Научно-технический вестник Санкт-Петербургского государственного университета информационных технологий, механики и оптики. Вып. 53. Автоматное программирование. 2008, с. 221–229.

УДК 004.4’242

МЕТОД АВТОМАТИЧЕСКОЙ ДИНАМИЧЕСКОЙ ВЕРИФИКАЦИИ АВТОМАТНЫХ ПРОГРАММ

О. Г. Степанов

В настоящей работе предлагается метод автоматической динамической верификации систем автоматов Мили, основанный на обходе альтернирующих автоматов. Описывается структура метода, правила построения протоколов и спецификации, а также выполняется анализ функциональных характеристик метода на основе разработанной реализации.

Введение

Актуальной является задача верификации программ – проверки их соответствия заданным свойствам [1]. Существует два подхода к верификации – статическая и динамическая [1]. Наиболее распространена статическая верификация на основе метода *Model Checking* – проверки свойств программ на их моделях. При использовании этого метода верифицируемую программу требуется представить в специальной форме – в виде модели Крипке, описывающей возможные изменения вычислительных состояний программы [1]. С этой моделью связана и основная проблема метода *Model Checking* – экспоненциальный рост размера модели Крипке при линейном росте размера программы. Эта проблема получила название «экспоненциальный взрыв».

Другой подход – это динамическая верификация, при использовании которой протоколы выполнения программы проверяются на соответствие заданной спецификации. Эта разновидность верификации применяется для проверки поведения программы во время выполнения, например, при отсутствии доступа к коду программы или при исследовании правильности взаимодействия со сторонними компонентами. Хотя верификация протокола конкретного запуска программы не может гарантировать выполнения заданных свойств при любых значениях входных параметров, при правильном подборе тестовых сценариев ее результаты могут оказаться достаточно точными. При этом трудоемкость динамической верификации не зависит от сложности верифицируемой программы, а только лишь от сложности проверяемой спецификации и размера протоколов.

Для верификации автоматных программ в настоящее время наиболее широко используемым является метод *Model checking* [2]. Однако исследования показали, что текущие реализации этого метода позволяют верифицировать лишь программы с небольшим числом автоматов, так как их число определяет размер состояния программы. Следовательно, размер модели Крипке системы автоматов экспоненциально зависит от числа автоматов в системе [2]. Динамическая верификация автоматных программ в настоящее время практически не изучена.

В настоящей работе предлагается метод динамической верификации автоматных программ, основанный на общем методе динамической верификации [3]. Этот метод основан на обходе альтернирующих автоматов, что позволяет верифицировать протоколы

программ с трудоемкостью, линейно зависящей от размера протокола и числа подформул в спецификации.

Значительным ограничением динамической верификации программ является ненадежность этого подхода к верификации систем параллельных программ, так как последовательность передачи управления между программами может значительно изменяться от запуска к запуску. В данной работе рассматриваются системы автоматов Мили, в которых события обрабатываются по очереди. Это делает метод динамической верификации применимым для таких систем.

В первом разделе описывается существующий метод динамической верификации программ, а во втором – разработанный автором метод динамической верификации автоматных программ. При этом приводится структура метода, описываются схема построения протокола и выразительные возможности спецификации. В третьем разделе излагаются функциональные особенности и характеристики предложенного метода.

Динамическая верификация программ с использованием альтернирующих автоматов

Для методов как статической, так и динамической верификации, в качестве языка спецификации используются языки темпоральной логики. Наиболее распространенными их них являются: язык линейной темпоральной логики (*LTL*) и язык логики ветвящихся вычислений (*CTL*). Одним из наиболее распространенных языков темпоральной логики является *LTL*. Формулы этого языка построены на множестве *атомарных высказываний* (*Prop*) и замкнуты через применение булевых операторов, унарного темпорального оператора **N** («на следующем шаге») и бинарного темпорального оператора **U** («до тех пор, как») [4].

