Метод разработки и автоматической верификации распределенных автоматных программ

М. А. Лукин,
аспирант
А. А. Шалыто,
доктор техн. наук, профессор
Санкт-Петербургский Национальный Исследовательский Университет Информационных Технологий, Механики и Оптики

Рассмотрен комплексный подход к разработке и верификации распределенных автоматных программ, в которых иерархические автоматы могут реализовываться в разных потоках и взаимодействовать друг с другом. Предложен интерактивный подход к верификации распределенных автоматных программ при помощи инструмента Spin, который включает в себя автоматическое построение модели на языке Promela, приведение LTL-формулы в формат, определяемый инструментом Spin и построение контрпримера в терминах автоматов.

The complex approach to the development and verification of distributed automata-based programs is considered in the paper. In these programs hierarchical state machines can be implemented in separated threads and interact among themselves. The interactive approach to the verification using Spin is proposed. It includes robotized model construction, LTL-formula conversion into Spin format and construction of counter-

Введение

Формальные методы набирают все большую популярность при проверке программного обеспечения. При этом они не конкурируют с традиционным тестированием, а гармонично дополняют его. В данной работе рассматривается верификация методом проверки моделей (model checking) [1 – 3] при помощи верификатора Spin [4]. Метод проверки моделей характеризуется высокой степенью автоматизации [1]. По данной теме проводятся исследования в России и за рубежом [5 – 30]. Настоящая работа является продолжением работ [16, 22, 26].

Описание метода

Описание автоматной модели

В методе используется распределенная система взаимодействующих иерархических конечных автоматов [31 – 33]. При этом каждый иерархический автомат в системе работает в отдельном потоке. Под иерархическим автоматом в данной

работе понимается система вложенных автоматов.

В данной работе каждый граф переходов задает не конкретный автомат, а тип автоматов (по аналогии с типом данных или классом в ООП). Назовем его *автоматным типом*. У каждого автоматного типа может быть несколько экземпляров (по аналогии с объектом в ООП). Назовем эти объекты *автоматными объектами*. Каждый автоматный объект имеет уникальное имя. В дальнейшем, если не указано иное, автоматные объекты будут называться просто автоматами.

Переходы автоматов осуществляются по событиям. Также на переходе могут быть охранные условия [34]. Однако что делать, если встретилось событие, по которому нет перехода? Традиционно в теории языков и вычислений детерминированный конечный автомат в таком случае переходит в недопускающее состояние. Но такое поведение не всегда удобно. Альтернативой переходу в недопускающее состояние может быть игнорирование таких событий, которое реализуется как неявное добавление пустых (без выходных воздействий) петель по всем событиям, переходы по которым не были добавлены пользователем. Таким образом, в предлагаемом методе автомат может работать в одном из двух режимов:

- при появлении события, по которому нет перехода, это событие игнорируется (добавляются пустые петли по всем событиям);
- при появлении события, по которому нет перехода, автомат переходит в недопускающее состояние.

Есть специальное событие «`*`», которое означает переход по любому событию, кроме тех, которые есть на других переходах из этого состояния (аналог `default` в блоках `switch` для C-подобных языков или `else` в условных конструкциях).

Автомат может иметь конечное число переменных целочисленных типов (включая массивы). Для переменных есть следующие модификаторы:

- `volatile` – переменная может быть использована в любом месте программы;
- `external` – переменная может быть использована другим автоматом;
- `param` – переменная является параметром автомата.

По умолчанию считается, что переменная не используется нигде, кроме как на диаграмме переходов автомата.

Все события общие для всей системы автоматов.

Выходные воздействия автомата бывают двух типов:

1. На переходах и в состояниях может быть выполнен любой код. Однако верификатор и генератор кода перенесут его без изменений, поэтому код должен быть допустимым в целевом языке.
2. Запуск на переходах и в состояниях функций, определяемых пользователем на целевом языке программирования (после того, как сгенерирован код).

Автомат может иметь вложенные автоматы любого типа, кроме собственного, иначе будет бесконечная рекурсия. Циклическая рекурсия также запрещена.

