УДК 004.05

РАЗРАБОТКА И АВТОМАТИЧЕСКАЯ ВЕРИФИКАЦИЯ ПАРАЛЛЕЛЬНЫХ АВТОМАТНЫХ ПРОГРАММ

М. А. Лукин,
программист
А. А. Шалыто,
доктор техн. наук, профессор
Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики

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

Ключевые слова *— автоматы, параллельные автоматные программы, верификация, проверка моделей, линейная темпоральная логика, Spin.*

Введение

Формальные методы все шире используются для повышения качества программного обеспечения. Эти методы не конкурируют с традиционным тестированием, а 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], особенно для автоматных систем, так как сам метод основан на автоматах. По этой теме проводятся исследования в России и за рубежом [5–35]. В настоящей статье, являющейся продолжением работ [17, 24, 29], предлагается комплексный подход к разработке и верификации параллельных автоматных программ. На основе предложенного подхода было разработано инструментальное средство Stater, которое позволяет создавать параллельную систему взаимодействующих иерархических конечных автоматов, импортировать конечные автоматы из инструментального средства Stateflow, которое входит в состав MatLab, верифицировать созданную систему конечных автоматов при помощи верификатора Spin, генерировать программный код по созданной системе конечных автоматов.

Описание подхода

Описание автоматной модели

Предлагаемый подход предназначен для построения параллельных систем взаимодействующих иерархических конечных автоматов [36–38]. При этом каждый такой автомат работает в отдельном потоке. Под иерархическим автоматом 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 объектно-ориентированном программировании). Назовем его *автоматным типом*. У каждого автоматного типа может быть несколько экземпляров (по аналогии с объектом в объектно-ориентированном программировании). 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 [39]. А 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. Однако 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 при появлении события, по которому нет перехода, автомат может работать в одном из двух режимов:

— это событие игнорируется (добавляются пустые петли по всем событиям);

— автомат переходит в недопускающее состояние.

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

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

— `volatile` — переменная может применяться в любом месте программы;

— `external` — переменная может использоваться 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 формируются 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 не обладать этим свойством.

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]. Поскольку в данной работе используется верификатор Spin, то языком темпоральной логики является LTL [1]. Построение модели описано в пункте «Генерация кода на языке Promela». Модель строится для автоматной программы, поэтому построение может быть выполнено автоматически.

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. Построение модели — генерация кода на языке Promela [4]. Для автоматных программ, как отмечено выше, это выполняется 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. Это преобразование автоматных программ также выполняется автоматически.

Эти этапы похожи на этапы ручной верификации при помощи Spin. Основным отличием является бóльший уровень автоматизации и бóльшая приближенность модели к реализации, чем при верификации неавтоматных программ. Ниже описана реализация интерактивности, а затем все четыре этапа верификации.

Интерактивность. Одна из главных проблем при верификации методом проверки моделей — это размер модели Крипке. С целью уменьшить модель (отсечь лишние подробности) будем строить ее 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. Опишем уровни абстракции по разным аспектам верификации: по переменным, параллелизму, источникам событий и по самому процессу верификации.

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 — 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 переменным;

c) публичные — 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 этим автоматом.

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

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

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.

Интерактивность *процесса верификации* основывается на возможностях верификатора Spin и описана в пункте «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. Процесс аналогичен тому, как это делается с состояниями.

Так же аналогично все вызовы вложенных и запуски параллельных автоматов перенумеровываются и для них создаются константы.

Каждый тип автоматов записывается в inline-функцию, которая моделирует один шаг автомата. Переходы записываются при помощи охранных команд Дейкстры [39]. Для каждого типа автоматов создается структура со следующими элементами:

— `byte state` — номер текущего состояния;

— `byte curEvent` — номер последнего пришедшего события;

— `byte ID` — номер автомата;

— `byte functionCall` — номер последней запущенной функции, если такая существует;

— `byte nestedMachine` — номер текущего вложенного автомата, если такой существует;

— 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 на каждом шаге 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 он

запускается (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 номер отправляемого события 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` — автомат `aObject` перешел в состояние s_i;

— `aObject.ei` — в автомат `aObject` пришло событие e_i;

— `aObject.zi` — автомат `aObject` вызвал функцию (внешнее воздействие) z_i;

— `aObject->nested` — в автомате `aObject` управление передано вложенному автомату `nested`;

— `aObject || fork` — 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 языке Promela и записывается в макрос.

