. 2006. № 8, с.36–57.
8. *Quielle J., Sifakis J.* Specification and verification of concurrent systems in *CESAR* /Proceedings 5-th International Symposium on Programming, LNCS 137, pp. 337–351, 1982.
9. *Clarke E. M. and Emerson E. A.* Synthesis of synchronisation skeletons for branching time logic /Logic of Programs, LNCS 131, pp. 52 – 71, 1981.
10. *Pnueli A.* The temporal logic of programs /Proceedings 18th IEEE Symposium on Foundations of Computer Science, pp. 46 – 57, 1977.

11. *Шалыто А. А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/switch/1
12. *Шалыто А. А.* Логическое управление. Методы аппаратной и программной реализации СПб.: Наука, 2000. http://is.ifmo.ru/books/log_upr/1
13. *Кузьмин Е. В., Соколов В. А.* Моделирование, спецификация и верификация «автоматных» программ. Ярославский государственный университет им. П. Г. Демидова, 2006.
14. *Кормен Т., Лейзерсон Ч., Ривест Р, Штайн К.* Алгоритмы: построение и анализ, 2-е издание. Перевод с английского – М.: Издательский дом «Вильямс», 2005.
15. *Вельдер С. Э., Бедный Ю. Д.* Универсальный инфракрасный пульт для бытовой техники. СПбГУ ИТМО. Курсовая работа. 2005. http://is.ifmo.ru/projects/irrc/
16. *Kripke S. A.* Semantical considerations on modal logic //Acta Philosophica Fennica 16: 83 – 94, 1963.
17. *Clarke E. M., Draghicescu I. A.* Expressibility Results for Linear-Time and Branching-Time Logics. Computer Science Department. Carnegie Mellon University, 2005.
18. *Finite state machine*. http://en.wikipedia.org/wiki/Finite_state_machine
19. *Mealy machine*. http://en.wikipedia.org/wiki/Mealy_machine
20. *Moore machine*. http://en.wikipedia.org/wiki/Moore_machine
21. *Sebastiani R.* Introduction to Formal Methods, 2005–2006. http://dit.unitn.it/~rseba/DIDATTICA/fm2005/02_transition_systems.pdf
22. *Margaria T.* Model Structures. Service Engineering – SS 06. https://www.cs.uni-potsdam.de/sse/teaching/ss06/sveg/ps/2-ServEng-Model-Structures.pdf
23. *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

24. *Hull R.* Web Services Composition: A Story of Models, Automata and Logics. Bell Labs, Lucent Technologies, 2004. http://edbtss04.dia.uniroma3.it/Hull.pdf
25. *Сайт проекта UniMod*. http://unimod.sf.net
26. *Сайт eVelopers Corporation*. http://www.evelopers.com
27. *Шопырин Д. Г., Шалыто А. А.* Синхронное программирование //Информационно-управляющие системы. 2004. № 3, с. 35–42. http://is.ifmo.ru/works/sync_prog/
28. *Von Hanxleden R.* Modeling/Distributed RT Systems – Lecture 03, 2005. http://www.informatik.uni-kiel.de/inf/von-Hanxleden/teaching/ss05/v-rt2/lectures/lecture03-handout4.pdf
29. *Berry G.* The Esterel v5 Language Primer, April, 1999. ftp://ftp-sop.inria.fr/meije/esterel/papers/primer.pdf
30. *Potop D., de Simone R.* Optimizations for Faster Execution of *Esterel* Programs, 2003.
31. *Edwards S. A.* The Synchronous Language *Esterel*. Columbia University, Department of Computer Science, 2002. http://www1.cs.columbia.edu/~sedwards/classes/2002/w4995-02/esterel.pdf
32. *Лифшиц Ю.* Верификация программ и темпоральные логики. Лекция №3 курса «Современные задачи теоретической информатики». СПбГУ ИТМО, 2005. http://logic.pdmi.ras.ru/~yura/modern/03modernnote.pdf
33. *Лифшиц Ю.* Символьная верификация программ. Лекция №4 курса «Современные задачи теоретической информатики». СПбГУ ИТМО, 2005. http://logic.pdmi.ras.ru/~yura/modern/04modernnote.pdf
34. *Кузьмин Е. В., Соколов В. А.* Проверка модели для вполне структурированных систем переходов автоматного типа. Ярославль: Ярославский государственный университет. 2004. http://www.ict.nsc.ru/ws/dicr/7629/rep7629.pdf
35. *Puppis G.* Automata for Branching and Layered Temporal Structures. Thesis. Università degli Studi di Udine, Dipartimento di Matematica e Informatica. Dottorato di Ricerca in Informatica, 2006.

36. *Müller-Olm M., Schmidt D., Steffen B.* Model Checking: A Tutorial Introduction, 1999. http://eti.informatik.uni-dortmund.de/imperia/md/content/publikationen/muss99-it.pdf
37. *Lerda F.* LTL to Büchi Automata. Carnegie, Mellon University, 2005.
38. *Lerda F.* LTL Model Checking. Carnegie, Mellon University, 2005.
39. *Lerda F.* SPIN, An explicit state model checker. Carnegie, Mellon University, 2005.
40. *Biere A., Cimatti A., Clarke E., Zhu Y.* Symbolic model checking without BDDs. School of Computer Science, Carnegie Mellon University, 1999.
41. *Switch*-technology. http://en.wikipedia.org/wiki/Switch-technology
42. *Дейкстра Э.* Заметки по структурному программированию /Дал У., Дейкстра Э., Хоор К. Структурное программирование. М.: Мир, 1975.
43. *Мейер Б.* Объектно-ориентированное конструирование программных систем. М.: Русская редакция. 2005.
44. *Миронов А. М.* Математическая теория программных систем. http://intsys.msu.ru/study/mironov/mthprogsys.pdf