Автомат может запускать поток с новым автоматом любого типа. Задается тип автомата `<StateMachine>` и имя `<concreteStateMachine>`. Нельзя запускать несколько автоматов с одним именем. Нельзя запускать автоматы своего типа.

Автомат может взаимодействовать с другим автоматом, выступая источником

событий для него. События отправляются асинхронно.

Автомат может использовать переменные другого автомата, отмеченные специальным модификатором.

Таким образом, в системе могут быть несколько автоматов с одинаковым графом переходов, более того, часть этих автоматов могут быть вложенными, а часть нет.

Все запреты проверяются при помощи верификации.

Описание процесса верификации

Для того чтобы провести верификацию программы методом проверки моделей, требуется составить модель программы и формализовать требуемые свойства (спецификацию) на языке темпоральной логики [1]. Так как в данной работе используется верификатор Spin, то языком темпоральной логики является LTL [1]. В данной работе модель автоматной программы строится автоматически и построение модели описано в разделе «Генерация кода на Promela».

Обозначим автоматный тип через `AType`, автоматный объект через `aObject`. Пусть состояния `AType` называются s_0, s_1 и т. д., в автомат поступают события e_0, e_1 и т. д., а переменные называются x_0, x_1 и т. д., внешние воздействия второго типа z_0, z_1 и т. д. Пусть автоматный тип `AType` имеет вложенный автомат `nested`. Пусть `AType` запускает автомат `fork`.

Процесс верификации состоит из следующих этапов:

1. Генерация кода на языке Promela [4]. В нашем случае она происходит автоматически.
2. Преобразование LTL-формул (переход от нотации автоматной программы в нотацию Spin).
3. Запуск верификатора Spin.
4. Преобразование контрпримера в термины исходной системы автоматов. В нашем случае преобразование происходит автоматически.

Этапы процесса верификации будут описаны ниже. Эти этапы похожи на этапы ручной верификации при помощи Spin. Основным отличием является их максимальная автоматизированность и большая приближенность модели к реализации, чем при верификации неавтоматных программ.

Интерактивность

Одна из главных проблем в верификации методом проверки моделей – это размер модели Крипке. Для того чтобы уменьшить модель (отсечь лишние подробности), мы будем ее строить интерактивно. Для обеспечения интерактивности вводится возможность выбирать, какие уровни абстракции автоматной системы входят в модель, а какие нет. Кроме того, модель структурируется понятным для человека образом для того, чтобы пользователь мог самостоятельно модифицировать построенную модель.

Переменные

Для переменных введем следующие уровни абстракции:

1. Переменные в модели не учитываются.
2. Переменные в модель включены, но модель абстрагируется от их значения. Недетерминированно выбирается, какое охранное условие будет верно.
3. Модель вычисляет значения переменных. При этом переменные могут быть следующих видов:

a. Локальные. Эти переменные могут быть изменены только самим конечным автоматом. Все изменения таких переменных находятся только в выходных воздействиях автомата.
b. Параметры. Извне изменяются только один раз, при запуске автомата. В остальном они подобны локальным.
c. Публичные. Такие переменные могут быть изменены в любом месте программы, в которую входит построенная автоматная система. В модели перед каждым переходом автомата таким переменным недетерминированно присваивается произвольное значение.
d. Совместно используемые. К таким переменным данного автомата имеют доступ другие автоматы, параллельно работающие с данным.

Параметры и публичные переменные могут быть также одновременно и совместно используемыми.

Параллелизм

Вводятся два уровня: параллелизм есть либо нет. Если нет, то в модель не вводятся взаимодействия параллельных автоматов. Остаются только взаимодействия по вложенности.

Источники событий

В качестве источников событий для автоматов в системе могут выступать внешняя среда и другие автоматы. Внешняя среда как источник событий для каждого автомата может работать в одном из трех режимов:

— внешняя среда не взаимодействует с автоматом (события от внешней среды не приходят);
— внешняя среда отправляет только те события, которые автомат может в данный момент обработать;
— внешняя среда отправляет любые события.