Моделью для *LTL*-формул является *вычисление* — функция $\pi : \omega \to 2^{\mathrm{Pr}op}$, которая задает значения истинности высказываний из множества *Prop* в каждый момент времени, задаваемый натуральным числом. Вычисление π в момент времени $i \in \omega$ удовлетворяет *LTL*-формуле φ (обозначается $\pi, i \rhd \varphi$) при выполнении следующих условий:

- $\pi, i \rhd p$ для $p \in Prop \Leftrightarrow p \in \pi(i)$;
- $\pi, i \rhd \xi \wedge \psi \Leftrightarrow (\pi, i \rhd \xi \wedge \pi, i \rhd \psi)$;
- $\pi, i \rhd \xi \vee \psi \Leftrightarrow (\pi, i \rhd \xi \vee \pi, i \rhd \psi)$;
- $\pi, i \rhd \overline{\varphi} \Leftrightarrow \overline{\pi, i \rhd \varphi}$;
- $\pi, i \rhd N\varphi \Leftrightarrow \pi, i+1 \rhd \varphi$;
- $\pi, i \rhd \xi U \psi \Leftrightarrow \exists j > i : \pi, j \rhd \psi, \forall k : i < k < j : \pi, k \rhd \xi$.

Говорят, что π *удовлетворяет* формуле φ (обозначается $\pi \rhd \varphi$), если и только если $\pi, 1 \rhd \varphi$.

Вычисление π является бесконечным словом на алфавите *Prop*. Таким образом, каждой *LTL*-формуле φ соответствует множество бесконечных слов, удовлетворяющих этой формуле. Это множество называется *языком формулы* φ и обозначается $L(\varphi)$.

Отметим, что для верификации спецификаций, заданных в виде *LTL*-формул на протоколах, требуется расширить определение *LTL* на конечные вычисления, в которых π задано на отрезке [1, *n*]. Это сделано в работе [3].

Для данного множества *X* определим $B^+(X)$ как множество положительных булевых формул над *X* (булевых формул, построенных из элементов *X*, которые соединены операторами $\vee$ и $\wedge$) и формул **истина** и **ложь**.

Альтернирующим автоматом Бюхи называется набор $A = (\Sigma, S, s^0, \rho, F)$ [5], где Σ – непустой конечный алфавит, S – непустое конечное множество состояний, $s^0 \in S$ – начальное состояние, F – множество *принимающих состояний*, а $\rho : S \times \Sigma \to B^+(S)$ – функция перехода.

Запуском альтернирующего автомата Бюхи на бесконечном слове $\omega = \{a_0, a_1, ...\}$ называется S-помеченное дерево r такое, что его корень помечен значением s^0 и справедливо следующее: если вершина дерева x глубиной i помечена значением s и $\rho(s, a_i) = \theta$, то эта вершина имеет k детей $x_1, x_2 \ldots x_k$, где $k \le |S|$, и множество их пометок $\{r(x_1), r(x_2), ...r(x_k)\}$ обращает формулу θ в истину. Запуск называется *принимающим*, если на каждой его ветви принимающие состояния или переход $\rho(s, a_i)$= **истина** встречаются бесконечно часто.

Таким образом, каждому альтернирующему автомату A соответствует множество бесконечных слов, для которых существует принимающий запуск. Это множество называется *языком автомата A* и обозначается *L(A)*.

Можно построить аналог альтернирующего автомата Бюхи для конечных слов. Такой автомат будет называться *альтернирующим автоматом*.

В работе [6] показано, что для любой *LTL*-формулы φ можно построить альтернирующий автомат Бюхи A такой, что $L(A)=L(\varphi)$, причем число состояний автомата A линейно зависит от размера формулы φ.

Таким образом, для верификации соответствия протокола работы программы спецификации, заданной в виде *LTL*-формулы, достаточно по этой спецификации построить альтернирующий автомат и проверить, является ли данный протокол элементом языка полученного автомата. В работе [3] предложены три алгоритма построения принимающего запуска для данных альтернирующего автомата и протокола.

Первый алгоритм (*обход в глубину)* пытается построить принимающий запуск, обходя альтернирующий автомат рекурсивно в глубину из начального состояния. Этот алгоритм наиболее прост в реализации, но часто требует обхода протокола несколько раз. Например, для спецификации вида **GF**φ (где Fφ = **истина U** φ и $G\varphi = \overline{F\overline{\varphi}}$), «хвост» протокола будет повторно проанализирован на каждом шаге. Таким образом, для длинных протоколов алгоритм обхода в глубину работает неприемлемо медленно.

Второй алгоритм (*обход в ширину)* обходит автомат в ширину, поддерживая на каждом шаге все возможные комбинации записей протокола, которые могут являться элементами принимающего запуска в данный момент. Алгоритм обхода в глубину анализирует протокол лишь однажды, но вычислительное состояние алгоритма имеет размер, экспоненциально зависящий от размера спецификации. Для небольших формул множества состояний, допустимых в данный момент, невелики, но с ростом сложности формул размер этих множеств может представлять проблему.