3. Макросы подставляются в исходную LTL-формулу.

При использовании этого алгоритма формула (1) преобразуется к виду

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

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

Запуск верификатора Spin. Верификация построенной модели при помощи инструментального средства Spin состоит из следующих этапов.

1. Построение верификатора pan. Pan — это верификатор для конкретной модели с конкретными темпоральными формулами. По сути, Spin является не верификатором, а генератором верификаторов, каждый из которых работает для конкретного частного случая. При запуске с ключом `-a` по модели на языке Promela инструментальное средство 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. Этот верификатор также может быть запущен с разными ключами, важнейшим из которых является `-a` (поиск допускающих циклов).

4. Анализ контрпримера и его преобразование (описано в пункте «Преобразование контрпримера»).

Интерактивность достигается за счет предоставления пользователю возможности использовать вышеперечисленные варианты работы на этапах верификации.

Преобразование контрпримера. Для того чтобы было удобнее понимать контрпример, опишем 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 [40]. 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 [29].

1.1. Генерация программного кода

Подход предполагает использование объектно-ориентированных языков, но может быть расширен и для других языков программирования. Однако это выходит за рамки данной работы.

В отличие от таких инструментов как Unimod [41] и Stateflow [42], 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 (ограниченный объем 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).

1. Для каждого автоматного типа генерируется отдельный класс в отдельном файле. Такой класс называется автоматизированным классом [38].

2. 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.

3. В коде специальными комментариями помечаются фрагменты программы, которые полностью переписываются и в которые не следует писать пользовательский код. Такой код из остальных мест будет полностью сохранен.

4. Если пользователь добавит новые выходные воздействия второго типа, то их определения будут добавлены к сгенерированному коду.

5. 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.

6. Если пользователь добавит к автоматизированному классу наследование от базового класса или интерфейса, то повторная генерация кода сохранит это наследование.

7. Генерируются вспомогательные классы, включая менеджер потоков, которые обеспечивают взаимодействие автоматов, находящихся в разных потоках. Если многопоточность не требуется, их генерацию можно отключить.

8. Пользователь может ввести произвольное число автоматных объектов, которые будут запущены при запуске менеджера потоков.

1.2. Хранение графов переходов

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 [43], Git [44], Mercurial [45] и т. д.). Однако системы контроля версий хорошо объединяют только текстовые файлы. Графы переходов конечных автоматов у популярных инструментов плохо приспособлены для совместной разработки. Для того чтобы облегчить объединение графов переходов, в данной работе предлагаются следующие свойства, которыми должен обладать формат хранения таких графов:

1) формат должен быть текстовым;

2) каждый граф переходов должен быть в отдельном файле или в отдельном множестве файлов;

3) структура графа переходов и информация, которая требуется для отображения, хранятся в разных файлах.

Первое свойство связано с тем, что, как отмечалось выше, современные системы контроля версий умеют объединять версии только текстовых файлов. Типичный пример совместной разработки: два программиста одновременно модифицируют файл. Первый программист отправил файл в репозиторий, а второй программист обновляет файл из репозитория, и получается конфликт версии, хранящейся в репозитории, и 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, которые помогают пользователю (в нашем примере — второй программист) разрешить такие конфликты. 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 графы переходов. Кроме того, оно позволяет использовать графы переходов повторно.

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

На основе предложенных свойств и был разработан формат хранения этих графов.

Описание инструментального средства Stater

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

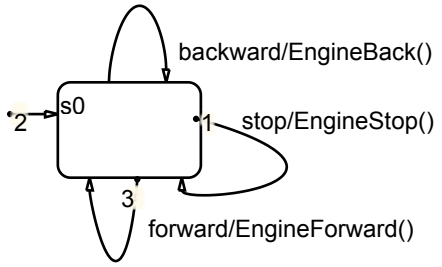
— создавать параллельную систему конечных иерархических автоматов;

— импортировать конечные автоматы из инструментального средства 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 графов переходов из файлов и преобразования LTL-формул, а также модуля генерации кода.



■ ***Рис. 1.*** Граф переходов автоматного типа AEngine

Ни предложенный подход, ни разработанное на его основе инструментальное средство Stater не претендуют на полноту верификации систем, разработанных с помощью Stateflow. Это связано с тем, что спецификация к Stateflow имеет более 1300 страниц [46].

