Санкт-Петербургский национальный исследовательский университет
информационных технологий, механики и оптики

На правах рукописи

Лукин Михаил Андреевич

Верификация автоматных программ

05.13.11 – Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

Диссертация на соискание ученой степени кандидата
технических наук

Научный руководитель –
доктор технических наук,
профессор В. Г. Парфенов

Санкт-Петербург – 2014

Оглавление

Введение

Актуальность

Автоматное программирование, основанное на применении конечных автоматов, всё шире используется при разработке программного обеспечения (ПО). В частности, оно применяется для построения программ управления **ответственными объектами**, для которых особую важность имеет правильность их работы. Обеспечение качества ПО достигается за счёт использования соответствующих методов проектирования программ, а также тестирования и верификации. Настоящая работа посвящена вопросу верификации автоматных программ на основе метода проверки моделей (*model checking*) [1 – 5], который может быть автоматизирован.

По данной теме проводятся исследования в России и за рубежом [6 – 40]. Настоящая работа является продолжением работ [20, 28, 32].

Цель диссертационной работы

Цель работы – разработка метода верификации автоматных программ. Для достижения этой цели в диссертации решаются следующие задачи:

- выбор математических моделей управляющих автоматов;
- разработка алгоритма построения *Spin*-модели управляющего автомата;
- разработка алгоритма построения *Spin*-модели системы вложенных управляющих автоматов (иерархического автомата);
- разработка алгоритма построения *Spin*-модели параллельно работающих иерархических автоматов;
- разработка метода верификации автоматных программ: построение *Spin*-модели, преобразование расширенных LTL-формул в *Spin*-формулы, преобразование контрпримера в путь в системе автоматов;
- разработка подхода для верификации в случаях, когда модель внешней среды нельзя представить в виде автомата;
- разработка инструментального средства, поддерживающего предлагаемый метод.

Методы исследования

В работе использованы методы дискретной математики, построения и анализа алгоритмов, теории автоматов, теории верификации, объектно-ориентированного программирования.

Научная новизна

В работе получены следующие результаты, которые выносятся на защиту:

- обоснован выбор математических моделей управляющих автоматов;
- разработан алгоритм построения *Spin*-модели управляющего автомата;
- разработан алгоритм построения *Spin*-модели системы вложенных управляющих автоматов (иерархического автомата);
- разработан алгоритм построения *Spin*-модели параллельно работающих иерархических автоматов;
- разработан метод верификации автоматных программ: построение *Spin*-модели, преобразование расширенных LTL-формул в *Spin*-формулы, преобразование контрпримера в путь в системе автоматов;
- разработано инструментальное средство, поддерживающее предлагаемый метод;
- разработан подход для верификации в случаях, когда модель внешней среды нельзя представить в виде автомата;
- результаты работы внедрены в практику проектирования для сред с конечным и бесконечным числом состояний.

Достоверность

Достоверность научных положений, выводов и практических рекомендаций, полученных в диссертации, подтверждается корректным обоснованием постановок задач, точной формулировкой критериев, компьютерным моделированием, а также результатами использования методов, предложенных в диссертации, на практике.

Практическое значение

Практическое значение работы состоит в том, что предложенный метод обеспечивает возможность верификации программ, базирующихся на разных типах автоматов, используемых в промышленности.

На основе этого метода автором разработано инструментальное средство *Stater*, которое позволяет графически строить иерархические автоматы [41, 42, 44, 45], импортировать их из сред для проектирования автоматных программ, генерировать по этим автоматам программный код и проводить верификацию автоматных программ.

Экспериментальные исследования

На примере задачи поиска пути с *O(1)* памяти на бесконечном клетчатом поле были выполнены экспериментальные исследования указанного выше инструментального средства. Исследования проводились на 800 автоматных программах, созданных при помощи генетического программирования. Удалось доказать корректность работы для 231 программы (вопрос о корректности остальных программ остаётся открытым).

Использование и внедрение результатов работы

Результаты работы использованы в ходе выполнения государственного контракта по теме «Разработка технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода» (отчёт по I этапу «Выбор направления исследований и базовых методов» (2007, http://is.ifmo.ru/verification/_2007_01_report-verification.pdf), по II этапу «Теоретические исследования поставленных перед НИР задач» (2007, http://is.ifmo.ru/verification/_2007_02_report-verification.pdf), по III этапу «Экспериментальные исследования поставленных перед НИР задач» (2007, http://is.ifmo.ru/verification/_2007_03_report-verification.pdf), по IV этапу «Обобщение и оценка результатов исследований» (2007, http://is.ifmo.ru/verification/_2007_04_report-verification.pdf). По результатам

исследований при участии автора написана монография, опубликованная в издательстве «Наука» в 2011 году.

Результаты использованы также при проведении занятий по курсу «Верификация программного обеспечения» для магистрантов кафедры «Компьютерные технологии» в 2012/2013 и 2013/2014 учебных годах, что подтверждается соответствующим актом. Для поддержки этого курса в 2012 году в НИУ ИТМО было опубликовано учебное пособие [46]. Акт использования имеется.

Кроме того, по курсу при участии автора был создан учебно-методический комплекс, опубликованный в системе дистанционного обучения НИУ ИТМО. Там же опубликован его перевод на английский язык, выполненный автором [47]. Акт использования имеется.

Результаты работы внедрены в 2012 году в ООО «Специальный технологический центр» (Санкт-Петербург) при разработке редактора описания антенных систем, что подтверждается актом внедрения.

Результаты также внедрены в 2012 году в студии «PointOmega Games» (Санкт-Петербург) при разработке игры *Turtleball HD* (https://play.google.com/store/apps/details?id=com.pointomega.turtleball), что подтверждается актом внедрения.

Апробация диссертации

Основные положения диссертационной работы докладывались на следующих научных и научно-практических конференциях: Международная научно-методическая конференция «Высокие интеллектуальные технологии и инновации в образовании и науке» (СПбГПУ, 2008), Конференция молодых ученых ИТМО 2009, Всероссийская научная конференция по проблемам информатики СПИСОК (Матмех СПбГУ, 2011), Международная научно-практическая конференция Tools & Methods of Program Analysis (Костромской государственный технический университет, TMPA-2013), Haifa Verification Conference (Haifa, 2014).

Публикации

По результатам диссертации лично автором и в соавторстве опубликовано 11 научных работ, среди которых монография (издательство «Наука»), учебное пособие Университета ИТМО (на русском и английском языках), три статьи (все в изданиях из перечня ВАК).

Кроме того, автором получено свидетельство о регистрации программ на описываемое ниже инструментальное средство *Converter*.

Структура диссертации

Диссертация изложена на 185 страницах и состоит из введения, пяти глав, заключения и двух приложений. Список литературы состоит из 87 наименований. Работа иллюстрирована 34 рисунками и содержит четыре таблицы.

В первой главе приведён обзор существующих решений. Во второй главе приведены основные понятия. В третьей главе описан предлагаемый метод для разработки и верификации автоматных программ, а также описание разработанных автором инструментальных средств. Четвёртая глава содержит описание внедрения полученных результатов на практике. В пятой главе приведено описание подхода к верификации в случаях, когда модель внешней среды нельзя представить в виде автомата, например, для клетчатого поля неограниченных размеров. Подход продемонстрирован на примере доказательства корректности автоматных программ, решающих задачу о поиске пути с $O(1)$ памяти.

Глава 1. Основные понятия

В теории автоматов рассматриваются два класса автоматов: абстрактные и структурные [42]. Сначала рассмотрим абстрактные автоматы.

1.1. Конечный автомат

Определение. *Детерминированный конечный автомат* (ДКА) [48] – это пятерка $<Q, \Sigma, \delta, s, T>$, где

- Q – конечное множество состояний;
- Σ – конечное множество входных воздействий;
- $\delta: Q \times \Sigma \to Q$ – *функция переходов*. Получает на вход текущее состояние и входной символ, на выходе выдает следующее состояние;
- $s \in Q$ – *начальное состояние*;
- $T \subset Q$ – множество *допускающих состояний*.

Приведем теперь неформальное определение. Детерминированный конечный автомат состоит из бесконечной ленты с символами, считывающей головки и состояний с переходами. Набор символов на ленте называется *алфавитом* ДКА, а последовательность символов – *словом*. ДКА либо *допускает* слово, либо *не допускает*. Каждый переход помечается одним из символов алфавита. Считывающая головка движется по ленте только в одну сторону. ДКА стартует из начального состояния. Каждую итерацию ДКА делает следующее:

1. Считывает очередной символ с ленты.
2. Если из текущего состояния существует переход, помеченный считанным символом, то ДКА переходит по нему.
3. Если перехода нет, то ДКА не допускает слово на ленте (переходит в *недопускающее состояние*) и завершает работу.
4. Если ДКА переходит в допускающее состояние, то он допускает слово.

1.2. Автоматы Мили и Мура

Автоматы Мили [49] и *Мура* [50] генерируют выходную последовательность символов. Автомат Мили генерирует её на переходах, а

автомат Мура – в состояниях. Поэтому говорят, что выходные воздействия *автомата Мили* зависят от состояний и входных воздействий, а выходные воздействия *автомата Мура* зависят только от состояний.

Определение. *Автомат Мили* – это шестерка $<Q, s0, X, Y, f, g>$, где:

- Q – конечное множество состояний;
- $s0 \in Q$ – начальное состояние;
- X – конечное множество входных воздействий;
- Y – конечное множество выходных воздействий;
- f: $Q \times X \rightarrow Q$ – функция переходов;
- g: $Q \times X \rightarrow Y$ – функция генерации выходных воздействий.

Определение. *Автомат Мура* – это шестерка $<Q, s0, X, Y, f, g>$, где:

- Q – конечное множество состояний;
- $s0 \in Q$ – начальное состояние;
- X – конечное множество входных воздействий;
- Y – конечное множество выходных воздействий;
- f: $Q \times X \rightarrow Q$ – функция переходов;
- g: $Q \rightarrow Y$ – функция генерации выходных воздействий.

Любой *автомат Мили* может быть преобразован в эквивалентный ему *автомат Мура* и наоборот.

1.3. Структурные автоматы

В состояния автоматов могут быть вложены другие автоматы. Переход во вложенном автомате осуществляется, только если в основном автомате нет перехода по текущему символу. Система вложенных автоматов называется *иерархическим автоматом* [41, 42, 44, 45].

1.4. Верификация

По стандарту PMBOK guide, включенному в систему стандартов IEEE, Верификация – это ответ на вопрос «Соответствует ли продукт спецификации?». Это проверяется формально.

Существуют следующие разновидности верификации:

- формальная верификация;
- на основе теории зависимых типов (специальные языки программирования, такие как *Agda*, *Coq*);
- статический и динамический анализ кода;
- проверка моделей.

Формальная верификация [56] предполагает наличие средств для составления спецификаций и правил вывода. Пользуясь этими правилами вывода, можно математически доказать, что программа соответствует спецификации. Однако это достаточно сложная и трудоёмкая задача, и поэтому формальная верификация мало распространена.

На формальную верификацию похож метод верификации на основе теории **зависимых типов** [57]. Этот метод основан на том, что в специальных языках программирования (таких как, *Coq* и *Agda*) можно одновременно и писать программы, и доказывать теоремы. Поэтому, теорема о том, что программа корректна и доказательство этой теоремы являются составными частями таких программ.

Под анализом кода автор понимает анализ, который проводится при помощи специальных инструментальных средств, например, *Valgrind* [58], *Coverity* [59], *PVS Studio* [60].

Анализ кода делится на статический и динамический. **Динамический анализ –** это анализ, выполняемый во время исполнения программы [61]. Он обладает теми же недостатками, что и тестирование: не проверяются все возможные пути исполнения программы. Кроме того, динамический анализ замедляет исследуемую программу (так как выполняется в ходе её выполнения) и, как следствие, непригоден либо ограниченно пригоден для проверки высоконагруженных систем.

Статический анализ – это анализ кода без его выполнения [62]. Он не обладает указанным выше недостатком, но методами статического анализа можно найти только некоторые классы ошибок.

Метод проверки моделей [1 – 5] состоит в том, что для проверяемой программы или алгоритма строится модель (обычно это *модель Крипке*), спецификация к программе записывается на языке темпоральной логики, а затем специальные алгоритмы автоматически проверяют, соответствует ли модель спецификации. Если модель не соответствует спецификации, то автоматически строится контрпример.

Метод проверки моделей тесно связан со статическим анализом кода. Некоторые инструменты, например, *Java Pathfinder* [63], совместно используют методы проверки моделей и статического анализа кода.

Главным недостатком метода проверки моделей является то, что модель требуется построить вручную. При этом невозможно гарантировать, что если модель удовлетворяет спецификации, то и исходная программа соответствует этой спецификации.

И хотя предпринимаются попытки автоматизировать методы проверки моделей для программ общего вида (например, *Java Pathfinder*), но все они сводятся к тому, что ищутся ошибки из заранее заданных классов, либо проводится ограниченная проверка моделей [52] (как, например, в инструментальном средстве *CBMC* [64]).

Среди парадигм программирования всё чаще начинает использоваться автоматное программирование. Его наиболее целесообразно применять для ответственных систем. Именно для автоматных программ удаётся повысить уровень автоматизации процесса верификации за счёт формальных переходов от программы к модели и от модели к программе при наличии ошибки. В настоящей работе созданы инструментальные средства для верификации автоматных программ.

В ходе выполнения работы по государственному контракту [65 – 69] был выполнен обзор существующих инструментальных средств для проведения верификации [69], который показал, что для автоматных программ рассматриваемого в диссертации типа (сложные условия на переходах автоматов, выходные воздействия и т. д.) – структурных автоматов [42] эффективные решения отсутствуют. Настоящая работа призвана восполнить указанный пробел.

1.5. Темпоральная логика

Темпоральная логика [1, 4, 5] состоит из пропозициональной логики и *темпоральных операторов*. Приведём определение *линейной темпоральной логики* (LTL), которая используется в настоящей работе.

Пусть AP – множество атомарных предложений. Тогда:

- p является формулой для всех $p \in AP$.
- Если φ – формула, то $\neg\varphi$ – формула.
- Если φ и ψ – формулы, то $\varphi \vee \psi$ – формула.
- Если φ – формула, то $\mathbf{X}\ \varphi$ – формула.
- Если φ и ψ – формулы, то $\varphi\ \mathbf{U}\ \psi$ – формула.

Множество формул, построенных в соответствии с этими правилами, называется *формулами LTL*. **X** (ne**X**t, «в следующем состоянии») и **U** (**U**ntil, «пока не») называются *темпоральными операторами*.