Экспоненциальный рост вычислительного состояния алгоритма вызван недетерминизмом формулы. Эту проблему можно решить, анализируя протокол от «хвоста к голове» [7]. Третий из предложенных в работе [3] алгоритмов, алгоритм обратного обхода, похож на обход в глубину, но вместо рекурсивных вызовов использует уже вычисленное для «хвоста» протокола состояние. На каждом шаге алгоритм поддерживает набор состояний альтернирующего автомата, допускаемых просмотренным «хвостом» протокола.

В следующей части статьи предлагается метод динамической верификации автоматных программ, основанный на описанном подходе. Также проанализирована эффективность работы описанных трех алгоритмов для динамической верификации автоматных программ.

Метод динамической верификации автоматных программ

В настоящий время для верификации программ, заданных в виде системы автоматов Мили, в основном используется статическая верификация – метод *Model Checking*. Однако, существующие реализации этого метода позволяют верифицировать лишь относительно несложные системы автоматов (сотни состояний во всей системе) [2]. В этом разделе предлагается метод динамической верификации систем автоматов Мили. Этот метод включает в себя алгоритм построения протокола выполнения автоматной системы, а также семантику и правила вычисления значений предикатов, используемых при записи спецификаций.

Дадим краткое описание системы автоматов. Имеется набор конечных детерминированных автоматов Мили [8] с несколькими выходными воздействиями на ребрах. Для автомата задается набор событий, с которыми он может вызываться. Для каждого события определяется: может ли это событие поступать от источника событий или только от автомата?

Каждый переход может содержать:

- событие, при котором он происходит;
- условие, при котором он происходит. В условии в качестве атомарных утверждений можно использовать входные переменные $x_1,x_2,\ldots,x_k$ и выражения вида $s_{i,j}$ (оно истинно, если автомат A_i находится в состоянии $s_{i,j}$);
- последовательность действий. Она может содержать выходные воздействия z_i и передачу управления другим автоматам $A_i(e_j)$.

События обрабатываются последовательно – одно событие за один раз. В качестве входных переменных используются булевы переменные или состояния другого автомата. Переменные других типов необходимо преобразовывать в булевы.

На рис. 1 изображен пример простой системы автоматов Мили, которая может использоваться для управления грузовым лифтом. Автомат *A1* моделирует поведения лифта (*s1* – «Ожидание, двери закрыты», *s2* – «Движение», *s3* – «Ожидание, двери открыты»). Автомат *A2* моделирует лампу в кабине лифта (*s1* – «Свет выключен», *s2*– «Свет включен»). Когда автомат *A1* получает событие *e1* («Вызов»), он переходит в состояние *s2,* в котором он ожидает событие *e2* («Прибытие»). При получении этого события в автомате *A1* осуществляется переход в состояние *s3* и вызывается автомат *A2* с событием *e3,* включая тем самым свет. При получении события *e4* («Двери закрыты») автомат *A1* пересылает это событие автомату *A2,* который в ответ выключает свет. Конечно, это довольно простой пример, но он иллюстрирует взаимодействие основных компонентов автоматной системы.

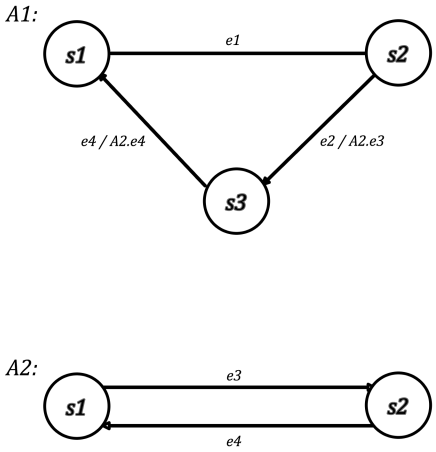

Рис. 1. Диаграммы состояний автоматов системы управления лифтом

Опишем метод динамической верификации такой системы автоматов. Он состоит из следующих простых шагов:

- запись отрицания верифицируемой спецификации в виде *LTL*-формул. Это преобразование производится вручную в соответствии с рекомендациями, изложенными в работе [9];
- автоматическое построение по полученным формулам альтернирующего автомата;
- запуск системы автоматов и автоматическое построение протокола ее работы;
- автоматический обход альтернирующего автомата в соответствии с полученным протоколом с помощью одного из алгоритмов, описанных в предыдущем разделе;
- автоматическое построение контрпримера в модели при обнаружении нарушения спецификации.