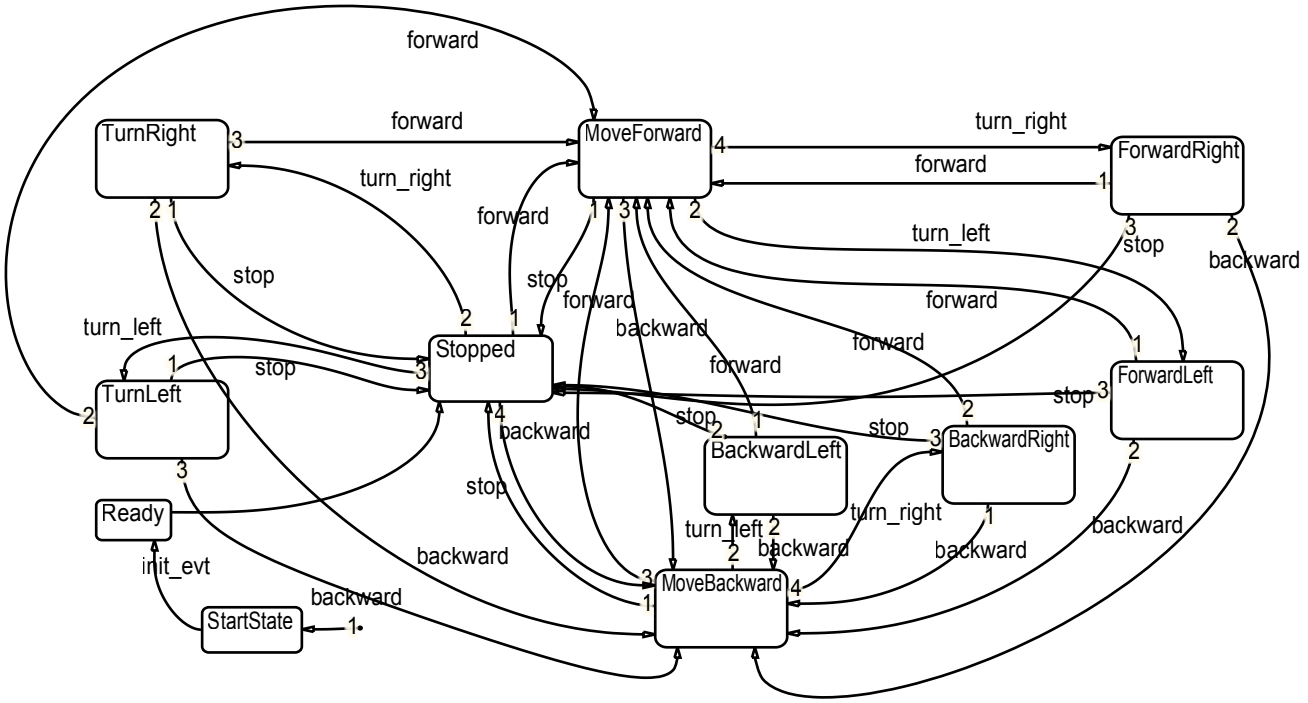
Пример верификации реактивной программы

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

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 ← forward, right ← backward;`
— `TurnLeft: left ← backward, right ← forward;`
— `ForwardRight: left ← forward, right ← stop;`
— `ForwardLeft: left ← stop, right ← forward;`
— `BackwardRight: left ← backward, right ← stop;`
— `BackwardLeft: left ← stop, right ← backward.`

Проверим свойство «В любой момент если поступила команда "стоп", то будет подана команда остановки левого двигателя». Отметим, что нет возможности проверить, что двигатель остановился, так как это утверждение относится к аппаратной части. Формализуем указанное свойство. Высказывание «Поступила команда "стоп"»



■ ***Рис. 2.*** Граф переходов автоматного типа AManager

означает, что в автомат `manager` пришло событие `stop`. В нотации инструментального средства Stater оно записывается следующим образом: `{manager.stop}`. Высказывание «Подана команда остановки левого двигателя» означает, что автомат `left` вызвал функцию `EngineStop`. В нотации средства Stater оно записывается следующим образом: `{left.EngineStop}`. Поэтому рассматриваемое свойство переписывается следующим образом: в любой момент времени в автомат `manager` пришло событие `stop`, следовательно, в будущем автомат `left` вызовет функцию `EngineStop`:

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

Данное свойство не должно выполняться в следующих состояниях: `StartState`, `Ready` и `Stopped`. В первых двух шасси еще не готово к работе, а в состоянии `Stopped` двигатель и так остановлен. В итоге получаем следующую формулу:

```
[] ( ({manager.stop} && !{manager.
StartState} && !{manager.Ready} &&
!{manager.Stopped}) ->
(<> {left.EngineStop} ))                (3)
```

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

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

Заключение

Таким образом, описан комплексный подход к разработке и верификации параллельной системы иерархических конечных автоматов. На основе этого подхода разработано инструментальное средство Stater. Его работа была продемонстрирована на примере прототипа управления гусеничным шасси.