Другие автоматы как источники событий можно отключить, если отключить параллелизм.

Процесс верификации

Интерактивность процесса верификации основывается на возможностях верификатора Spin и описана в разделе «Запуск верификатора Spin».

Генерация кода на языке Promela

Все состояния каждого автоматного типа перенумеровываются и для них создаются константы. Для каждого автоматного типа состояния нумеруются отдельно. Имя константы состоит из имени автоматного типа и имени состояния, разделенных знаком подчеркивания. Это сделано для того, чтобы состояния разных автоматов с одинаковыми именами не конфликтовали друг с другом. Пример:

```
#define AType_s0 0
#define AType_s1 1
```

Все события перенумеровываются и для них создаются константы. Для событий применяется сквозная нумерация. Пример:

```
#define e0 1
#define e1 1
```

Все внешние воздействия второго типа (вызываемые функции) перенумеровываются и для них создаются константы аналогично состояниям.

Все вызовы вложенных и запуски параллельных автоматов перенумеровываются аналогично состояниям.

Каждый тип автоматов записывается в inline-функцию, которая моделирует один шаг автомата. Переходы записываются при помощи охранных команд Дейкстры [34]. Для каждого типа автоматов создается структура. Элементы структуры:

- `byte state` – номер текущего состояния;
- `byte curEvent` – номер последнего пришедшего события;
- `byte ID` – номер автомата;
- `byte functionCall` – номер последней запущенной функции, если такая есть;
- `byte nestedMachine` – номер текущего вложенного автомата, если такой есть;
- Все переменные автомата.

Для каждого экземпляра автомата создается экземпляр структуры и *канал*, по которому происходит передача событий.

Для каждого экземпляра автомата, кроме вложенных, создается *процесс*, который извлекает из канала событие и запускает *встраиваемую* (inline) функцию автомата с этим событием.

Для каждого экземпляра автомата, кроме вложенных, создается процесс, который недетерминированно выбирает событие и отправляет его в канал автомата.

Для *публичных переменных* каждый шаг автомата вызывается специальная функция, которая их недетерминированно изменяет.

Для переменных-*параметров* такая функция вызывается один раз – при запуске автомата.

Если по данному событию нет перехода, и в текущем состоянии есть вложенный автомат, то запускается вложенный автомат (запускается встраиваемая функция автомата).

Если в текущем состоянии автомат запускает другой автомат, то запускается заранее созданный процесс запускаемого автомата.

Если автомат отправляет событие другому автомату, то он записывает его номер в канал этого автомата.

Преобразование LTL-формул

Расширим нотацию LTL-формул верификатора Spin. В фигурных скобках будем записывать высказывания в терминах рассматриваемой автоматной модели. Добавим следующие высказывания:

- `aObject.si`, которое означает, что автомат aObject перешел в состояние s_i.
- `aObject.ei`, которое означает, что в автомат `aObject` пришло событие e_i.
- `aObject.zi`, которое означает, что автомат `aObject` вызвал функцию (внешнее воздействие) z_i.
- `aObject->nested`, которое означает, что в автомате `aObject` управление передано вложенному автомату `nested`.
- `aObject || fork`, которое означает, что автомат `aObject` запустил автомат `fork`.
- Бинарные логические операции с переменными автоматов, например, `aObject.x0 >= fork.x0[fork.x1]`.

Пример LTL-формулы в расширенной нотации:

```
[] ({aObject.x0 <= 5} U
{aObject.s1})          (1)
```

Алгоритм преобразования формулы в нотацию Spin следующий:

1. Все высказывания в фигурных скобках перенумеровываются.

2. Каждое такое высказывание преобразовывается в терминах модели на Promela и записывается в макрос.
3. Макросы подставляются в исходную LTL-формулу.