Синтаксис LTL может быть задан и в нотации Бэкуса-Наура. Для $p \in AP$ множество LTL-формул определяется следующим образом:

$$\varphi ::= p \mid \neg\varphi \mid (\varphi \vee \varphi) \mid \mathbf{X}\ \varphi \mid (\varphi\ \mathbf{U}\ \varphi).$$

Темпоральные операторы **G** (**G**lobally, «всегда») и **F** (**F**uture, «когда-нибудь») определяются следующим образом:

$\mathbf{F}\ \varphi = \text{true}\ \mathbf{U}\ \varphi$,

$\mathbf{G}\ \varphi = \neg\mathbf{F}\ \neg\varphi$.

Приведём неформальное определение семантики LTL-формул:

- φ **U** ψ означает, что свойство ψ должно выполниться в будущем, а до тех пор, пока оно не выполнилось, должно выполняться φ;
- **X** φ означает, что в следующем состоянии должно выполниться φ;
- **F** φ означает, что φ должно выполниться когда-нибудь в будущем;
- **G** φ означает, что φ должно выполниться в текущем состоянии и во всех следующих.

1.6. Модель Крипке

Введём понятие *множество атомарных предложений* (*элементарных высказываний*) – предложений, внутренняя структура которых не может изменяться. Атомарные предложения (*AP*) – это базовые предложения, которые могут быть сделаны. Примерами атомарных предложений являются предложения «*x* больше 0» или «*x* равно 1» для некоторой переменной *x*. Атомарные предложения определяются над множеством переменных *x*, *y*, …, констант 0, 1, 2, …, функций *max*, *gcd*, … и предикатов таких как, например, $x = 2$, $x \ mod \ 2 = 0$. Также допускаются предложения вида $max(x, y) \leq 3$ или $x = y$.

Выбор множества атомарных предложений *AP* важен, так как он фиксирует базовые свойства, которые могут быть сформулированы для исследуемой программы. Поэтому фиксация множества атомарных предложений может быть названа *первой ступенью абстракции*. Если, например, не позволить множеству *AP* ссылаться на некоторые переменные программы, то ни одно свойство, ссылающееся на эти переменные, не может быть сформулировано, и, как следствие, ни одно такое свойство не может быть проверено.

Моделью Крипке (*структурой Крипке*) [1] над множеством атомарных предложений *AP* называется тройка (*S*, *R*, *Label*), где

- *S* – непустое множество *состояний*;
- $R \subseteq S \times S$ – *тотальное отношение переходов* на S, которое сопоставляет элементу $s \in S$ его возможных потомков;

- *Label*: $S \to 2^{AP}$ сопоставляет каждому состоянию $s \in S$ атомарные предложения *Label(s),* которые верны в *s*.

Отношение $R \subseteq S \times S$ называется *тотальным*, если оно ставит в соответствие каждому состоянию $s \in S$ как минимум одного потомка ($\forall s \in S$: $\exists s' \in S$: $(s, s') \in R$).

Иногда требуют, чтобы для *модели Крипке* был задан набор *начальных состояний* $S_0 \subseteq S$.

Путь в *модели Крипке* из состояния s_0 – это бесконечная последовательность состояний $\pi = s_0 s_1 s_2 \ldots$ такая, что для всех $i \geq 0$ выполняется $R(s_i, s_{i+1})$.

Модель Крипке может обеспечить различную степень детализации переходов. Переходы в *модели Крипке* должны моделировать *атомарные* переходы исходной программы.

При этом нельзя допускать ситуации, когда *модель Крипке* содержит состояния, которые не наблюдаются в верифицируемой программе. В этом случае может оказаться, что при верификации будут найдены не существующие ошибки.

Для пояснения приведем пример из книги [1]. Рассмотрим программу с двумя переменными x, y и двумя переходами α и β, которые могут выполняться параллельно:

α: $x := x + y$,

β: $y := y + x$.

В начальном состоянии $x = 1 \wedge y = 2$. Рассмотрим более детальную реализацию переходов α и β, в которой используются команды ассемблера:

α_0: load R_1, x $\quad$ β_0: load R_2, y

α_1: add R_1, y $\quad$ β_1: add R_2, x

α_2: store R_1, x $\quad$ β_2: store R_2, y

Если в программе выполняется сначала переход α, а затем переход β, то она попадает в состояние $x = 3 \wedge y = 5$. Если переходы выполняются в обратном порядке, то программа попадает в состояние $x = 4 \wedge y = 3$. Если же переходы совмещаются в следующем порядке: α_0, β_0, α_1, β_1, α_2, β_2, то в результате она попадет в состояние $x = 3 \wedge y = 3$.

Пусть состояние $x = 3 \wedge y = 3$ является ошибочным для программы, и требуется проверить, может ли система оказаться в этом состоянии. Допустим также, что система реализована так, что переходы α и β выполняются атомарно. В этом случае система не может оказаться в ошибочном состоянии. Однако если построить *модель Крипке* с более детальными переходами, то в процессе верификации будет найдена ошибка, которой не существует в исходной программе.

Пусть теперь, программа реализована детально, и возможно выполнение последовательности $\alpha_0, \beta_0, \alpha_1, \beta_1, \alpha_2, \beta_2$. Тогда программа неверна, поскольку может попасть в ошибочное состояние. Однако если построить *модель Крипке* с использованием лишь переходов α и β, то верификация покажет, что программа верна, но это будет неправильным результатом.

1.7. Автомат Бюхи

Пусть AP – множество атомарных предложений. *Автоматом Бюхи* [1, 4] над алфавитом 2^{AP} называется четверка $A = (Q, q_0, \delta, F)$, в которой

- Q – конечное множество *состояний*;
- q_0 – начальное состояние;
- $\delta \subseteq Q \times 2^{AP} \times Q$ – тотальное отношение переходов;
- $F \subseteq Q$ – множество *допускающих* состояний.

Автомат Бюхи A допускает слово тогда и только тогда, когда хотя бы одно из состояний множества F встречается бесконечно часто. *Автоматы Бюхи* используются в алгоритмах верификации методом проверки моделей.

1.8. Автоматные программы

Парадигма автоматного программирования предлагает мыслить о программе как о системе взаимодействующих автоматов и объектов управления (рисунок 1.8.1).

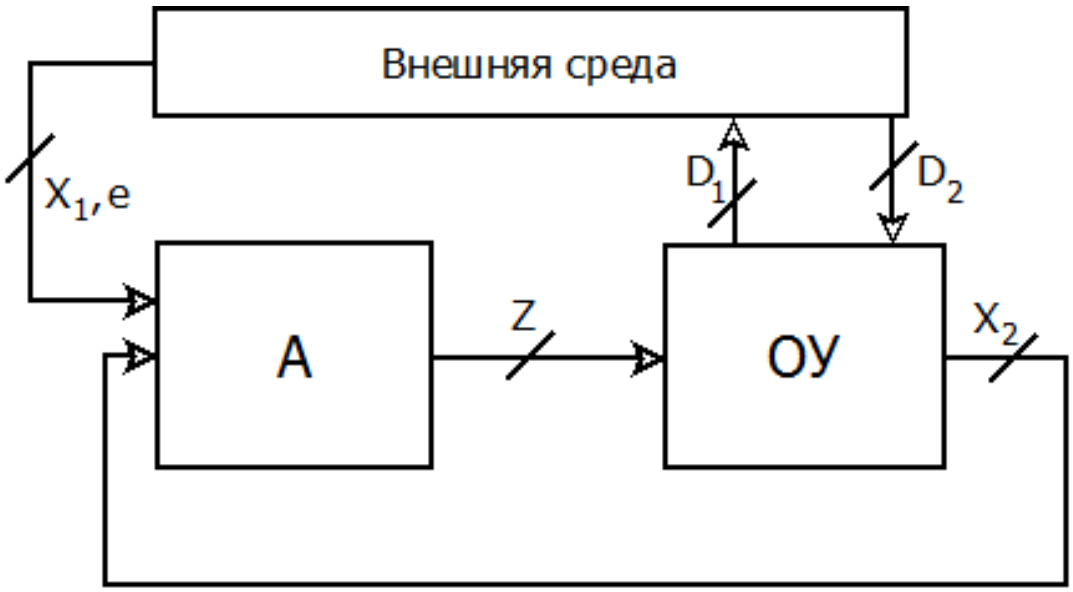

Рисунок 1.8.1. Автоматные программы

От внешней среды в автомат (*A*) приходят входные воздействия (переменные X_1 и события *e*). Выходные воздействия автомата передаются объекту управления (*ОУ*). Объект управления обменивается данными (D_1 и D_2) с внешней средой и подаёт на вход автомата переменные (X_2).

Выводы по главе 1

Описаны основные понятия, использующиеся в диссертации

Глава 2. Обзор методов верификации автоматных программ разных типов

2.1. Сравнение с аналогами

Сравнение с аналогами приведено в табл. 1.2.1.

Таблица 1.2.1. Сравнение с аналогами

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1.	LTL	-	+	-	-	-	-	-
2.	LTL	-	+	-	-	-	-	-
3.	CTL	-	+	-	-	-	-	-
4.	CTL	-	+	-	-	-	-	-
5.	LTL	-	+	-	-	-	+	-
6.	LTL	-	+	-	-	-	+	-
7.	-	+	-	+	-	-	?	-
8.	-	+	-	+	-	-	?	-
9.	LTL	+	+	+	+	-	+	-
10.	CTL	+	-	-	-	+	-	+

В столбцах указаны свойства, поддерживаемые соответствующими методами:

1. Темпоральная логика.
2. Поддержка многопоточных программ.
3. Поддержка UML-диаграмм *Statechart*.
4. Поддержка автоматов *Stateflow*.
5. Интерактивность.
6. Возможность передавать события между параллельными автоматами.
7. *Bounded Model Checking* (ограниченная проверка моделей).
8. Поддержка автоматов стандарта *IEC 61499*.

В строках приведены известные инструментальные средства и методы верификации автоматных программ.

1. *Unimod.Verifier* (http://is.ifmo.ru/verification/_jaminov.pdf) [21].
2. *Automata Verificator* (http://is.ifmo.ru/papers/automataverificator) [25].
3. *FSM Verifier* (http://is.ifmo.ru/download/2008-02-25_politech_verification_kurb.pdf) [27].
4. *CTL Verifier* (http://is.ifmo.ru/download/2008-02-25_politech_verification.pdf) [33].
5. *Latella D., Majzik I., Massink M.* Automatic verification of a behavioral subset UML stetechart diagrams using the *Spin* model-checker (http://home.mit.bme.hu/~majzik/publicat/fac99.pdf) [7].
6. *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) [19].
7. *Chen C., Sun J., Liu Y., Dong J., Zheng M.* Formal Modeling and Validation of Stateflow Diagrams [40]. (http://link.springer.com/article/10.1007%2Fs10009-012-0235-0#page-1).
8. *Miyazawa A., Cavalcanti A.* Refinement-based verification of sequential implementations of Stateflow charts. (https://www.academia.edu/3531758/Refinement-based_verification_of_sequential_implementations_of_Stateflow_charts) [39]
9. *Pingree P. J., Mikk E., Holzmann G. J., Smith M. H., Dams D.* Validation of mission critical software design and implementation using model checking [spacecraft]. (http://ieeexplore.ieee.org/xpl/abstractAuthors.jsp%3Farnumber%3D1067982&rct=j&q=&esrc=s&sa=U&ei=OJ8hVPj_D8PmywORpIDQBg&ved=0CBMQFjAA&sig2=w8kMGk_oXI3Ptc6MFdmCNg&usg=AFQjCNHnIrnNTGE8BmOQVcZ3hQOvPXBQVw) [12].
10. *Вяткин В. В., Дубинин В. Н.* Верификация приложений IEC 61499 на основе метода Model checking. (http://cyberleninka.ru/article/n/verifikatsiya-prilozheniy-iec-61499-na-osnove-metoda-model-checking) [38].

Из таблицы 1.2.1 следует, что в настоящее время отсутствует метод верификации автоматных программ, модели которых предложены в третьей главе диссертации.

Выводы по главе 2

1. Обосновано, что для автоматных программ может быть обеспечен высокий уровень автоматизации процесса верификации
2. Выполнен анализ инструментальных средств для верификации программ
3. Из выполненного сравнения подходов к верификации автоматных программ следует, что в настоящее время отсутствует метод для верификации автоматных программ рассматриваемого в диссертации вида.

Глава 3. Описание предлагаемого метода

3.1. Описание автоматной модели

В методе используется параллельная система взаимодействующих иерархических конечных автоматов [41 – 45]. Под иерархическим автоматом в данной работе понимается система вложенных автоматов.

3.1.1. Математическая модель управляющего автомата

Каждый автомат – это кортеж $<S, s_0, T, E, X, Z, B, \delta, \lambda>$, где:

- S – конечное множество состояний;
- s_0 – начальное состояние;
- $T \subseteq S$ – множество допускающих состояний (возможно, пустое);
- E – множество входных воздействий (событий);
- X – множество переменных;
- Z – множество выходных воздействий;
- B – множество логических функций от переменных из множества X (условий на переменные из множества X);
- $\delta: S \times E \times B \to S$ – отношение переходов;
- $\lambda = \lambda_1 \cup \lambda_2 \mid \lambda_1: S \times X \times B \to F, \lambda_2: S \to F$ – отношение выходных воздействий.

Каждое множество одинаковых автоматов назовём *автоматным типом*.

Множество Z состоит из двух множеств Z_1 и Z_2. Первое из них – это множество выходных воздействий, которые состоят из имён вызываемых подпрограмм. Множество Z_2 состоит из выходных воздействий, которые состоят из произвольного программного кода. В модель на языке *Promela* этот код переносится без изменений.

Переходы автоматов выполняются по событиям. На переходах могут быть также и охранные условия [77]. Однако что делать, если в автомат пришло 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, автомат переходит в недопускающее состояние;
- при появлении события, по которому нет перехода, оно игнорируется (добавляются пустые петли по всем таким событиям).

Автомат может иметь конечное число переменных целочисленных типов (включая массивы). Эти переменные могут иметь 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 являются общими для всей системы автоматов.

Выходные воздействия автомата бывают двух типов:

1. На переходах и в состояниях может быть выполнен любой код.
2. Запуск на переходе, а также при входе в состояние и при выходе из него, функций на целевом языке программирования, определяемых пользователем после того, как сгенерирован код.

3.1.2. Математическая модель вложенных автоматов

Рассмотрим автомат типа A_i. Пусть S_{A_i} – множество его состояний. Пусть M – множество автоматов в системе. Определим отношение вложенности как множество отношений $\nu : \{\nu_i \mid \nu_i : S_{A_i} \to M\}$.