Литература

1. **Кларк Э., Грамберг О., Пелед Д.** Верификация моделей программ: Model Checking. – М.: МЦНМО, 2002. – 416 с.
2. **Вельдер С. Э., Лукин М. А., Шалыто А. А., Ямилов Б. Р.** Верификация автоматных программ. – СПб.: Наука, 2011. – 244 с.
3. **Карпов Ю. Г.** Model Checking: верификация параллельных и распределенных программных систем. – СПб.: БХВ-Петербург, 2010. – 560 с.
4. **Официальный сайт** инструментального средства Spin. http://spinroot.com. (дата обращения: 03.09.2013).
5. **Mikk E., Lakhnech Y., Siegel M., Holzmann G.** Implementing Stateacharts in Promela/SPIN // Proc. of WIFT'98. 1998. http://spinroot.com/gerard/pdf/wift98b.pdf (дата обращения: 03.09.2013).
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. 1999. Vol. 11. P. 637–664.
7. **Lilius J., Paltor I. P.** Formalising UML State Machines for Model Checking // Proc. of 2nd Int. Conf. UML. Lect. Notes Comp. Sci. 1999. P. 430–445.
8. **Eschbah R.** A verification approach for distributed abstract state machines // PSI. 2001. Vol. 2. P. 109–115.
9. **Shaffer T., Knapp A., Merz S.** Model checking UML state machines and collaborations // Electronic Notes in Theoretical Computer Science. 2001. Vol. 7. P. 1–13.
10. **Tiwari A.** Formal semantics and analysis methods for Simulink Stateflow models. Technical report. SRI International, 2002. http://www.csl.sri.com/users/tiwari/html/stateflow.html (дата обращения: 09.09.2013).
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 (дата обращения: 03.09.2013).
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 (дата обращения: 03.09.2013).
13. **Gnesi S., Mazzanti F.** A model checking verification environment for UML statecharts // Proc. of XLIII Congresso Annuale AICA. 2005. http://fmt.isti.cnr.it/~gnesi/matdid/aica.pdf (дата обращения: 20.07.2013).
14. **Канжелев С. Ю., Шалыто А. А.** Автоматическая генерация автоматного кода // Информационно-управляющие системы. 2006. № 6. С. 35–42.
15. **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.
16. **Васильева К. А., Кузьмин Е. В.** Верификация автоматных программ с использованием LTL // Моделирование и анализ информационных систем. 2007. № 1. С. 3–14.
17. **Лукин М. А.** Верификация автоматных программ: бакалаврская работа. СПбГУ ИТМО, 2007. http://