Формула (1) будет преобразована алгоритмом в следующий вид:

```
#define p0 (aObject.x0 <= 5)
#define p1 (aObject.state ==
AType_s1)
ltl f0 {[] (p0 U p1) }
```

Spin поддерживает несколько LTL-формул в одной модели, поэтому формулы нумеруются `f0`, `f1`, и т. д.

Запуск верификатора Spin

Верификация построенной нами модели при помощи инструмента Spin состоит из следующих этапов:

1. Построение верификатора pan. При запуске с ключом `-a` по модели на языке Promela Spin генерирует верификатор pan на языке C.
2. Компиляция верификатора pan. При компиляции можно определить константы, которые влияют на то, как в памяти будет храниться модель Крипке [1]. Наиболее компактный вариант задается константой `BITSTATE`, однако в этом случае происходит аппроксимация, и верификация может быть не точна.
3. Запуск верификатора pan. Верификатор pan также может быть запущен с разными ключами, важнейший из которых является `-a` (поиск допускающих циклов).
4. Анализ контрпримера. Описан в разделе «Преобразование контрпримера».

Интерактивность достигается за счет предоставления пользователю возможности использования вышеперечисленных вариантов работы на этапах верификации.

Преобразование контрпримера

Для того чтобы было удобнее понимать контрпример, приведем метод автоматической трансляции контрпримера, который получается на выходе верификатора Spin, в термины используемой автоматной модели.

Для каждого действия автомата создается пометка при помощи функции `printf`. На языке Promela функция `printf` работает аналогично функции `printf` из языка C [35]. Во время *случайной симуляции* [4] она выводит текст на экран, а во время верификации этот текст появляется в контрпримере. Остается его считать и вывести пользователю. Подробнее преобразование контрпримера описано в работе [26].

Генерация программного кода

Метод разработан для объектно-ориентированных языков, но может быть расширен и для прочих языков. Однако это выходит за рамки данного исследования.

В отличие от таких инструментов, как Unimod [36] и Stateflow [37] в данном подходе предлагается генерировать не самостоятельную программу, а подпрограмму. Для объектно-ориентированных языков это набор классов, который пользователь может включить в свою программу. Для того, чтобы обеспечить удобство использования сгенерированного кода, делаются следующие шаги (ограничение на размер статьи не позволяет подробно описать алгоритмы первичной и повторной генерации кода, отметим лишь, что они используют конечные автоматы и были разработаны при

помощи самого инструментального средства Stater):

- Для каждого автоматного типа генерируется отдельный класс в отдельном файле. Такой класс называется *автоматизированным классом* [33].
- Сгенерированный класс содержит функцию переходов для автомата, перечисление, содержащее события, необходимые переменные для переходов и определения функций (выходных воздействий второго типа), в которые пользователь может дописать собственный код, и этот код не исчезнет при повторной генерации кода.
- В коде специальными комментариями помечаются места, которые полностью переписываются, и в которые не следует писать пользовательский код. Пользовательский код из остальных мест будет полностью сохранен.
- Если пользователь добавит новые выходные воздействия второго типа, то их определения будут добавлены к сгенерированному коду.
- Пользователь может задать *пространство имен* (или *пакет* в языке Java), в котором будет находиться сгенерированный код. Если между генерациями кода пространство имен было удалено, то оно будет восстановлено.
- Если пользователь добавит к автоматизированному классу наследование от базового класса или интерфейса, то повторная генерация кода сохранит это наследование.
- Генерируются вспомогательные классы, включая *менеджер потоков*, которые обеспечивают взаимодействие автоматов, находящихся в разных потоках. Если многопоточность не требуется, их генерацию можно отключить.
- Пользователь может ввести произвольное количество автоматных объектов, которые будут запущены при запуске менеджера потоков.