Отметим, что автомат может иметь вложенные автоматы любого типа, кроме собственного, так как иначе будет бесконечная рекурсия. Циклическая рекурсия также запрещена.

3.1.3. Математическая модель параллельно работающих автоматов

Модель для параллельно работающих автоматов – это тройка <σ, ρ, ω>, где

- $\sigma : \{\sigma_i \mid \sigma_i : S_{A_i} \to M\}$ – множество отношений, описывающих запуск автоматов;
- $\rho = \{\rho_i \mid \rho_i : S_{A_i} \times E \times B \to M \times E\}$ – множество отношений, описывающих коммуникацию между автоматами;
- $\omega = \{\omega_{i,j} \mid \omega_{i,j} : X_i \to X_j\}$ – множество отношений, описывающих совместно используемые переменные.

При реализации параллельных автоматов каждый автомат будем реализовывать в своём потоке.

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

Автомат может взаимодействовать с другим автоматом, выступая источником событий для него. События отправляются асинхронно.

Автомат может также использовать отмеченные специальным модификатором переменные другого автомата.

Таким образом, в системе может быть несколько автоматов с одинаковыми графами переходов. При этом часть этих автоматов могут быть вложенными, а часть нет.

3.2. Описание предложенного метода верификации

Для того чтобы провести верификацию программы при помощи верификатора *Spin* [73, 74], требуется создать модель программы на языке *Promela* [73, 74] и формализовать требуемые свойства (спецификацию) на языке линейной темпоральной логики (LTL). В данной работе модель автоматически строится по системе графов переходов автоматов [75, 76]. Процесс построения модели описан в разделах 3.2.1 – 3.2.3.

Обозначим автоматный тип – *AType*, автоматный объект – *aObject*. Пусть для автоматного типа *AType* состояния s_0, s_1 и т. д., а события – e_0, e_1 и т. д. Переменные этого автомата – x_0, x_1 и т. д., а внешние воздействия (второго типа – вызываемые функции) z_0, z_1 и т. д. Пусть автоматный тип *AType* имеет вложенный автомат *nested*. Пусть *AType* запускает автомат *fork*.

Метод верификации состоит из следующих этапов:

1. Генерация кода на языке *Promela* [4].
2. Преобразование LTL-формул.
3. Запуск верификатора *Spin*.
4. Преобразование контрпримера.

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

3.2.1. Построение *Spin*-модели для управляющего автомата

Этап 1. Подготовка данных

Все состояния каждого автоматного типа нумеруются, и для них создаются константы. Для каждого автоматного типа состояния нумеруются отдельно. Имя константы состоит из имени автоматного типа и имени состояния, разделённых 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 нумеруются с единицы и для них создаются константы. Для событий применяется сквозная нумерация. Ноль означает, что в данный момент событие не обрабатывается. Пример:

```
#define e0 1
#define e1 2
```

Все выходные воздействия второго типа (вызываемые функции) нумеруются, и для них создаются константы аналогично состояниям.

Все вызовы вложенных и запуски параллельных автоматов нумеруются аналогично состояниям.

Для каждого типа автоматов создаётся следующая структура:

- `byte state` – номер текущего состояния;
- `byte curEvent` – номер обрабатываемого 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 автомата создаётся экземпляр этой структуры.

Этап 2. Моделирование шага автомата

Для каждого автоматного типа создаётся функция, которая моделирует один шаг работы автомата

```
inline Machine(machine, evt),
```

где `machine` – это переменная соответствующего автоматного типа, а `evt` – событие при входе в состояние.

Переходы записывается при помощи охранных команд Дейкстры [77] – охранных условий. На листинге 3.2.1.1 приведён код на *Promela* для перехода из состояния s_1 в s_2 по событию e_1 с выходным воздействием z_1. При этом `(evt == e1)` – охранное условие.

Листинг 3.2.1.1. Переход автомата из состояния s_1 в s_2 по событию e_1

```
if(machine.state == s1)
  if
  ::((evt == e1)) ->
    machine.curEvent = e1;
    machine.state = s2;
    machine.functionCall = 1;
  ::((evt == e2)) ->
    …
  fi
fi
```

Если переход на графе помечен условием, то оно дописывается к охранному условию. На листинге 3.2.1.2 приведён код на *Promela*, который моделирует условие $v_1 > 0$ (на переходе из листинга 3.2.1.1).

Листинг 3.2.1.2. Условие $v_1 > 0$ на переходе

```
if
  ::((evt == e1) && (machine.v1 > 0)) ->
    machine.curEvent = e1;
    machine.state = s2;
    machine.functionCall = 1;
  fi
```

3.2.2. Построение *Spin*-модели для вложенных автоматов

Все вложенные автоматы нумеруются.

Пусть автомат A_2 вложен в состояние s_2 автомата A_1. Так как переход автомата A_2 осуществляется, когда пришло событие, по которому нет перехода в автомате A_1, то переход дописывается при помощи охранного условия *else*:

```
:: else ->
        machine.nestedMachine = a2_call;
        A2(a2, evt);
```

где *evt* – событие, которое пришло на вход автомату A_1.

3.2.3. Построение *Spin*-модели для параллельных автоматов

Для каждого автомата создаётся *канал*, по которому происходит передача событий. Каналу присваивается имя <Имя_автомата>_ch. Канал для автомата a_1 будет называться a1_ch.

Для каждого автомата, кроме вложенных, создаётся *процесс*, который извлекает из канала событие и запускает функцию перехода автомата с этим событием. На листинге 3.2.3.1 приведён процесс с автоматом a_1 типа A_1.

Листинг 3.2.3.1. Процесс с автоматом a_1

```
proctype a1Proc ()
{
    byte newEvt;
    a1.started= true;
    a1_ch ? newEvt;
    a1ParamChange();
    do
        :: a1.finished == false ->
            a1_ch ? newEvt;
```

```
                A1(a1, newEvt);
            :: else -> skip;
        od;
    }
```

Жирным выделен запуск функции, моделирующей шаг автомата. Построение функции описано во втором этапе раздела 3.2.1.

Также, для каждого автомата, кроме вложенных, создаётся процесс, который недетерминированно выбирает событие и отправляет его в канал автомата.

Для *публичных переменных* перед каждым шагом автомата вызывается функция, которая их недетерминированно изменяет. Этой функции присваивается имя <Имя_автомата>VolatileChange. На листинге 3.2.3.2 для автомата a_1 приведена функция для изменения массива b, состоящего из пяти четырёхбайтных элементов (массивы используются для того, чтобы верифицировать автоматы *Stateflow*).

Листинг 3.2.3.2. Функция изменения переменных для автомата a_1

```
inline a1VolatileChange()
{
    int r;
    int ind = 0;
    do
        :: ind < 5 ->
            random(r, 32);
            a1.b[ind] = r;
            ind = ind + 1;
        :: else -> break;
```

```
    od;
}
```

Для переменных-*параметров* автомата строится аналогичная функция, которая вызывается один раз при его запуске.

Множество отношений σ, описывающих запуск автоматов, кодируется следующим образом: пусть автомат a_1 запускает автомат a_2 в состоянии s_1. Тогда в модель дописывается следующий код:

```
assert (!a2.started);
run a2Proc();
```

Множество отношений ρ, описывающих коммуникацию между автоматами, кодируется следующим образом: пусть автомат a_1 отправляет событие e_2 на переходе в состояние s_2 по событию e_1 автомату a_2. Тогда в код соответствующего перехода в функции, моделирующей шаг автомата a_1, дописывается отправка события e_2 в канал автомата a_2 (листинг 3.2.3.3).

Листинг 3.2.3.3. Коммуникация между автоматами

```
if
    ::((evt == e1)) ->
        machine.state = A1_s2;
        machine.curEvent = e1;
        sendEvt = e1;
        a2_ch ! sendEvt;
    :: … ->
    …
fi;
```

Множество отношений ω, описывающих совместно используемые переменные, моделируется так: пусть переменная x_1 автомата a_1 является

входной переменной x_2 автомата a_2. Тогда на каждом переходе автомата a_1 дописывается следующая инструкция:

```
a2.x2 = x1;
```

На рисунке 3.2.3.1 схематически изображена модель параллельных иерархических автоматов.

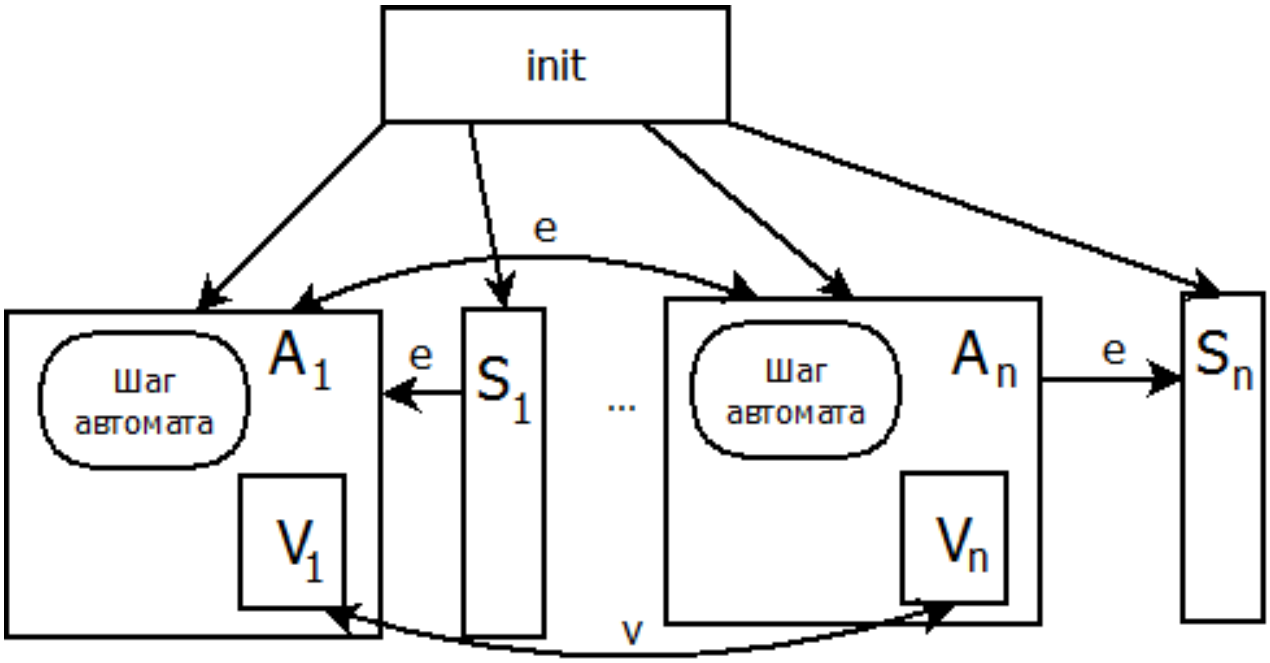

Рисунок 3.2.3.1. Модель параллельных иерархических автоматов

Основные компоненты модели:

- A_i – процессы, содержащие автоматы. Каждый процесс содержит один автомат. Процесс состоит из цикла, который принимает события при помощи *каналов Spin* и запускает функцию, реализующую шаг автомата. Каждая функция реализует один автоматный тип и принимает на вход автомат и событие.
- *init* – процесс, запускающий процессы A_i.
- S_i – процессы-модели поставщиков событий.
- X_i – переменные автоматов.

3.2.4. Преобразование LTL-формул

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

- `aObject.si` – автомат *aObject* перешёл в состояние s_i.
- `aObject.ei` – в автомат *aObject* пришло событие e_i.
- `aObject.zi` – автомат *aObject* вызвал функцию (выходное воздействие) z_i.
- `aObject->nested` – автомат *aObject* передаёт управление 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 расширенной формулы в нотацию *Spin*:

1. Все высказывания в фигурных скобках перенумеровываются.
2. Каждое такое высказывание преобразовывается в терминах модели на *Promela*, и из него создаётся макрос.
3. Макросы подставляются в исходную LTL-формулу.

Формула (1) преобразуется 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.

3.2.5. Запуск верификатора *Spin*

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

1. Построение программы *pan*: при запуске с ключом `-a` верификатор генерирует по *Spin*-модели соответствующую программу *pan* на языке *C*.
2. Компиляция программы *pan*. При компиляции можно определить константы, которые влияют на то, как в памяти будет храниться *модель Крипке*. Наиболее компактный вариант хранения задаётся константой `BITSTATE`, однако в этом случае происходит аппроксимация, и верификация может быть не точна.
3. Запуск программы *pan*. Эта программа может быть запущена с разными ключами, важнейшим из которых является `-a` (поиск допускающих циклов).
4. Анализ контрпримера. Описан в разделе «Преобразование контрпримера».

3.2.6. Преобразование контрпримера

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

Для каждого действия автомата создаётся пометка при помощи функции `printf`. На языке *Promela* она работает аналогично функции `printf` из языка *C* [78, 79]. Во время *случайной симуляции* [80] она выводит текст на экран, а во время верификации этот текст появляется в контрпримере. После этого проводится его разбор и вывод на пользователю.

3.2.7. Интерактивность

Одним из главных ограничений проверки моделей является ограничение на размер *модели Крипке*, так как она растёт экспоненциально от размера исходной программы. Например, *модели Крипке* для программ, описанных в

пятой главе, имеют более 10^7 состояний и занимают более 800 Мб памяти каждая.

Для того чтобы уменьшить модель, в работе предлагается строить её интерактивно. Для этого вводится возможность выбирать, какие уровни абстракции автоматной системы входят в модель, а какие нет. Также модель структурируется понятным для человека образом для того, чтобы пользователь мог самостоятельно модифицировать построенную модель.

Подразделы этого раздела описывают компоненты *Spin*-моделей, обеспечивающие интерактивность.

3.2.7.1. Переменные

Для переменных введём 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. Публичные. Такие переменные могут быть изменены в любом месте программы, в которую входит система иерархических автоматов. В модели перед каждым переходом автомата каждой такой переменной недетерминированно присваивается произвольное значение.
    c. Совместно используемые. К таким переменным автомата имеют доступ другие автоматы, параллельно работающие с данным.
    d. Параметры. Извне изменяются только один раз, при запуске автомата. В остальном они подобны локальным.

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

3.2.7.2. Параллелизм

Введём два уровня абстракции: учитывать параллелизм в модели или не учитывать его. Если параллелизм в модели не учитывается, то в *Spin*-модель не добавляется информация из отношений σ и ρ математической модели для параллельно работающих автоматов, а из отношения ω (этой из математической модели) добавляется информация только о совместных переменных для вложенных автоматов.