is.ifmo.ru/papers/_lukin_bachelor.pdf (дата обращения: 03.09.2013).
18. **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 (дата обращения: 03.09.2013).
19. **Вельдер С. Э., Шалыто А. А.** О верификации простых автоматных систем на основе метода Model Checking // Информационно-управляющие системы. 2007. № 3. С. 27–38.
20. **Ma G.** Model checking support for CoreASM: model checking distributed abstract state machines using Spin. 2007. http://summit.sfu.ca/item/8056 (дата обращения: 03.09.2013).
21. **David A., Moller O., Yi W.** Formal Verification of UML Statecharts with Real-time Extensions // Formal Methods. 2006. Vol. 3. P. 15–22.
22. **Егоров К. В., Шалыто А. А.** Методика верификации автоматных программ // Информационно-управляющие системы. 2008. № 5. С. 15–21.
23. **Курбацкий Е. А.** Верификация программ, построенных на основе автоматного подхода с использованием программного средства SMV // Научно-технический вестник СПбГУ ИТМО. 2008. № 8. С. 137–144.
24. **Лукин М. А., Шалыто А. А.** Верификация автоматных программ с использованием верификатора SPIN // Научно-технический вестник СПбГУ ИТМО. 2008. № 8. С. 145–162.
25. **Гуров В. С., Яминов Б. Р.** Верификация автоматных программ при помощи верификатора UNIMOD. VERIFIER // Научно-технический вестник СПбГУ ИТМО. 2008. № 8. С. 162–176.
26. **Егоров К. В., Шалыто А. А.** Разработка верификатора автоматных программ // Научно-технический вестник СПбГУ ИТМО. 2008. № 8. С. 177–188.
27. **Гуров В. С., Мазин М. А., Шалыто А. А.** Автоматическое завершение ввода условий в диаграммах состояний // Информационно-управляющие системы. 2008. № 1. С. 24–33.
28. **Prashanth C. M., Shet K. C.** Efficient Algorithms for Verification of UML Statechart Models // J. of Software. 2009. Vol. 3. P. 175–182.
29. **Лукин М. А.** Верификация визуальных автоматных программ с использованием инструментального средства SPIN: магистерская дис. СПбГУ ИТМО, 2009. http://is.ifmo.ru/papers/_lukin_master.pdf (дата обращения: 03.09.2013).
30. **Тимофеев К. И., Астафуров А. А., Шалыто А. А.** Наследование автоматных классов с использованием динамических языков программирования (на примере языка RUBY) // Информационно-управляющие системы. 2009. № 4. С. 21–25.
31. **Ремизов А. О., Шалыто А. А.** Верификация автоматных программ // Состояние, проблемы и перспективы создания корабельных информационно-управляющих комплексов: сб. докл. науч.-техн. конф. – М.: ОАО «Концерн Моринформсистема Агат». 2010. С. 90–98. http://is.ifmo.ru/works/_2010_05_25_verific.pdf (дата обращения: 03.09.2013)
32. **Клебанов А. А., Степанов О. Г., Шалыто А. А.** Применение шаблонов требований к формальной спецификации и верификации автоматных программ // Семантика, спецификация и верификация программ: теория и приложения: тр. семинара. 2010. С. 124–130. http://is.ifmo.ru/works/_2010-10-01_klebanov.pdf (дата обращения: 03.09.2013).
33. **Вельдер С. Э., Шалыто А. А.** Верификация автоматных моделей методом редуцированного графа переходов // Научно-технический вестник СПбГУ ИТМО. 2009. № 6. С. 66–77.
34. **Янкин Ю. Ю., Шалыто А. А.** Автоматное программирование ПЛИС в задачах управления электроприводом // Информационно-управляющие системы. 2011. № 1. С. 50–56.
35. **Chen C.** et al. Formal modeling and validation of Stateflow diagrams // Intern. J. on Software Tools for Technology Transfer. 2012. Vol. 6. P. 653–671.
36. **Шалыто А. А.** Switch-технология. Алгоритмизация и программирование задач логического управления. – СПб.: Наука, 1998. – 617 с.
37. **Cardei I., Jha R., Cardei M.** Hierarchical architecture for real-time adaptive resource management. Secaucus. – NJ, USA: Springer-Verlag, 2000. – 20 p.
38. **Поликарпова Н. И., Шалыто А. А.** Автоматное программирование. – СПб.: Питер, 2010. – 176 с.
39. **Dijkstra E. W.** Guarded commands, non-determinacy and formal derivation of programs // CACM. 1975. Vol. 8. P. 453–457. http://www.cs.virginia.edu/~weimer/615/reading/DijkstraGC.pdf (дата обращения: 03.09.2013).
40. **Последний** черновик спецификации языка С. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf (дата обращения: 03.09.2013).
41. **Официальный** сайт проекта UniMod. http://unimod.sf.net (дата обращения: 03.09.2013).
42. **Официальный** сайт продукта Stateflow. http://www.mathworks.com/products/stateflow/ (дата обращения: 03.09.2013).
43. **Официальный** сайт проекта SVN. http://subversion.apache.org (дата обращения: 03.09.2013).
44. **Официальный** сайт проекта Git. http://git-scm.com (дата обращения: 03.09.2013).
45. **Официальный** сайт проекта Mercurial. http://mercurial.selenic.com (дата обращения: 03.09.2013).
46. **The MathWorks**. Stateflow and Stateflow coder – User's Guide. http://www.mathworks.com/help/releases/R13sp2/pdf_doc/stateflow/sf_ug.pdf (дата обращения: 09.09.2013).