Хранение диаграмм

При совместной разработке программы несколькими разработчиками существует проблема объединения программного кода, когда один файл редактируется несколькими разработчиками одновременно. Для обычных программ эта проблема решается при помощи систем контроля версий (SVN [38], Git [39], Mercurial [40] и т. д.). Однако системы контроля версий хорошо объединяют только текстовые файлы. Диаграммы графов переходов конечных автоматов у популярных инструментов плохо приспособлены для совместной разработки. Для того чтобы облегчить объединение диаграмм, в данной работе предлагаются следующие свойства, которыми должен обладать формат хранения диаграмм:

1. Формат должен быть текстовым.
2. Каждая диаграмма должна быть в отдельном файле или в отдельном множестве файлов.
3. Структура диаграммы хранится в отдельных файлах от информации, которая нужна для отображения диаграммы.

Первое свойство происходит из факта, что, как уже отмечалось выше, современные системы контроля версий умеют объединять версии только у текстовых файлов. Типичный пример совместной разработки: два программиста одновременно модифицируют файл. Первый программист отправил файл в репозиторий. Второй программист обновляет файл из

репозитория и получает конфликт версии, хранящейся в репозитории, и своей рабочей версии. Существующие системы контроля версий во многих случаях автоматически разрешают подобные конфликты, корректно объединяя две версии файла, а когда не могут их разрешить, предоставляют инструменты помогающие пользователю (в нашем примере второй программист) разрешить такие конфликты версий. Однако это верно только для текстовых файлов. Разрешать конфликты версий в двоичных файлах разработчики должны самостоятельно.

Второе свойство позволяет избежать конфликта версий, когда в автоматной программе модифицируются разные диаграммы. Кроме того, оно позволяет использовать диаграмму повторно.

Третье свойство происходит из того, что изменения диаграммы делятся на два типа: структурные и геометрические. Геометрические изменения затрагивают только внешний вид диаграмм и не влияют на автоматную программу. Таким образом, третье свойство облегчает разрешение конфликтов в структурной части диаграмм.

На основе предложенных свойств и был разработан формат хранения диаграмм.

Описание инструментального средства Stater

Для поддержки предложенного метода было разработано инструментальное средство Stater. Оно позволяет следующее:

- Создавать распределенную систему конечных иерархических автоматов.
- Импортировать конечные автоматы из Stateflow.
- Верифицировать созданную систему конечных автоматов при помощи верификатора Spin.
- Генерировать программный код по созданной системе конечных автоматов.

Инструментальное средство Stater использовалось при разработке самого себя, а именно модулей загрузки диаграмм из файлов и преобразования LTL-формул, а также модуля генерации кода.

Ни предложенный метод, ни разработанное на его основе инструментальное средство Stater не претендуют на полноту верификации систем, разработанных в Stateflow. Это связано, в первую очередь, с тем, что спецификация к Stateflow имеет более 1300 страниц [41].

Пример

Продемонстрируем работу метода на примере прототипа программы управления гусеничным шасси для робота. В шасси два двигателя: по одному на левую гусеницу и на правую. Прототип программы состоит из двух автоматных типов: `AEngine` и `AManager`. Автомат `manager` типа `AManager` Два автомата `left` и `right` типа `AEngine` (рис. 1) управляют соответственно левым и правым двигателями.