3.2.7.3. Источники событий

В качестве источников событий для автоматов 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.

Если паралеллизм в модели не учитывается, то автоматы не будут являться источниками событий друг для друга, в то время как по переменным они могут взаимодействовать.

3.2.7.4. Процесс верификации

Интерактивность процесса верификации основывается на возможностях верификатора *Spin*, которые описаны в разделе 3.2.5.

3.3. Генерация программного кода

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

В отличие от таких инструментов, как *Unimod* [81] и *Stateflow* [51], 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:

- Для каждого автоматного типа генерируется отдельный класс в отдельном файле. Такой класс называется автоматизированным классом [45].
- Сгенерированный класс содержит: функцию переходов 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, так как он не сохранится. Пользовательский код из остальных мест будет полностью сохранён.
- Если пользователь добавит новые выходные воздействия второго типа, то их определения будут добавлены к сгенерированному коду.
- Пользователь может задать пространство имён (или пакет в языке *Java*), в котором будет находиться сгенерированный код. Если между генерациями кода пространство имён 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, то генерацию таких классов можно отключить.

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

3.3.1. Первичная генерация кода

При первичной генерации кода файл с автоматизированным классом полностью перезаписывается. Если файла не существовало, то он создаётся.

3.3.2. Повторная генерация кода

При повторной генерации кода файл с предыдущей версией считывается в память и построчно отправляется на вход обработчику *CodeProcessor*. *CodeProcessor* читает строку, и в зависимости от её содержимого, отправляет события автомату *GenFinder*. Этот автомат записывает код обратно в файл.

На рисунке 3.3.2.1 изображена диаграмма состояний конечного автомата *GenFinder*, который управляет повторной генерацией кода. Для удобства переходы перенумерованы и их номера отмечены красным.

Список событий автомата *GenFinder*:

- `_class` – объявление автоматизированного класса.
- `_namespace` – объявление генерируемого пространства имён.
- `action_found` – объявление выходного воздействия второго типа.
- `eof` – конец файла.
- `gen_functions` – маркер начала или конца генерируемых функций.
- `next_line` – новая строка кода, в которой не встретилось ничего из вышеперечисленного.

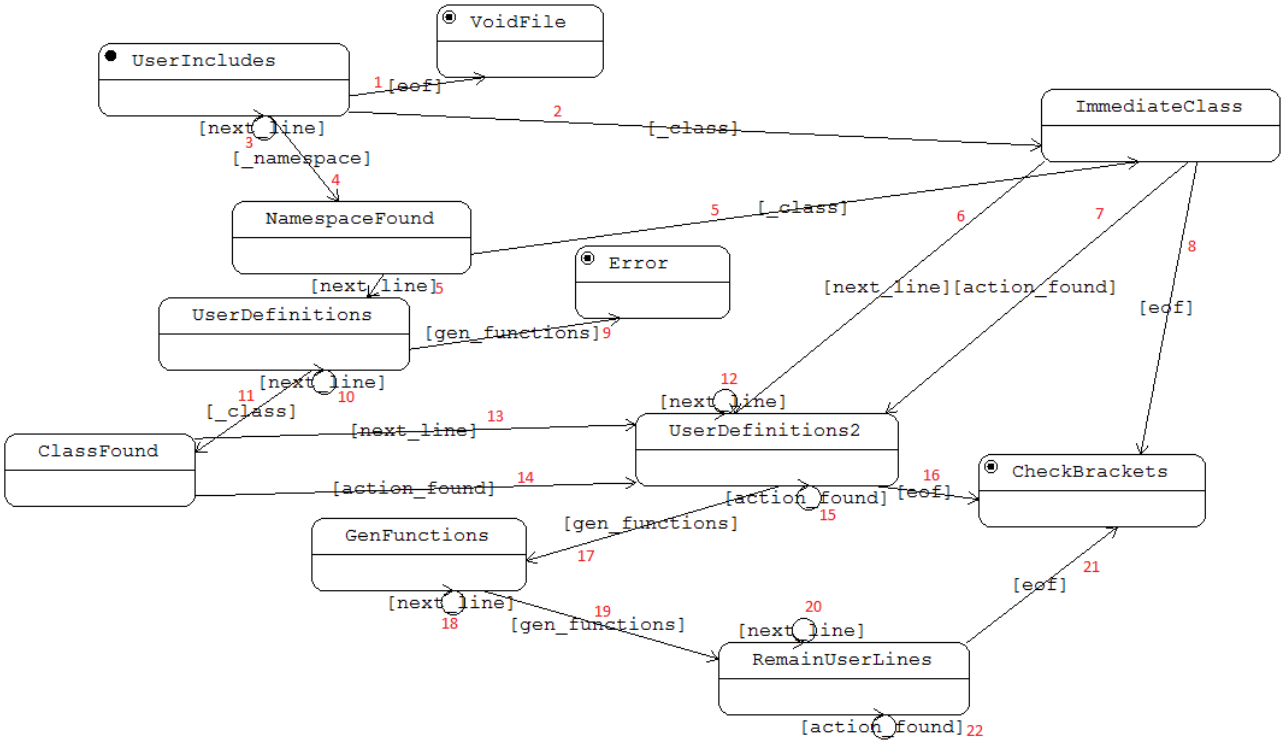

Рисунок 3.3.2.1. Автомат *GenFinder*

В таблице 3.3.2.1 приведён список выходных воздействий автомата *GenFinder*.

Таблица 3.3.2.1. Выходные воздействия автомата *GenFinder*

Имя выходного воздействия	**Описание**
`CheckBrackets`	Проверяет корректность расстановки фигурных скобок. Если в конце файла не хватает фигурных скобок, дописывает их.
`InsertNamespace`	Записывает объявление пространства имён в буфер.
`RegisterAction`	Отмечает найденное выходное воздействие второго типа.
`WriteBuffer`	Записывает буфер в файл.
`WriteCurLine`	Записывает текущую строку в буфер.

Продолжение таблицы 3.3.2.1. Выходные воздействия автомата *GenFinder*

`WriteGeneratedFunctions`	Записывает в буфер все сгенерированные члены автоматизированного класса (всё, что непосредственно относится к автомату).
`WriteNewFile`	Перезаписывает файл полностью (проводит первичную генерацию кода).
`WriteRemainActions`	Записывает в буфер те выходные воздействия второго типа, которые не были найдены в файле.

Для того чтобы не перегружать диаграмму переходов, выходные воздействия автомата приведем в табл. 3.2.2.2 (в автомате *GenFinder* все выходные воздействия на переходах). Моноширинным шрифтом набраны имена выходных воздействий, обычным – комментарии.

Таблица 3.2.2.2. Выходные воздействия автомата *GenFinder*

Номер перехода	**Список выходных воздействий**
1	`WriteNewFile`

2	`InsertNamespace`
	`WriteCurLine`
	`WriteBuffer`

3	`WriteCurLine`
4	`WriteCurLine` `WriteBuffer`

Продолжение таблицы 3.2.2.2. Выходные воздействия автомата *GenFinder*

5	WriteCurLine WriteBuffer
6	WriteCurLine
7	RegisterAction WriteCurLine
8	WriteGeneratedFunctions WriteRemainActions WriteBuffer
9	WriteNewFile
10	WriteCurLine
11	WriteCurLine WriteBuffer
12	WriteCurLine
13	WriteCurLine
14	RegisterAction WriteCurLine
15	RegisterAction WriteCurLine
16	WriteGeneratedFunctions WriteRemainActions WriteBuffer

Продолжение таблицы 3.2.2.2. Выходные воздействия автомата *GenFinder*

17	`WriteBuffer`
18	`WriteGeneratedFunctions`
19	`WriteCurLine`

20	`WriteRemainActions`
	`CheckBrackets`
	`WriteBuffer`

21	`RegisterAction` `WriteCurLine`

3.4. Верификация автоматов Stateflow

Stateflow – это составная часть *Simulink* (который входит в математический пакет *Matlab*), которая позволяет создавать системы автоматов и включать их в состав моделей *Simulink*. Примеры автоматов *Stateflow* приведены в разделе 3.8 на рисунке 3.8.1 и 3.8.2.

Верификация автоматов *Stateflow* состоит из двух шагов. На первом из них по этим автоматам строится система иерархических управляющих автоматов, описанных в разделах 3.1.1 – 3.1.3. На втором шаге проводится верификация построенной системы.

Первый шаг. Алгоритм построения системы иерархических управляющих автоматов следующий:

1. Автоматы, содержащие параллельные состояния, декомпозируются на систему параллельных автоматов;
2. Для каждого автомата *Stateflow* строится автомат $<S, s_0, T, E, X, Z, B, \delta, \lambda>$:

3. Состояния автоматы *Stateflow* переводятся в множество *S*. Начальное состояние переводится в s_0.
4. Конечные состояния переводятся в множество *T*.
5. События из автоматов *Stateflow* переводятся в множество *E*.
6. Переменные из автоматов *Stateflow* переводятся в множество X.
7. Выходные воздействия из автоматов *Stateflow* переводятся в множество *Z*.
8. Условия на переходах автоматов *Stateflow* переводятся в множество *B*.
9. По переходам автоматов *Stateflow* строится отношение δ.
10. По выходным воздействиям на переходах и в состояниях *Stateflow* строится отношение λ.

Второй шаг состоит в том, что построенная система верифицируется методом, изложенным в разделе 3.2.

3.5. Верификация автоматов стандарта IEC 61499

Стандарт *IEC 61499* предназначен для автоматизации технологических процессов и основан на системе конечных автоматов. В соответствии с этим стандартом, распределённая система управления представляется в виде взаимодействующих *функциональных блоков*. Эти блоки включают в себя *базовые функциональные блоки* (Basic Function Block) и *составные функциональные блоки* (Composite Function Block). Описание каждого функционального блоки представляет собой XML-файл. Верификация автоматов стандарта *IEC 61499* также состоит из двух шагов: построение системы иерархических управляющих автоматов и верификация построенной системы методом, описанным в разделе 3.2. В подразделах этого раздела описан первый шаг как для базовых, так и для составных функциональных блоков.

3.5.1. Верификация базовых функциональных блоков

Базовый функциональных блок (БФБ) (рисунок 3.5.1.1) состоит из следующих компонентов:

- описание конечного автомата;
- список входных событий;
- список выходных событий;
- список входных переменных;
- список выходных переменных;
- список *алгоритмов* (выходных воздействий).

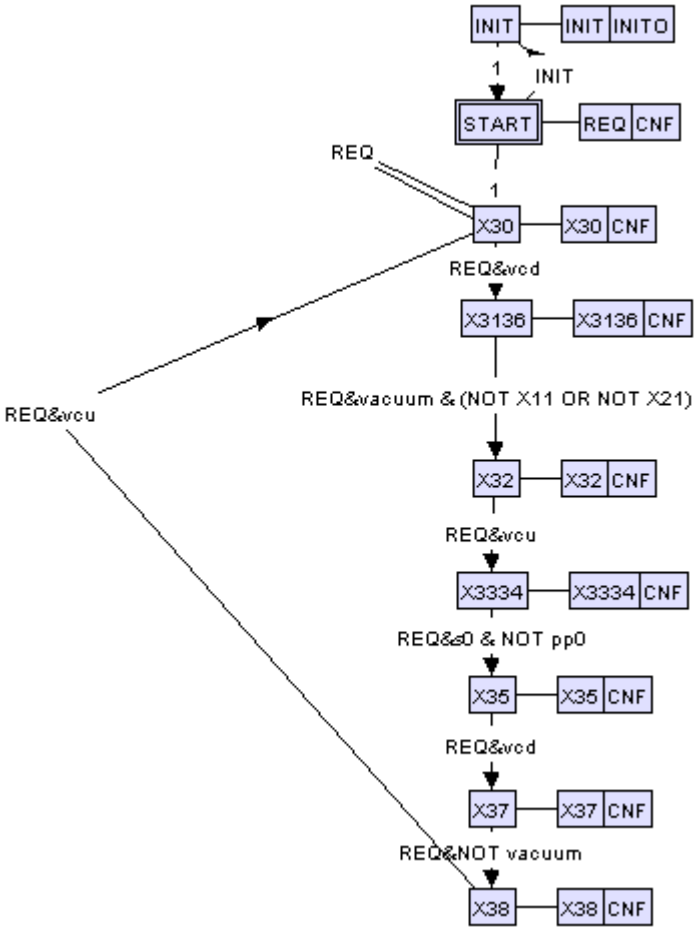

Рисунок 3.5.1.1. Пример автомата базового функционального блока

Описание конечного автомата состоит из состояний и переходов. Состояние, которое описано первым, является начальным. Состояния могут включать в себя алгоритмы. Переходы могут помечаться условиями. Условие на переходе состоит из события и логической функции от переменных.

Алгоритм представляет собой выходное воздействие, аналогичное выходным воздействиям из множества Z_2 из математической модели, описанной в разделе 3.1.1.

Листинг 3.5.1.1. XML-описание алгоритма

```
  <Algorithm Name="INIT" Comment="Initialization
algorithm" >
    <ST Text="
X30:=true;&#10;
VCGD:=false;&#10;
L2CGI:=false;&#10;
L1CGI:=false;&#10;
X31:=false;&#10;
X32:=false;&#10;
X33:=false;&#10;
X34:=false;&#10;
X35:=false;&#10;
X36:=false;&#10;
X37:=false;&#10;
X38:=false;&#10;
vc_down30:=false;&#10;
vc_down31:=false;&#10;
vc_down35:=false;&#10;
vc_down37:=false;&#10;
venturi_on:=false;&#10;
venturi_off:=false;&#10;
System.out.println(&#34;SFC4: INIT&#34;);&#10;" />
  </Algorithm>
```

В листинг для удобства чтения добавлены переводы строки внутри тэга `ST`, которых в XML-описании быть не должно.

Приведём алгоритм построения автомата <S, s_0, T, E, X, Z, B, δ, λ> (предложенного автором) по автомату из БФБ:

1. Все состояния из БФБ переносятся в множество S. При этом $s0$ – первое состояние из описания БФБ.
2. Множество T – пустое.
3. Все входные события из БФБ переносятся в множество E.
4. Все входные и выходные переменные переносятся в множество X.
5. Алгоритмы переносятся в множество Z_2.
6. Все логические функции на переходах БФБ переносятся в множество B.
7. По описаниям переходов БФБ строится отношение δ.
8. По описаниям вызываемых алгоритмов строится отношение λ.