Заметим, что все шаги метода, кроме первого, выполняются автоматически, а первый шаг в том или ином виде присутствует во всех подходах к верификации автоматов по методу *Model Checking*. Таким образом, предлагаемый метод не увеличивает объем ручной работы по сравнению с другими существующими автоматическими методами.

В качестве примера приведем верификацию описанной выше автоматной системы управления лифтом. В качестве спецификации выберем условие «через некоторое время после вызова у лифта открываются двери».

Для верификации системы автоматов Мили с использованием метода динамической верификации требуется определить:

- набор атомарных предикатов, которые разрешено использовать в *LTL*-формулах спецификации;
- способ построения протокола выполнения автомата;
- правила вычисления значений атомарных предикатов в записях протокола.

Начнем с описания набора предикатов, на которых можно основывать спецификации. Обработка одного события системой автоматов состоит из следующих шагов [2]:

- получение события;
- вычисление условий на переходах из текущего состояния;
- выполнение действий на переходе. При выполнении вызова другого автомата управление передается этому автомату с соответствующим событием, вызванный автомат совершает переход по этому событию и затем возвращает управление вызвавшему автомату;
- переход в новое состояние.

В соответствии с это схемой необходимо предоставить возможность установить факт совершения того или иного этапа обработки события. Таким образом, достаточным представляется следующий набор базовых предикатов:

- $e_{i,j}$ – автомат A_i обрабатывает событие e_j;
- x_i – при обработке текущего события значение входной переменной x_i истинно;
- z_i – выполняется выходное воздействие z_i;
- $s_{i,j}$ – A_i находится в состоянии $s_{i,j}$.

Предложенный набор атомарных предикатов покрывает описательные возможности, предложенные в работе [2].

Попробуем перевести спецификацию для системы управления лифтом («через некоторое время после вызова у лифта открываются двери») на язык формул, использующих предложенные предикаты. Она будет записана как $e_{1,1} \to \mathrm{F}\, s_{1,3}$.

Далее определим структуру протокола, позволяющую однозначно вычислять значения предложенных атомарных предикатов в каждой записи. Эту структуру будем основывать на разбиении работы системы автоматов на промежуточные состояния, данной в работе [2]. Протоколом является некоторое конечное слово над конечным алфавитом. В предложенном методе алфавитом протокола системы автоматов Мили из n автоматов (по $S_i\Big|_{i=1}^{n}$ состояний в каждом) с m событиями, k входными переменными и l выходными воздействиями является множество со следующими элементами: $e_{1,1},...,e_{n,m}, s_{1,S_1},...,s_{n,S_n}, x_1,...,x_k, \overline{x_1},...,\overline{x_k}, z_1,...,z_t$. В момент начала протоколирования в протокол делаются записи $s_{i,j}$ о текущих состояниях автоматов (эту последовательность записей мы назовем *заголовком протокола*). Протоколирование может быть начато лишь при условии, что ни один автомат не производит обработку события. При получении входного события e_j автоматом A_i в протокол делается запись $e_{i,j}$. Затем происходит вычисление входных переменных и для каждой переменной, значение которой – **истина**, в протокол делается запись x_i, а для каждой со значением **ложь** – $\overline{x_i}$. При выполнении выходного воздействия в протокол делается запись z_i. Наконец, после обработки события автоматом A_i, в протокол делается запись $s_{i,j}$, указывающая новое состояние автомата, даже если произошел переход по петле и состояние автомата при переходе не изменилось.

Остается определить значения описанных предикатов для каждой записи в протоколе. Назовем *секцией обработки события* автоматом A_i последовательность записей $e_{i,j},\ldots,s_k$. *Заголовком секции* назовем начальную подпоследовательность записей длины $1 + k$: $e_{i,j},\ldots,x_k$. Секции могут быть вложены. При этом вложенные друг в друга секции обязательно соответствуют разным автоматам [2]. Тогда предикат $e_{i,j}$ имеет значение **истина** во всех записях секции обработки события, начинающейся с записи $e_{i,j}$. Предикат x_i имеет значение **истина** для всех записей секции, если в заголовке секции встречается запись x_k и **ложь**, если встречается запись $\overline{x_k}$. Отметим, что в каждом заголовке для каждой входной переменной обязана присутствовать ровно одна из этих записей. При вызове вложенного автомата значения x_i, построенные на основании заголовка секции обработки вложенного события, перекрывают предыдущие значения предикатов. Предикат z_i имеет значение **истина** только в записи z_i. Предикат $s_{i,k}$ принимает значение **истина** в записи $s_{i,k}$ и сохраняет это значение во всех записях протокола до следующей записи вида $s_{i,k'}$, не включая саму эту запись. Во всех остальных случаях предикаты имеют значение **ложь**.