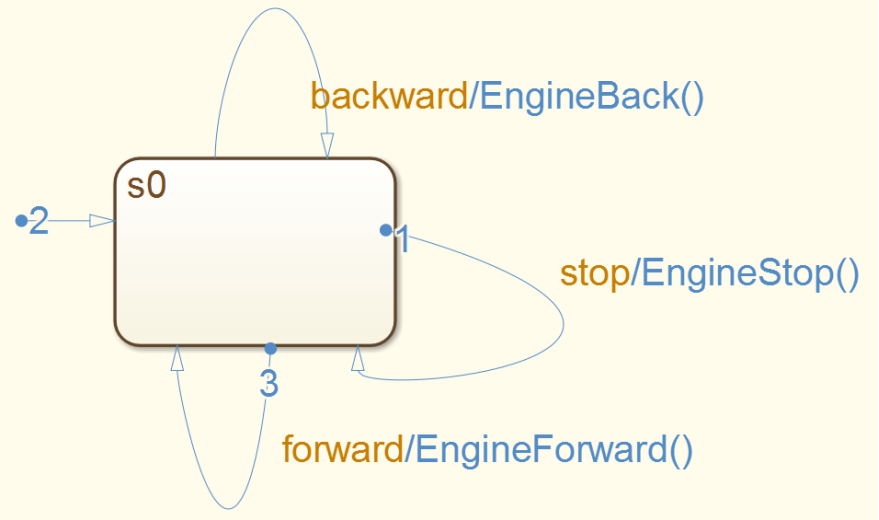

Рис. 1. *Диаграмма переходов автоматного типа AEngine*

Автомат типа `AManager` (рис. 2) отправляет команды на управление двигателями в зависимости от команд для шасси. При входе в состояния он отправляет следующие события автоматам `AEngine` (слева от стрелки написано имя автомата, справа – событие):

- `Stopped: left ← stop, right ← stop.`
- `MoveForward: left ← forward, right ← forward.`
- `MoveBackward: left ← backward, right ← backward.`
- `TurnRight: left ← backward, right ← forward.`
- `TurnLeft: left ← forward, right ← backward.`
- `ForwardRight: left ← stop, right ← forward.`
- `ForwardLeft: left ← forward, right ← stop.`
- `BackwardRight: left ← backward, right ← stop.`
- `BackwardLeft: left ← stop, right ← backward.`

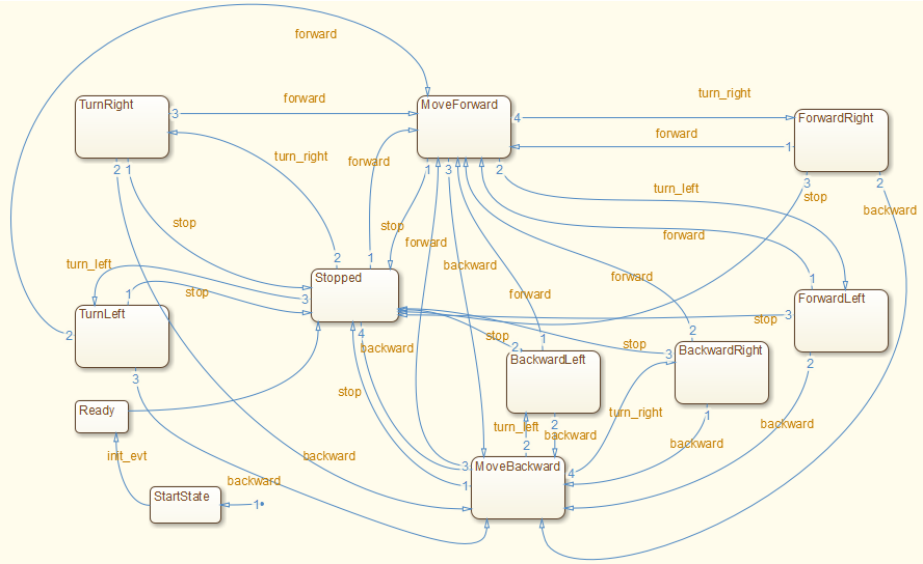

Рис. 2. *Диаграмма переходов автоматного типа AManager*

Проверим свойство: «В любой момент если поступила команда «стоп», то будет подана команда остановки левого двигателя». Мы не можем проверить, что остановился, так как это утверждение относится к аппаратной части. Формализуем это свойство. Высказывание «Поступила команда «стоп» означает, что в автомат manager пришло событие stop. В нотации Stater оно записывается следующим образом: `{manager.stop}`. Высказывание «подана команда остановки левого двигателя» означает, что автомат `left` вызвал функцию `EngineStop`. В нотации Stater оно записывается следующим образом: `{left.EngineStop}`. Поэтому, свойство переписывается так: в любой момент в автомат manager пришло событие stop, следовательно, в будущем автомат `left` вызовет функцию `EngineStop`:

G ({`manager.stop`} => (**F** {`left.EngineStop`})) (2)

В итоге получаем следующий вид:

```
[] ( {manager.stop} -> (<>
  {left.EngineStop} )) (3)
```