3.5.2. Верификация составных функциональных блоков

Составной функциональный блок (СФБ, рисунок 3.5.2.2) описывает, как взаимодействуют между собой БФБ. Он содержит список принадлежащих ему БФБ и связи между ними. Связи могут описывать общие переменные и взаимодействие при помощи отправки событий.

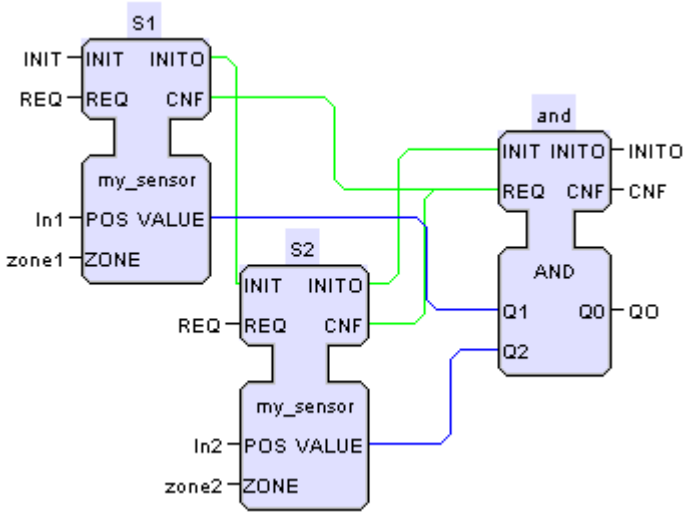

Рисунок 3.5.2.2. Составной функциональный блок

Пример связи, описывающей общие переменные:

```
<Connection Source="SFC1.X2" Destination="SFC3.X12"
            dx1="383.3333" />
```

Пример связи, описывающей взаимодействие при помощи отправки событий:

```
<Connection Source="SFC4.CNF" Destination="CNF"
            dx1="4005.5554" />
```

Приведём алгоритм, описывающий построение системы взаимодействующих иерархических конечных автоматов по СФБ.

1. По каждому БФБ, принадлежащему СФБ, строится автомат по алгоритму из раздела 3.5.1.
2. Отношение ρ, описывающее коммуникацию между автоматами в модели для параллельно работающих автоматов, строится следующим образом: для каждой связи, описывающей отправку события (обозначим его `FB1.E1`) в отношение ρ добавляются исходящие события `FB1.E1` из каждого состояния, в описании которого в `FB1` отмечено, что событие `E1` является выходным.
3. Отношение ω, описывающее общие переменные, напрямую строится при помощи связей, описывающих общие переменные.

Второй шаг. Система автоматов, построенная при помощи этого алгоритма, верифицируется при помощи метода, описанного в разделе 3.2.

3.6. Описание метода верификации однопоточных автоматов и инструментального средства Converter

В рамках бакалаврской работы [20] и магистерской диссертации [32] автором был разработан метод верификации однопоточных автоматных программ, написанных при помощи инструментального средства *Unimod* [81], позволяющего реализовывать систему вложенных автоматов. Практической реализацией разработанного метода является инструментальное средство *Converter*. Ниже описаны метод и его практическая реализация.

3.6.1. Описание метода

Идея метода состоит в следующем:

1. Построить модель системы графов переходов автоматной программы на языке *Promela*.
2. Преобразовать LTL-формулу с требованиями к автоматной программе к виду, который понятен верификатору *Spin*.
3. Построенную модель и формулу подать на вход верификатору *Spin*.
4. Проанализировать отчёт, выданный верификатором *Spin*, и построить контрпример.

Опишем метод построения модели.

5. Подготовка исходных данных.
   5.1. Для каждого автомата `Ai` завести переменную `stateAi`, в которой будет храниться номер текущего состояния. На языке *Promela* это описывается следующим образом:
   ```
   int stateAi;
   ```
   5.2. Для каждой пары автоматов (`Ai`, `Aj`) завести два канала передачи сообщений. На языке *Promela* это описывается следующим образом:
   ```
   chan AiAj = [1] of {byte};
   chan AjAi = [1] of {byte};
   ```
   5.3. Завести два канала передачи сообщений для процесса, который эмулирует источник событий:
   ```
   chan epin = [1] of {byte};
   chan epout = [1] of {byte};
   ```
   5.4. Завести одну переменную для событий:
   ```
   int lastEvent;
   ```
   5.5. Завести две переменных для выходных воздействий:
   ```
   int o;
   int z;
   ```
   5.6. Каждому состоянию присвоить уникальный номер, используя сквозную нумерацию для всех автоматов. Переход в новое

состояние с номером k будет осуществляться присвоением переменной stateAi числа k:

```
stateAi = k;
```

5.7. Событие exx (xx – номер события) на переходе между состояниями описывается в модели следующим кодом:

```
lastEvent = xx;
```

6. Построение

6.1. Для каждого автомата Ai создать процесс. На языке *Promela* это записывается следующим образом:

```
active proctype Ai() {
/* тело процесса */
}
```

Модификатор active означает, что процесс будет запущен в начальном состоянии модели.

6.2. Для каждого автомата Ai в теле созданного процесса выполнить шаги 2.3 – 2.18.

6.3. Определить переменную x для приема сообщений от других процессов и переменную y, в которой будет храниться номер процесса, от которого получено сообщение:

```
byte x;
byte y;
```

6.4. Для стартового автомата: инициализировать переменные o и z значением -1:

```
o = -1;
z = -1;
```

6.5. Определить начальное (стартовое) состояние s. Присвоить:

```
stateAi = s;
```

6.6. Для всех автоматов, кроме стартового: ждать, пока данному процессу не придёт сообщение; в переменную *y* записать номер автомата, приславшего сообщение; на языке *Promela* это записывается следующим образом:

```
if
::A1Ai ? x ->
 y = 1;
::A2Ai ? x ->
 y = 2;
…
::AnAi ? x ->
 y = n;
fi;
```

6.7. Для всех автоматов, кроме стартового, построить цикл:

```
do
::(x == 1) ->
::(x == 9) ->
 break;
od;
```

Здесь сообщение о запуске автомата имеет номер 1, а сообщение о завершении работы системы имеет номер 9.

6.8. Построить цикл:

```
do
::(stateAi == s1) ->
 printf("State Ai 1 : Имя состояния\n");
::(stateAi == s2) ->
 printf("State 2 : Имя состояния\n");
…
::(stateAi == sk) ->
 printf("State k : Имя состояния\n");
od;
```

Здесь `s1, …, sk` – номера состояний автомата `Ai`.

Инструкция `printf` аналогична соответствующей инструкции из языка *C*. Пометка используется в дальнейшем для восстановления контрпримера. Для всех автоматов, кроме стартового, этот цикл записывается в условие `(x == 1)` цикла, построенного на шаге 2.6.

6.9. Если состояние содержит выходные воздействия, то для каждого выходного воздействия `oi.zj` записать следующее:

```
printf("Action oi.zj\n");
o = i;
z = j;
o = -1;
z = -1;
```

6.10. Поставить метку (пусть `Ai` и `sj` – текущий автомат и текущее состояние соответственно):

```
Ai_STATE_j:
```

6.11. Оповестить источник событий. Для этого требуется по каналу `epin` отправить число 1. На языке *Promela* это описывается следующим образом:

```
epin ! 1;
```

6.12. Ждать ответ. Если текущий автомат – стартовый, то ждать пока по каналу `epout` от источника событий придёт число 1. На языке *Promela* это записывается следующим образом:

```
epout ? 1;
```

Если текущий автомат не является стартовым, то выполнить шаг 2.6.

6.13. Для каждого состояния `sj` найти все возможные переходы `(sj, sl)` из него. К условию `(stateAi == sj)` дописать конструкцию `if`, а для каждого перехода `(sj, sl)` дописать в конструкции `if` следующее:

```
::условие ->
  printf("Going to state Ai l : <Имя состояния>");
  stateAi = sl;
```

Условием может быть наступление события, условие на состояния автоматов и т. д.

6.14. Если в некоторое состояние `sj` вложен автомат `Am`, то дописать в условие `(stateAi == sj)` передачу сообщения на запуск автомата `Am` для процесса этого автомата и ждать завершения и

сообщения от поставщика событий (если текущий автомат – стартовый) либо сообщения от другого автомата (если текущий автомат не является стартовым):

```
AiAm ! 1;
if
::AmAi ? x;
::eoput ? 1 ->
    goto Ai_STATE_sj;
fi;
```

либо

```
AiAm ! 1;
if
::A1Ai ? x ->
    y = 1;
::A2Ai ? x ->
    y = 2;
  …
::AnAi ? x ->
    y = n;
fi;
```

6.15. Если состояние `st` – конечное, то в условие `(stateAi == st)` дописать инструкцию завершения цикла:

```
break;
```

6.16. Для всех автоматов, кроме стартового: после цикла, построенного на шаге 2.7 дописать отправку сообщения о завершении работы автомата `Ai` процессу автомата, который запустил автомат `Ai`. На языке *Promela* это записывается следующим образом:

```
if
:: y = 1 ->
    A1Ai ! x;
:: y = 2 ->
    A2Ai ! x;
 …
:: y = n ->
```

```
    AnAi ! x;
fi;
```

6.17. Для всех автоматов, кроме стартового, ждать, пока данному процессу не придёт сообщение. Выполнить шаг 2.6.

6.18. Для стартового автомата. После цикла, построенного на шаге 2.7, отправить всем остальным процессам сообщение о завершении работы:

```
AiA1 ! 9;
AiA2 ! 9;
…
AiAn ! 9;
epin ! 9;
```

6.19. Дописать в модель источник событий.

```
active proctype eventProvider() {
  byte x;
  do
  ::epin ? 1 ->
    if
    ::lastEvent = 0;
    …
    ::lastEvent = M;
    ::lastEvent = -1;
    fi;
  epout ! 1;
  ::epin ? 9 ->
    lastEvent = -1;
    epout ! 9;
  break;
  od;
}
```

Здесь все события имеют номера от 0 до *M*. Минус единица означает, что никакого события не произошло.

6.20. Дописать в модель требования (проверяемые свойства), преобразованные с помощью верификатора *Spin* с языка *LTL* на язык *Promela*.

Этот метод был доработан для многопоточных программ (см. разделы 3.1 – 3.5).

3.6.2. Описание инструментального средства *Converter*

Converter является практической реализацией описанного выше метода верификации автоматных программ и позволяет проводить верификацию программ указанного класса с высоким уровнем автоматизации.

По автоматной программе *Converter* создает модель, в которой отброшены несущественные детали. LTL-формула преобразовывается в пригодный для верификатора *Spin* вид.

При построении модели используются:

- графы переходов автоматов;
- взаимодействие автоматов по вложенности;
- выходные воздействия, обозначаемые, как **z** с соответствующими индексами;
- события на переходах.

Построенная модель абстрагируется от следующих сущностей:

- входных переменных, обозначаемых, как ***x*** с соответствующими индексами;
- выходных переменных, обозначаемых, как ***y*** с соответствующими индексами.

Такая абстракция позволяет генерировать более простые модели, обеспечивая возможность верификации программ большей размерности. Вместе с тем, более подробная модель позволяет более точно верифицировать программы.

Инструментальное средство принимает следующие входные параметры:

- путь к XML-описанию автоматной программы;
- имя файла, в который инструментальное средство сохранит построенную модель;
- LTL-формулу;
- имя файла, в который инструментальное средство сохранит отчёт о проведённой верификации.

Если LTL-формула не была задана, то верификация проводиться не будет. В этом случае *Converter* только построит модель автоматной программы.

Если не был задан какой-то из остальных параметров, то инструментальное средство использует значение по умолчанию. В частности, если не был задан путь к XML-описанию автоматной программы, *Converter* будет использовать в качестве входного файла `A1.xml`.

Для построения LTL-формул используются следующие переменные:

- *stateAi* для проверки условий на состояние автомата *Ai*;
- *lastEvent* для проверки условий на произошедшие события;
- Две переменные *o* и *z* для проверки условий на выходные воздействия. Переменная o используется для обозначения номера объекта управления, переменная *z* используется для обозначения индекса выходного воздействия.

3.6.2.1. Архитектура *Converter*

Первичное отображение

В первичном отображении представлены диаграмма классов (рисунок 3.6.2.1.1) и представление декомпозиции на модули в виде диаграммы компонентов (рисунок 3.6.2.1.2).

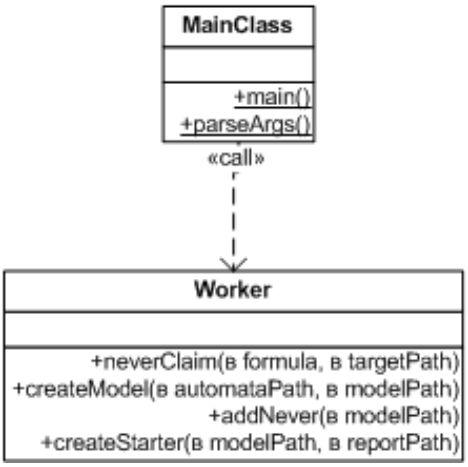

Рисунок 3.6.2.1.1. Диаграмма классов

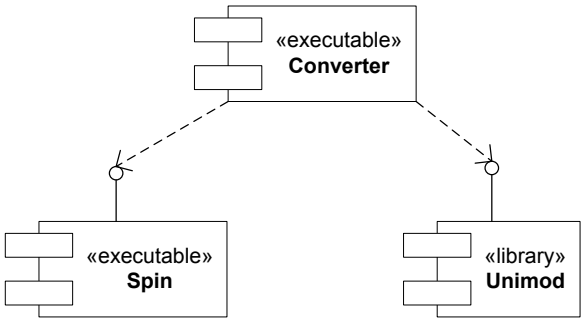

Рисунок 3.6.2.1.2. Диаграмма компонентов

Каталог элементов

Каталог элементов приведён в таблице 3.6.2.1.1.

Таблица 3.6.2.1.1. Каталог элементов

Элемент	Описание
1. Компонент *Unimod*	Библиотека инструментального средства *Unimod*. Используется для чтения автоматной программы из *XML*-файла.
2. Компонент *Spin*	Инструментальное средство *Spin* и программа *pan*, которую строит *Spin* во время работы. Производит верификацию модели.