Например, при одиночном вызове лифта и прибытии лифта на этаж, рассматриваемая система управления лифтом произведет действия, описываемые следующим протоколом (в квадратные скобки взяты секции обработки событий): $s_{1,1}s_{2,1}[e_{1,1}s_2][e_{1,2}[e_{2,3}s_{2,3}]s_{1,3}]$.

Отметим, что в результате разделения записей заголовка протокола ($e_{i,j},\ldots,x_k$), некоторые моменты времени, на которых производится обход альтернирующего автомата, могут быть представлены несколькими записями протокола. Так как всему заголовку каждой секции соответствует один момент времени, то для определения значений предикатов в этот момент времени требуется наличие информации обо всех записях заголовка. Эту особенность необходимо учитывать при реализации предложенного метода на практике, но, так как заголовок всегда имеет фиксированную длину, то при получении последней записи заголовка можно немедленно выяснить значения всех предикатов, описывающих состояние системы, в момент начала обработки события.

При обнаружении несоответствия записей протокола заданной спецификации требуется построить контрпример: путь в системе автоматов, который приводит к нарушению условий спецификации. При использовании предложенного метода для построения контрпримера достаточно взять заголовки секций протокола от начала до момента обнаружения нарушения спецификации. Эта последовательность записей в совокупности с заголовком протокола и даст путь в автоматной системе, являющийся контрпримером.

Функциональные особенности и характеристики метода

Для проверки возможности практического использования предложенного метода динамической верификации была осуществлена простейшая реализация алгоритмов построения и обхода альтернирующих автоматов. Реализация выполнена на языке *C#* и исполнялась в среде *Microsoft CLR 2.0 SP1* под операционной системой *Microsoft Windows Vista 64*. Код написан без применения специальных оптимизаций, применение которых может дополнительно ускорить скорость работы верификатора.

Для тестирования в качестве примера использовалась две спецификациию. Одна из них – простая спецификация вида $\boldsymbol{G}x1 \rightarrow \boldsymbol{F}z1$: после получения события, при котором значение входной переменной *x1* – **истина**, всегда выполняется выходное воздействие *z1*. Вторая – более сложная: $\boldsymbol{G}x1 \vee (\boldsymbol{F}z1 \wedge \boldsymbol{F}z2)$, соответствует утверждению «при получении

любого события либо значение переменной *x1* – **истина** либо в некоторый момент времени после него будут выполнены выходные воздействия *z1* и *z2*.

Так как алгоритм анализа протокола не зависит от структуры автомата, то для тестирования были искусственно построены протоколы различной длины, удовлетворяющие и не удовлетворяющие спецификации.

Для спецификации φ и протокола π алгоритм обратного обхода имеет трудоемкость $O(|\varphi| * |\pi|)$ и требует $O(|\varphi|)$ единиц памяти. Это делает его наиболее эффективным алгоритмом при наличии всего протокола в целом. Тестовая реализация алгоритма проводила верификацию протокола относительно спецификации вида $\boldsymbol{Gx1 \rightarrow Fz1}$ с производительностью около 200 000 записей в секунду.

При отсутствии протокола в целом выбор алгоритма ограничен обходом автомата в глубину и в ширину. Алгоритм обхода в глубину в худшем случае имеет трудоемкость $O\left(|\varphi| * |\pi|^2\right)$ и требует $O(|\pi|)$ единиц памяти, что в ряде случаев (например, при верификации спецификации вида **GF**φ) делает его не лучшим выбором. На спецификации $\boldsymbol{Gx1 \rightarrow Fz1}$ скорость работы алгоритма составила около 100 000 записей в секунду.

Наконец, обход в ширину позволяет верифицировать протоколы с трудоемкостью $O(|\varphi| * |\pi|)$, не требуя наличия всего протокола целиком. Однако оценка потребления памяти этим алгоритмом в худшем случае составляет $O\left(2^{|\varphi|}\right)$.