Запускаем верификацию и получаем ответ, который означает, что свойство выполняется в построенной системе:

```
0. [] ( {manager.stop} ->
(<> {left.EngineStop} ))
Verification successful!
```

Источники

1. *Кларк Э., Грамберг О., Пелед Д.* Верификация моделей программ: Model Checking. М.: МЦНМО, 2002.
2. *Вельдер С. Э., Лукин М. А., Шалыто А. А., Яминов Б. Р.* Верификация автоматных программ. СПб.: Наука, 2011.
3. *Карпов Ю. Г.* Model Checking: верификация параллельных и распределенных программных систем. СПб.: БХВ-Петербург, 2010.

4. Официальный сайт инструмента Spin. http://spinroot.com
5. *Mikk E., Lakhnech Y., Siegel M. , Holzmann G. J.* “Implementing Stateacharts in Promela/SPIN” in Proc. of WIFT'98, 1998
6. *Latella D., Majzik I., Massink M.* Automatic verification of a behavioral subset UML stetechart diagrams using the SPIN model-checker // Formal Aspects of Computing 11:637–664, 1999.
7. *Lilius, J., Paltor I. P.* Formalising UML State Machines for Model Checking, in: R. B. France and B. Rumpe, editors, Proc. 2nd Int. Conf. UML, Lect. Notes Comp. Sci. 1723 (1999), pp. 430–445.
8. *Eschbah R.* A verification approach for distributed abstract state machines. // PSI '02 109-115, 2001. http://dl.acm.org/citation.cfm?id=705973
9. *Shaffer T., Knapp A., Merz S.* Model checking UML state machines and collaborations. //Electronic notes in theoretical computer science 47:1-13, 2001.
10. *Tiwari A.*. Formal semantics and analysis methods for Simulink Stateow models. Technical report, SRI International, 2002. http://www.csl.sri.com/~tiwari/~stateflow.html
11. *Roux C., Encrenaz E.* CTL May Be Ambiguous when Model Checking Moore Machines. UPMC LIP6 ASIM, CHARME, 2003. http://sed.free.fr/cr/charme2003.ps
12. *Gnesi S., Mazzanti F*. On the fly model checking of communicating UML state machines. 2004. http://fmt.isti.cnr.it/WEBPAPER/onthefly-SERA04.pdf
13. *Gnesi S., Mazzanti F.* A model checking verification environment for UML statecharts / Proceedings of XLIII Congresso Annuale AICA, 2005. http://fmt.isti.cnr.it/~gnesi/matdid/aica.pdf
14. *Виноградов Р. А., Кузьмин Е. В., Соколов В. А.* Верификация автоматных программ средствами CPN/Tools // Моделирование и анализ информационных систем. 2006. № 2, с. 4–15. http://is.ifmo.ru/verification/_cpnverif.pdf
15. *Васильева К. А., Кузьмин Е. В.* Верификация автоматных программ с использованием LTL // Моделирование и анализ информационных систем. 2007. № 1, с. 3–14. http://is.ifmo.ru/verification/_LTL_for_Spin.pdf
16. *Лукин М. А.* Верификация автоматный программ. Бакалаврская работа. СПбГУ ИТМО, 2007. http://is.ifmo.ru/papers/_lukin_bachelor.pdf
17. *Яминов Б. Р.* Автоматизация верификации автоматных UniMod-моделей на основе инструментального средства Bogor. Бакалаврская работа. СПбГУ ИТМО, 2007. http://is.ifmo.ru/papers/_jaminov_bachelor.pdf
18. *Ма G.* Model checking support for CoreASM: model checking distributed abstract state machines using Spin. 2007. http://summit.sfu.ca/item/8056
19. *David A., Moller O., Yi W.* Formal Verification of UML Statecharts with Real-time Extensions. / Formal Methods 2006
20. *Егоров К. В., Шалыто А. А.* Методика верификации автоматных программ // Информационно-управляющие системы. 2008. № 5, с. 15–21. http://is.ifmo.ru/works/_egorov.pdf
21. *Курбацкий Е. А.* Верификация программ, построенных на основе автоматного подхода с использованием программного средства SMV // Научно-технический вестник СПбГУ ИТМО.