Продолжение таблицы 3.6.2.1.1. Каталог элементов

3. Компонент *Converter*	Запускается в пакетном режиме. Решает следующие задачи: • строит из автоматной программы модель; • преобразовывает LTL-формулу; • подает построенную модель и LTL-формулу на вход компоненту *Spin*. • собирает отчёт о проведённой верификации.
3.1.Класс *MainClass*	Содержит в себе пользовательский интерфейс.
3.1.1. Метод *main*	Точка входа в программу *Converter*.
3.1.2. Метод *parseArgs*	Производит разбор входных параметров.
3.2.Класс *Worker*	Решает следующие задачи: • строит из автоматной программы модель; • преобразовывает LTL-формулу; • подает построенную модель и LTL-формулу на вход компоненту *Spin*. собирает отчёт о проведённой верификации.
3.2.1. Метод *neverClaim*	Преобразовывает LTL-формулу в формат верификатора *Spin* и запускает парсер LTL-формул верификатора *Spin*. В результате работы получается конструкция *never claim*.

Продолжение таблицы 3.6.2.1.1. Каталог элементов

3.2.2. Метод *createModel*	Создает модель автоматной программы на языке *Promela*.
3.2.3. Метод *addNever*	Копирует конструкцию *never claim*, сгенерированную при помощи метода *neverClaim* в конец файла с построенной моделью.
3.2.4. Метод *createStarter*	Запускает инструментальное средство *Spin* в режиме верификации.

3.6.2.2. Пользовательский интерфейс

Инструментальное средство *Converter 2.0* запускается в пакетном режиме и имеет консольный пользовательский интерфейс. Командная строка для запуска инструментального средства выглядит следующим образом:

```
Converter [список ключей].
```

Входные параметры задаются при помощи следующих ключей:

- `-s <путь к файлу>` – задать путь к *XML*-описанию автоматной программы. Если данный параметр отсутствует, то будет использоваться значение по умолчанию `A1.xml`.
- `-m <path>` – задать путь к файлу, в который будет сохранена построенная модель. Если данный параметр отсутствует, то будет использоваться значение по умолчанию `model.ltl`.
- `-r <path>` – задать путь к файлу, в который будет записан отчёт о проведённой верификации. Если данный параметр отсутствует, то будет использоваться значение по умолчанию `report.txt`.
- `-f <formula>` – задать LTL-формулу. Если параметр отсутствует, то верификация не проводится.
- `-h` – вывести справку и завершить работу.

3.6.2.3. Обратное преобразование контрпримера

Инструментальное средство *Converter* на выходе выдает текстовый файл с отчетом, в котором содержатся полные сведения о проведённой верификации. В начале отчета содержатся общие сведения о сгенерированной *модели Крипке*, о режиме работы верификатора, об использованной памяти, о числе ошибок в модели (если они есть) и т. д. Если модель не удовлетворяет LTL-формуле, в отчете будет следующая запись:

```
pan: claim violated!
```

Если модель не содержит ошибок, то в отчете будет строка следующего вида:

```
State-vector 24 byte, depth reached 0, errors: 0.
```

Верификатор *Spin* выдает контрпример в виде пути в модели, описанной на языке *Promela*. Опишем, как из него вернуться к автоматной программе.

Строка отчета

```
StateAk s : <имя состояния>
```

(*stateAk* – состояние автомата *Ak*, *s* – номер состояния) обозначает вход автомата *Ak* в состояние *s*.

Строка отчета

```
Going to StateAk s : <имя состояния>
```

(*stateAk* – состояние автомата *Ak*, *s* – номер состояния) обозначает переход автомата *Ak* из текущего состояния в состояние *s*.

Строка отчета

```
Event = exx
```

(*exx* – некоторое событие) обозначает, что был совершен переход по событию *exx*.

Строка отчета

```
Action oi.zj
```

обозначает, что было вызвано выходное воздействие *zj* объекта управления *oi*.

Таким образом, строится путь в автоматной модели из файла отчета, выданного инструментом *Converter*.

Основным ограничением данного метода является отсутствие поддержки многопоточных систем. Поэтому был разработан усовершенствованный метод верификации.

3.7. Описание инструментального средства Stater

3.7.1. Описание интерфейса *Stater*

Инструментальное средство *Stater* предназначено для того, чтобы визуально разрабатывать автоматные программы и подпрограммы. Возможности *Stater*:

- создавать параллельные системы конечных автоматов;
- импортировать системы автоматов из *Stateflow* и стандарта *IEC 61499*.
- генерировать программный код на языке программирования *C#*;
- проводить верификацию системы конечных автоматов методом проверки моделей.

Для работы *Stater* требуется *.NET Framework* версии 4.0 или выше.

Генератор кода и верификатор реализованы в виде дополнений к основному модулю инструментального средства.

Верификация в *Stater* проводится при помощи дополнения *SpinVeriff*. Для работы верификатора *SpinVeriff* требуется наличие *Spin* версии 6.23 и компилятора *gcc* (например, из пакета *MinGW*), а также путей к ним, прописанных в переменной окружения *PATH* из ОС *Windows*.

На рисунке 3.7.1.1 изображен внешний вид инструментального средства *Stater*.

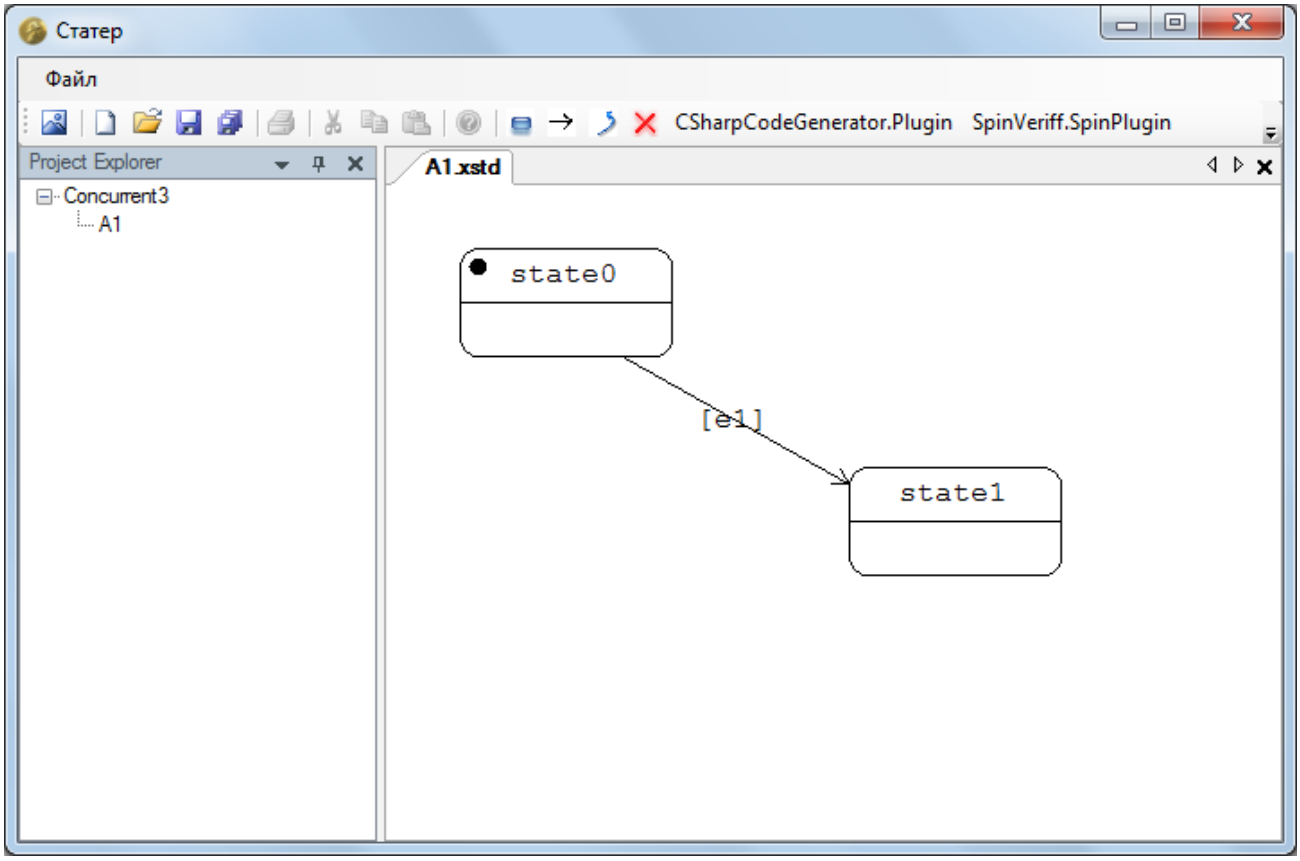

Рисунок 3.7.1.1. Интерфейс *Stater*

Создание системы конечных автоматов

Создание системы конечных автоматов рассмотрим на примере.

Для того чтобы разработать систему конечных автоматов, требуется создать проект, в котором находиться будет система конечных автоматов. Для этого в меню «Файл» необходимо выбрать пункт «Новый», а далее – «Проект». Появится диалоговое окно «Новый проект» (рисунок 4.3.1.2). В нем требуется выбрать местоположение проекта и его название и нажать кнопку «ОК».

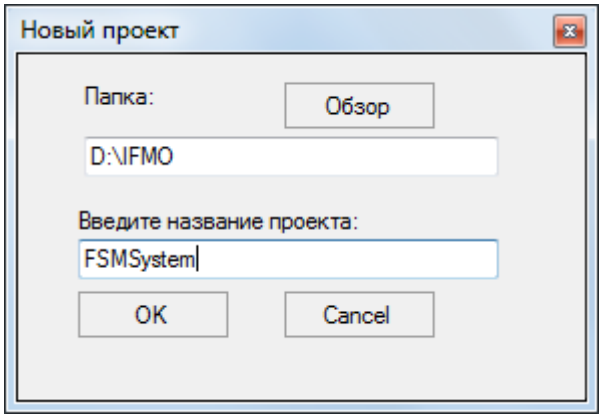

Рисунок 3.7.1.2. Новый проект

В результате будет создана папка с названием проекта, а в ней – файл `<имя_проекта>.stp`. При последующих открытиях *Stater* для работы с этим проектом надо в меню «Файл» выбрать пункт «Открыть», а в нем выбрать «Проект». В открывшемся диалоговом окне выбирается файл проекта. В примере на рисунке 3.7.1.2 проект называется *FSMSystem*.

Далее создаем диаграмму переходов автомата (рисунок 3.7.1.3).

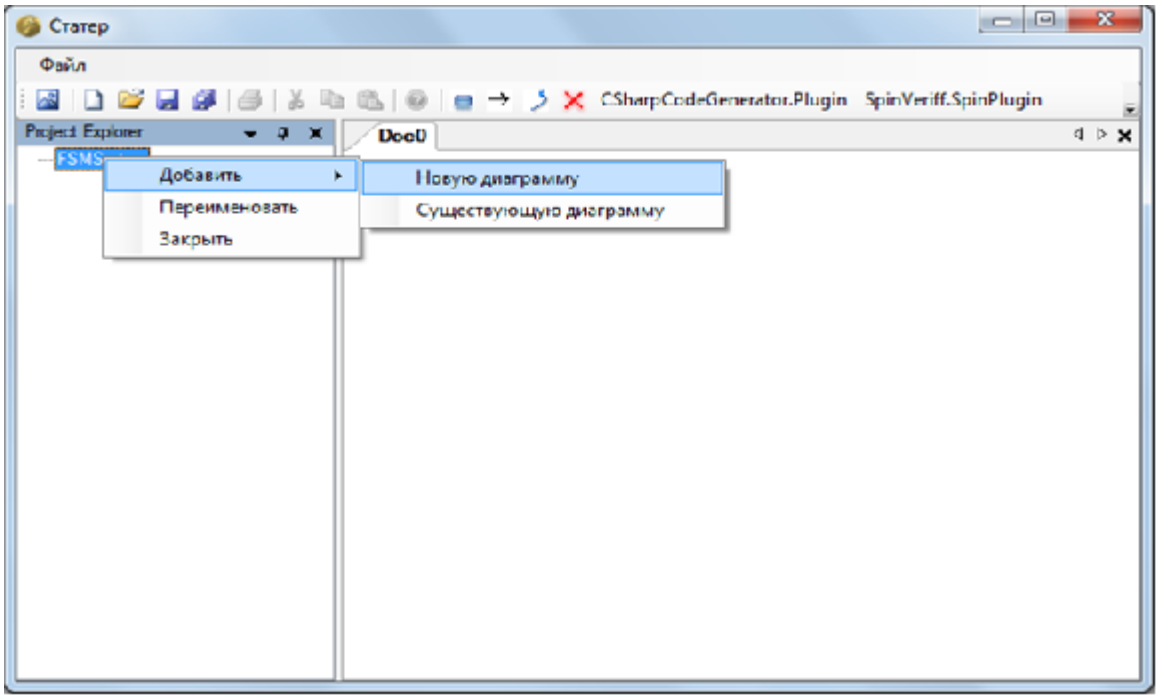

Рисунок 3.7.1.3. Новая диаграмма

В диалоговом окне «Новая диаграмма» введём название автомата *A1* и отметим кнопку «Конечный автомат» (рисунок 3.7.1.4). Если говорить в

терминах объектно-ориентированного программирования, то конечный автомат является классом, а не объектом. Таким образом, может быть несколько экземпляров одного конечного автомата могут быть реализованы одним классом.

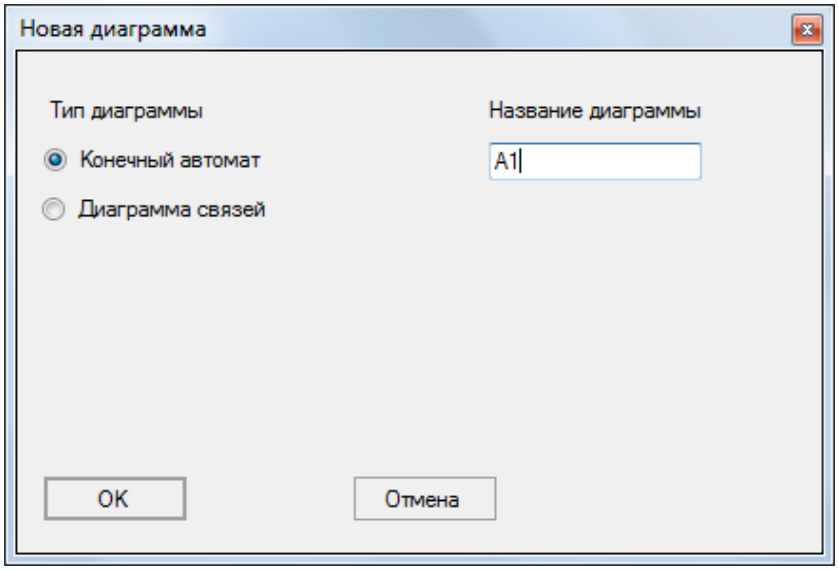

Рисунок 3.7.1.4. Новая диаграмма – продолжение

В результате диаграмма автоматного класса (графа переходова) появится в новой вкладке в клиентской части окна инструментального средства *Stater*.