В целом полученные результаты подтверждают рекомендации по использованию указанных алгоритмов, приведенные в работе [3]: при возможности рекомендуется использовать алгоритм обратного обхода. При невозможности его применения следует выбирать между обходом в ширину и глубину в зависимости от вида формулы.

Для изучения характеристик метода на практике был построен набор протоколов различной длины – от 10 до 10^6 записей, в которых записи x_i, z_1 и z_2 распределены равномерно. На каждом протоколе был несколько раз запущен алгоритм обратного обхода для каждой из двух рассматриваемых спецификаций. В результате было вычислено среднее время верификации протокола заданной длины. Результаты измерений приведены на рис. 2.

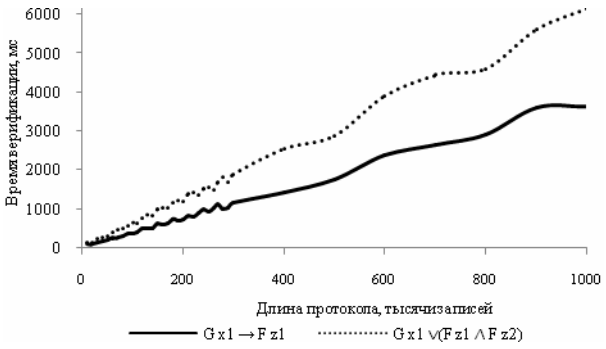

Рис. 2. Скорость работы алгоритма обратного обхода на протоколах различной длины

Важной особенностью динамической верификации программ является зависимость от точки прерывания протокола. Например, если оборвать протокол, используемый в описанном тесте, на записи x_1, то спецификация окажется не выполненной, так как

соответстующая этой записи запись z_1 о выполнении выходного воздействия просто не успела попасть в протокол. В связи с этой особенностью исследователей могут интересовать не столько факт нарушения спецификации, сколько статистика выполнения или не выполнения спецификации. В работе [3] приведены модифицированные алгоритмы обхода, позволяющие собирать такого рода статистику. Предложенный в данной работе метод можно использовать также с этими алгоритмами: последний шаг (построение и анализ контрпримера) следует заменить на анализ полученной статистики.

Заключение

В настоящей работе предложен метод автоматической динамической верификации программ. Предложенный метод позволяет верифицировать гораздо более сложные системы автоматов, нежели традиционный метод *Model Checking*. Предложенный метод основан на известном подходе к верификации программ, в котором используются альтернирующие автоматы. В работе проанализирована применимость этого подхода к верификации автоматных программ; в частности, исследованы особенности использования различных алгоритмов обхода альтернирующего автомата.

Автором произведено сравнение характеристик предложенного метода с рядом методов, описанных в работе [2], на основании чего выработано руководство к выбору того или иного метода верификации в зависимости от поставленной задачи.

Литература

1. Кларк Э., Грамберг О., Пелед Д. Верификация моделей программ: Model Checking. М.: МЦНМО, 2002.
2. Разработка технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода. Теоретические исследования поставленных перед НИР задач. СПбГУ ИТМО. 2007. http://is.ifmo.ru/verification/_2007_02_report-verification.pdf
3. Finkbeiner B., Sipma H. Checking Finite Traces Using Alternating Automata //Form. Methods Syst. Des. 2004. 24, 2.
4. Emerson E. A. Temporal and modal logic / In «Handbook of theoretical Computer Science (Vol. B): Formal Models and Semantics». MA: MIT Press, 1990, pp. 995–1072.
5. Vardi M. Y. Alternating Automata and Program Verification / Computer Science Today. Recent Trends and Developments. Vol. 1000 of LNCS. Springer–Verlag. 1995.
6. Vardi M. Y. Alternating Automata: Checking Truth and Validity for Temporal Logics / Proc. 14th International Conference on Automated Deduction. Vol. 1249 of LNCS. Springer–Verlag. 1997.
7. Havelund K., Roşu G. Testing Linear Temporal Logic Formulae on Finite Execution Traces. Technical Report TR 01–08, RIACS. 2001.
8. Mealy G. A Method to Synthesizing Sequential Circuits // Bell System Technical J. 1955. 34. pp. 1045–1079.
9. Разработка технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода. Выбор направления исследований и базовых компонентов. СПбГУ ИТМО. 2007. http://is.ifmo.ru/verification/_2007_01_report-verification.pdf