Вып. 53. Автоматное программирование. 2008, с. 137–144. http://books.ifmo.ru/ntv/ntv/53/ntv_53.pdf
22. *Лукин М. А., Шалыто А. А.* Верификация автоматных программ с использованием верификатора SPIN // Научно-технический вестник СПбГУ ИТМО. Вып. 53. Автоматное программирование. 2008, с. 145–162. http://books.ifmo.ru/ntv/ntv/53/ntv_53.pdf
23. *Гуров В. С., Яминов Б. Р.* Верификация автоматных программ при помощи верификатора UNIMOD.VERIFIER // Научно-технический вестник СПбГУ ИТМО. Вып. 53. Автоматное программирование. 2008, с. 162–176. http://books.ifmo.ru/ntv/ntv/53/ntv_53.pdf
24. *Егоров К. В., Шалыто А. А.* Разработка верификатора автоматных программ // Научно-технический вестник СПбГУ ИТМО. Вып. 53. Автоматное программирование. 2008, с. 177–188. http://books.ifmo.ru/ntv/ntv/53/ntv_53.pdf
25. *Prashanth C.M., Shet K. C.* Efficient Algorithms for Verification of UML Statechart Models. // Journal of Software. 2009. Issue 3 pp 175-182.
26. *Лукин М. А.* Верификация визуальных автоматных программ с использованием инструментального средства SPIN. СПбГУ ИТМО, 2009. http://is.ifmo.ru/papers/_lukin_master.pdf
27. *Ремизов А. О., Шалыто А. А.* Верификация автоматных программ / Сборник докладов научно-технической конференции «Состояние, проблемы и перспективы создания корабельных информационно-управляющих комплексов. ОАО «Концерн Моринформсистема Агат». М.: 2010, с. 90–98. http://is.ifmo.ru/works/_2010_05_25_verific.pdf
28. Клебанов А. А., Степанов О. Г., Шалыто А. А. Применение шаблонов требований к формальной спецификации и верификации автоматных программ / Труды семинара «Семантика, спецификация и верификация программ: теория и приложения». Казань, 2010, с. 124–130. http://is.ifmo.ru/works/_2010-10-01_klebanov.pdf
29. Вельдер С. Э., Шалыто А. А. Верификация автоматных моделей методом редуцированного графа переходов // Научно-технический вестник СПбГУ ИТМО. 2009. Вып. 6(64), с. 66–77. http://is.ifmo.ru/works/_2010_01_29_velder.pdf
30. *Chen C., Sun J., Liu Y., Dong J., Zheng M.* Formal modeling and validation of Stateflow diagrams // International Journal on Software Tools for Technology Transfer. 2012. Issue 6, pp 653 – 671.
31. *Шалыто А. А.* Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/switch/1
32. *I. Cardei, R. Jha, M. Cardei.* Hierarchical architecture for real-time adaptive resource management. Secaucus, NJ, USA: Springer-Verlag, 2000.
33. *Поликарпова Н. И., Шалыто А. А.* Автоматное программирование. СПб.: Питер, 2010. http://is.ifmo.ru/books/_book.pdf
34. *Dijkstra E.W.* Guarded commands, non-determinacy and formal derivation of programs // CACM. 18. 1975. № 8.
35. Последний черновик спецификации языка C. http://www.open-

std.org/jtc1/sc22/wg14/www/docs/n1570.pdf
36. Официальный сайт проекта UniMod. http://unimod.sf.net
37. Официальный сайт продукта Stateflow. http://www.mathworks.com/products/stateflow/
38. Официальный сайт проекта SVN. http://subversion.apache.org
39. Официальный сайт проекта Git. http://git-scm.com/
40. Официальный сайт проекта Mercurial. http://mercurial.selenic.com/
41. *The MathWorks*. Stateflow and Stateflow coder – User's Guide. 2009.