Для того чтобы создать состояние, требуется нажать кнопку «Состояние» на панели инструментов, а после этого кликнуть в том месте, где на диаграмме должно располагаться состояние (рисунок 3.7.1.5). Состоянию автоматически присвоится имя.

Аналогично создадим еще одно состояние.

Для того чтобы выделить состояние или переход, по нему надо щелкнуть левой кнопкой мыши. Для того чтобы передвинуть состояние, требуется его выделить и после этого выполнить перетаскивание на требуемое место расположения состояния на диаграмме. Для того чтобы удалить состояние или переход, надо его выделить и нажать на кнопку «Удалить»

(отображается в виде красного креста). Рядом с ней находится кнопка отмены последнего действия.

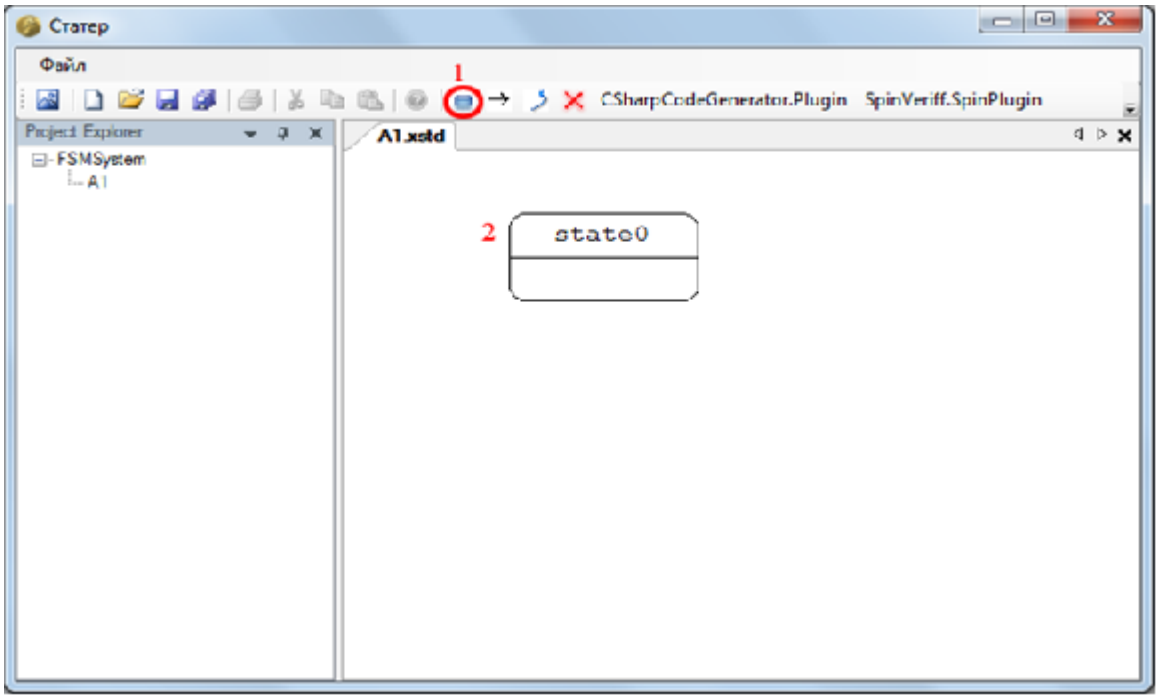

Рисунок 3.7.1.5. Создание состояния

Создадим переход между этими состояниями (рисунок 3.7.1.6). Для этого требуется нажать на кнопку «Переход» на панели инструментов, кликнуть на состояние-начало перехода и на состояние-конец перехода. Начало перехода может совпадать с концом перехода. В таком случае переход будет петлей.

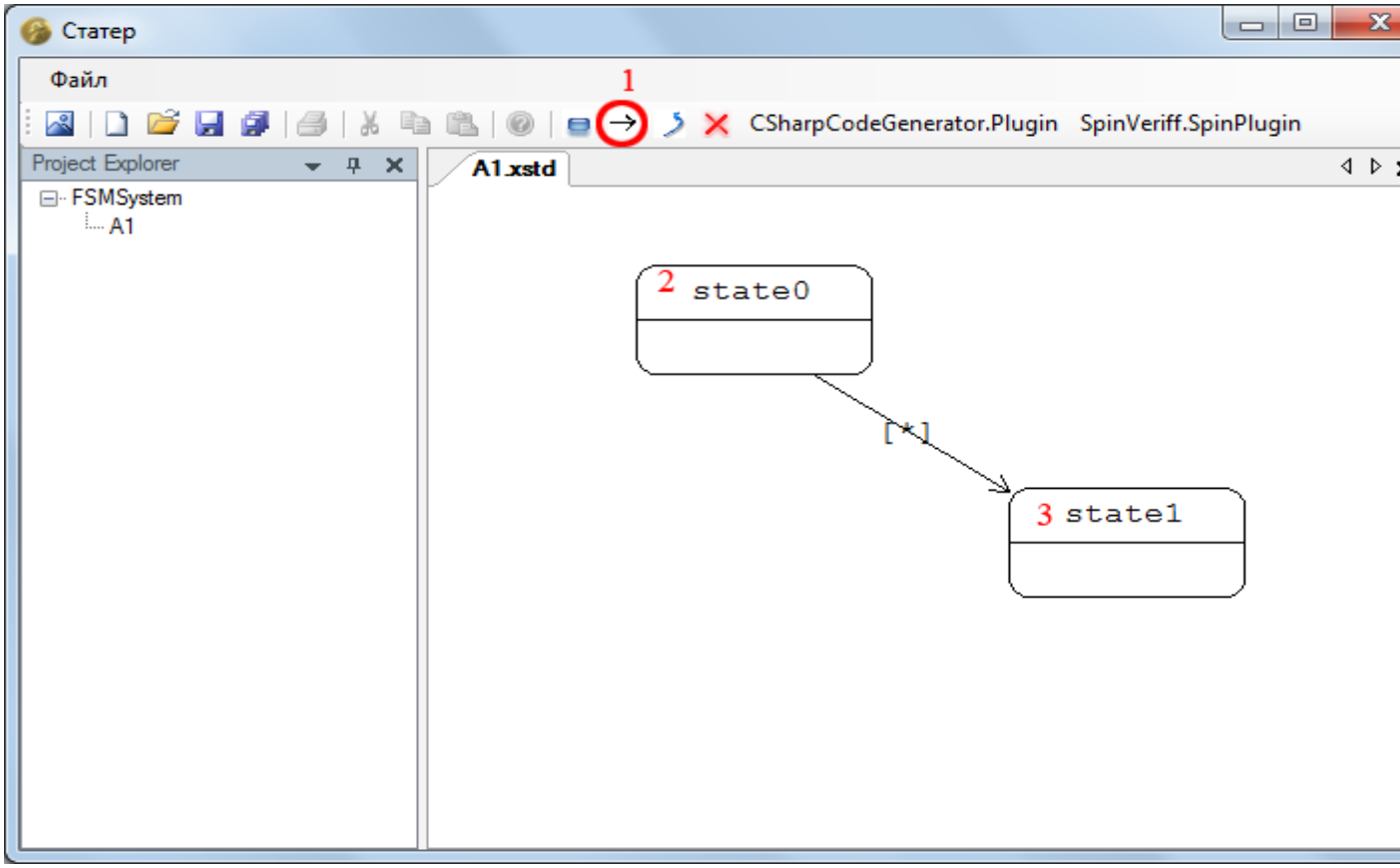

Рисунок 3.7.1.6. Создание перехода

По двойному щелчку по состоянию или переходу вызывается редактор атрибутов (рисунок 3.7.1.7 и 3.7.1.8).

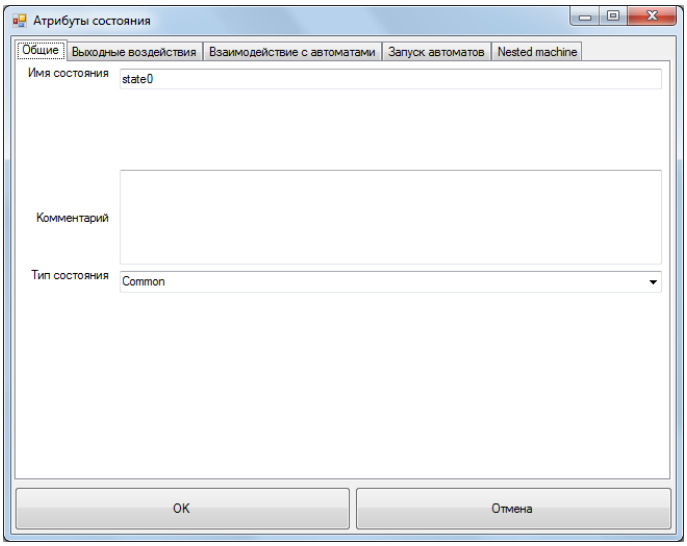

Рисунок 3.7.1.7. Атрибуты состояния

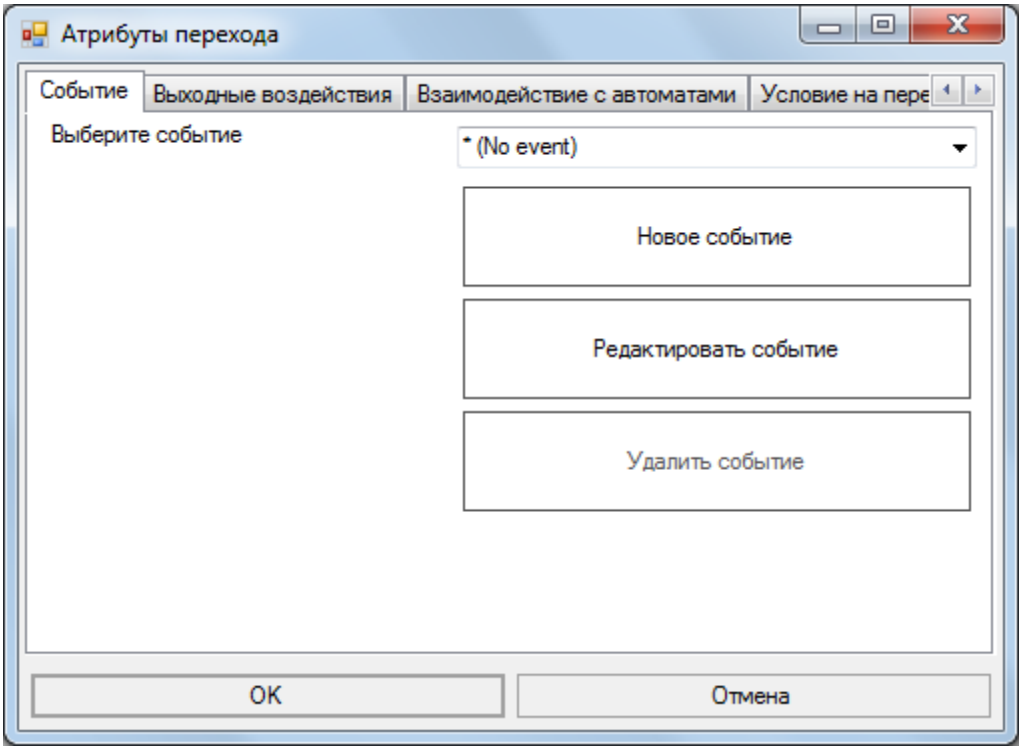

Рисунок 3.7.1.8. Атрибуты перехода

Имя состояния также является его идентификатором в сгенерированном программном коде и в модели для верификации. Таким образом, для имени состояния действуют все ограничения целевых языков программирования и языка *Promela*. Любое состояние может быть одного из трех типов: начальное (`start state`), обычное (`common state`) и конечное (`end state`). Начальное состояние помечается точкой. Конечное состояние помечается точкой внутри окружности. Начальное состояние в автомате может быть только одно. Остальные атрибуты будут описаны далее.

Состояние `state0` отметим как начальное.

Откроем двойным щелчком редактор атрибутов созданного перехода (рисунок 3.7.1.8). Переход помечается событием, по которому он выполняется. Событие выбирается из списка «Выберите событие». В нём указаны все события из всех автоматов проекта.

Для создания события требуется нажать на кнопку «Новое событие». Откроется диалоговое окно «Редактирование события» (рисунок 3.7.1.9).

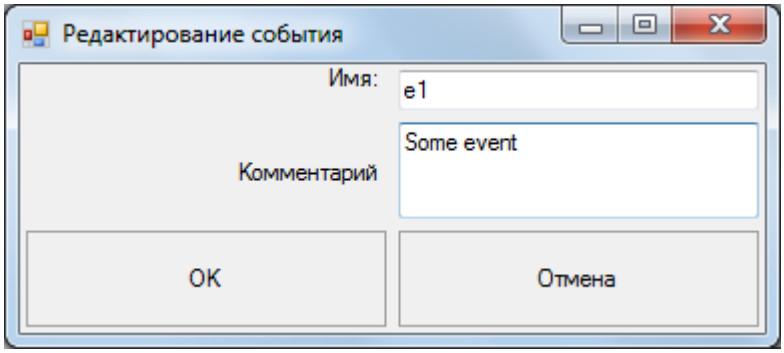

Рисунок 3.7.1.9. Создание события

На имя события, так же как и на имя состояния, накладываются все ограничения для идентификаторов переменных во всех целевых языках программирования.

Комментарий может в себя включать любые символы, а также может быть пустым.

Создадим выходное воздействие. Для этого перейдем во вкладку «Выходные воздействия» окна «Атрибуты перехода» и введём имя и описание выходного воздействия (рисунок 3.7.1.10).

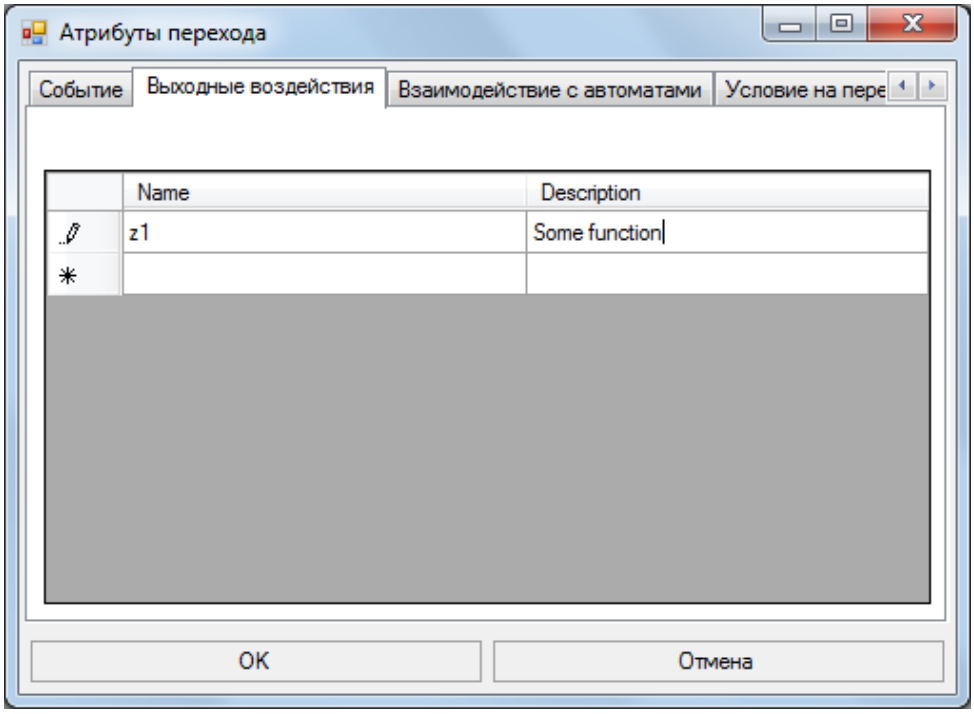

Рисунок 3.7.1.10. Выходные воздействия

Отметим состояние `state1` как конечное. Добавим логическую переменную `x1` в автомат. Для этого требуется щелкнуть правой кнопкой мыши по диаграмме автомата и выбрать в открывшемся контекстном меню пункт «Переменные». В диалоговом окне введём следующий текст:

```
bool x1;
```

Получаем автомат (рисунок 3.7.1.11).

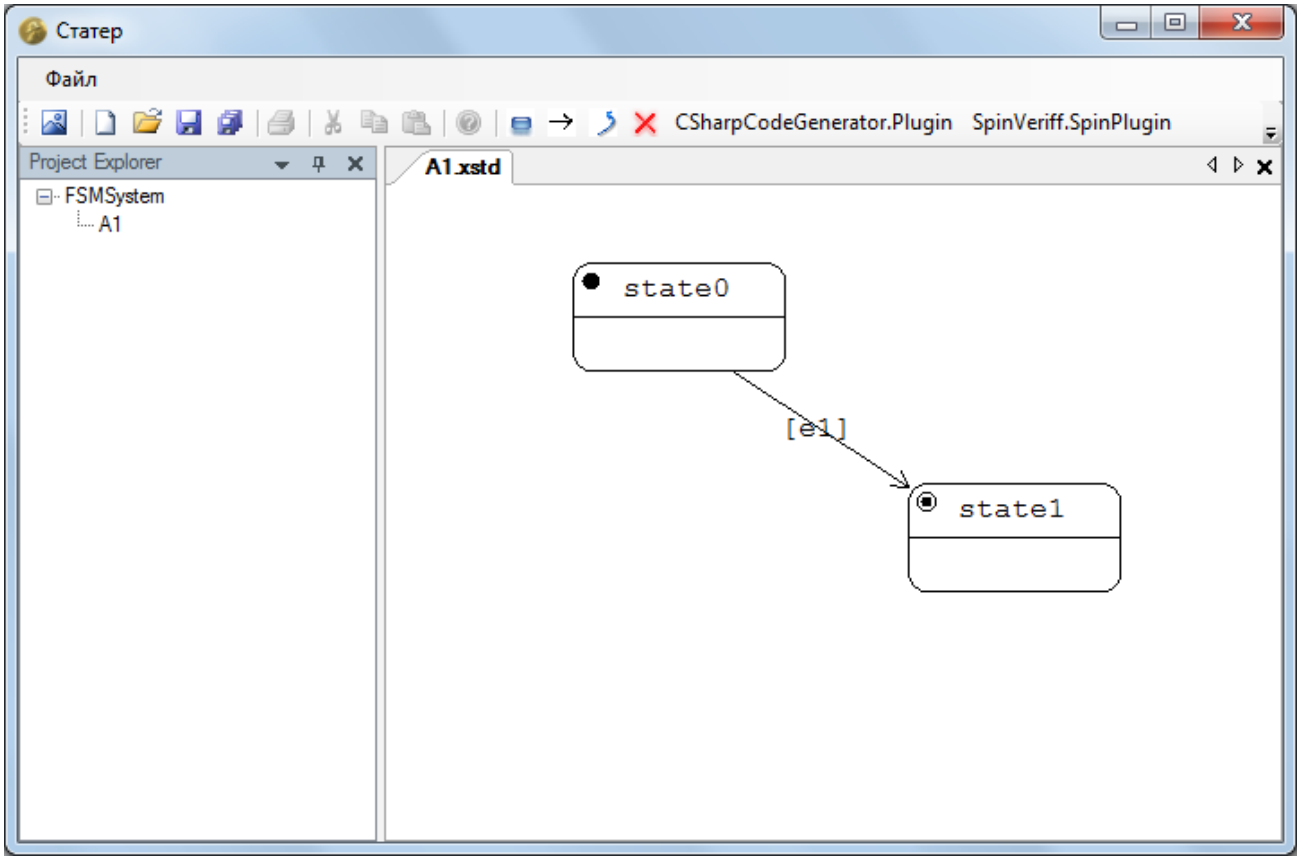

Рисунок 3.7.1.11. Конечный автомат `A1`

Добавим в систему еще два автомата `A2` и `A3` и создадим в каждом из них стартовое состояние `state0`.

Атрибуты состояния

Редактор атрибутов состоит из следующих вкладок:

- Общие.
- Выходные воздействия.
- Взаимодействие с автоматами.

- Запуск автоматов.
- Вложенные автоматы.

На вкладке «**Общие**» можно редактировать имя состояния в поле «Имя», его описание в поле «Комментарий» и выбрать тип состояния.

На вкладке «**Взаимодействие с автоматами**» можно объявить отправку событий другим автоматам в системе. Нельзя отправлять события вложенным автоматам. Отметим, что одни экземпляры автомата могут быть вложенными, а другие нет. Для отправки события экземпляру автомата требуется ввести тип автомата, имя экземпляра и имя события. Экземпляры автоматов могут быть запущены другими автоматами либо внешней средой.

Вкладка «**Запуск автоматов**» позволяет запустить автомат в новом потоке. Экземпляр такого же класса автоматов запускать нельзя, так как это порождает бесконечную рекурсию. Кроме того, запрещено, чтобы состояние, в котором запускается автомат, принадлежало какому-либо циклу в графе переходов. Для того чтобы объявить запуск автомата, требуется ввести его тип и имя экземпляра. Тип автомата выбирается из существующих в проекте автоматов.

Вкладка «**Вложенные автоматы**» позволяет объявлять автоматы, вложенные в данное состояние. Вложенные автоматы такого же типа запрещены. Для объявления вложенного автомата требуется ввести его тип и имя экземпляра.

Атрибуты перехода

Редактор атрибутов перехода состоит из следующих вкладок:

- Событие.
- Выходные воздействия.
- Взаимодействие с автоматами.
- Условия на переходах.
- Выходные воздействия: код.

Вкладка «**Событие**» позволяет создать или выбрать событие, по которому будет происходить данный переход.

Вкладка «**Выходные воздействия**» позволяет вызвать функции, которые являются выходными воздействиями автомата. Генератор кода на целевом языке создаёт все объявления функций, использованных в каждом автомате, пользователю останется только заполнить тело каждой функции.

Взаимодействие с автоматами на переходах в настоящее время не поддерживается инструментальным средством.

Вкладка «**Условие на переходе**» позволяет накладывать условия на переменные автомата. В качестве условий можно задавать одновременно допустимые в целевых языках программирования и в языке *Promela*. Примеры условий:

- `flag == true`;
- `a1 > a2`.

Вкладка «**Выходные воздействия: код**» позволяет описывать выходные воздействия не в виде названий вызываемых функций, а в виде программного кода. Код должен быть допустимым одновременно во всех целевых языках программирования и в *Promela*.

Переменные

Инструментальное средство *Stater* поддерживает переменные и массивы (рисунок 3.7.1.12) следующих типов:

- *bool*;
- *int8*;
- *int16*;
- *int32*.

Переменные объявляются следующим образом:

```
[модификатор] <тип> <имя> = значение;
```

либо

```
[модификатор] <тип> <имя>;
```

Массивы объявляются так:

```
[модификатор] <тип> <имя> [число_элементов];
```

В случае если при объявлении значение не указано, переменные инициализируются значением по умолчанию (ноль для числовых типов и false для логического). Все элементы массивов также инициализируются по умолчанию.

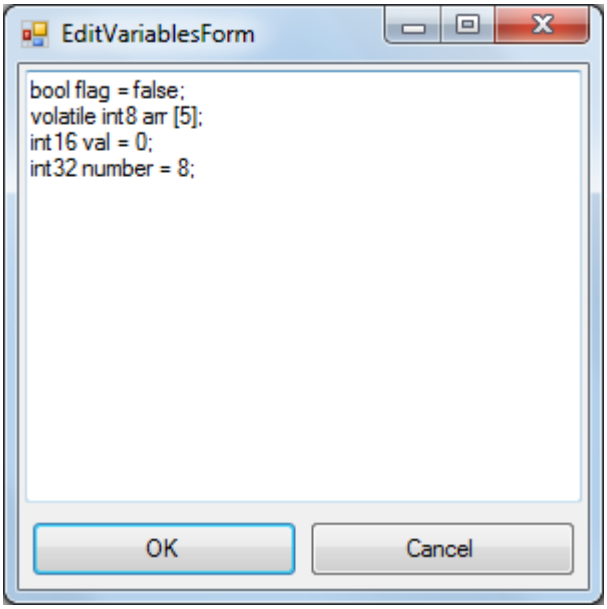

Рисунок 3.7.1.12. Окно ввода переменных

Описание флага *Autoreject* для выбора режима работы системы автоматов

Если на вход классическому детерминированному конечному автомату приходит символ, по которому из данного состояния нет перехода, то он переходит в недопускающее состояние. Однако для реактивных программ такое поведение неприемлемо. В конечных автоматах в данной работе входным языком является множество событий. Однако если в каждом состоянии явно прописывать переход по каждому возможному событию, то графическое отображение потеряет свою наглядность. Поэтому по умолчанию у каждого состояния есть неявные пустые переходы в то же состояние (петли). При необходимости можно включить переход в недопускающее состояние по неизвестному событию. Для этого требуется щелкнуть правой кнопкой мыши

по диаграмме автомата и в контекстном меню отметить галочкой флаг *autoreject*.

Открытие проекта

Для того чтобы открыть ранее созданный проект, требуется в меню «Файл» выбрать пункт «Открыть», и в нем выбрать пункт «Проект», а после этого – файл проекта. Расширение файлов проекта – «`stp`».

3.7.2. Генерация кода

Для генерации кода на языке *C#* используется дополнение *CSMultyThread*. На рисунке 3.7.2.1 показаны настройки дополнения. В форме настроек требуется ввести имя пространства имён, в котором будут лежать все сгенерированные классы, а также экземпляры автоматов, которые будут сразу запущены при старте программы, для которой этот код и генерируется. Весь код генерируется в папке проекта.

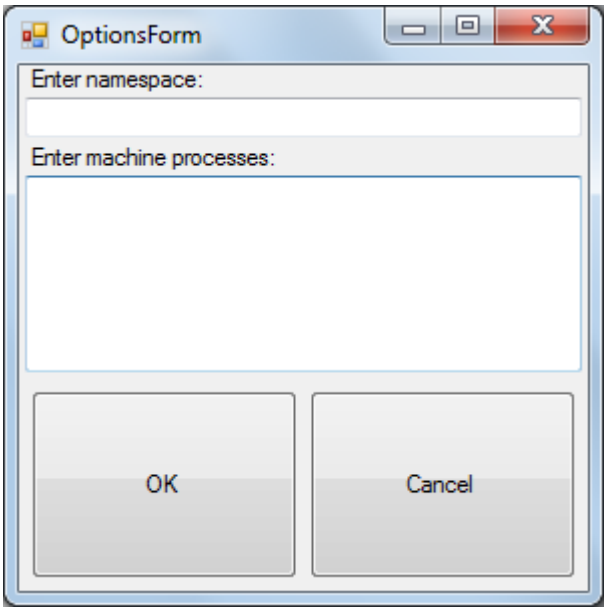

Рисунок 3.7.2.1. Настройки *CSMultyThread*

Для каждого типа автомата генерируется класс в отдельном файле. При повторной генерации кода будет перезаписана только часть файла, которая находится внутри региона «`~~~~~~Generated functions~~~~~~`». Кроме того, в файл будут добавлены недостающие объявления методов, которые являются выходными воздействиями автомата. Объявление класса не

перезаписывается, поэтому сгенерированный класс можно наследовать и определять конструкторы с параметрами. Всё это не будет перезаписано при повторной генерации кода. Можно определять свои члены класса вне региона «~~~~~~Generated functions~~~~~~», и генератор кода не будет их перезаписывать.

Все события записываются в перечисление (*enum*) в файле `Events.cs`.

Для обеспечения многопоточности генерируются классы `EventQueue` и `ThreadManager`.

Сгенерированный код не является завершенной программой. Для его использования требуется включить в свой проект сгенерированные классы и перечисление `Events.cs`. Для обработки событий у каждого класса есть два публичных метода `ProcessEvent`:

- на вход принимаются события из перечисления `Events.cs`;
- на вход принимаются строки с именами событий.

Генератор кода пишет в файл объявления всех методов класса, которые являются выходными воздействиями автомата. Тело каждого метода пользователь пишет самостоятельно. Генератор кода при повторном вызове не перезаписывает такие методы. Если генератор находит выходные воздействия, которые не были определены, он дописывает их определения. Кроме того, пользователь может определять собственные методы такого класса.

Внутрь региона «~~~~~~Generated functions~~~~~~» ничего дописывать не рекомендуется, так как при повторной генерации кода оно будет удалено.

3.7.3. Верификация

Для верификации предназначено дополнение *SpinVeriff*. Верификация проводится при помощи инструментального средства *Spin*. Для этого, как описано в разделе 3.2, строится модель распределенной системы автоматов на языке *Promela*. На рисунке 3.7.3.1 показаны настройки дополнения *SpinVeriff*.

В поле «*LTL formulae*» вводятся одна или несколько LTL-формул (по одной на каждую строку). Формулы имеют формат *Spin* за исключением того, что атомарные высказывания записываются в фигурных скобках.

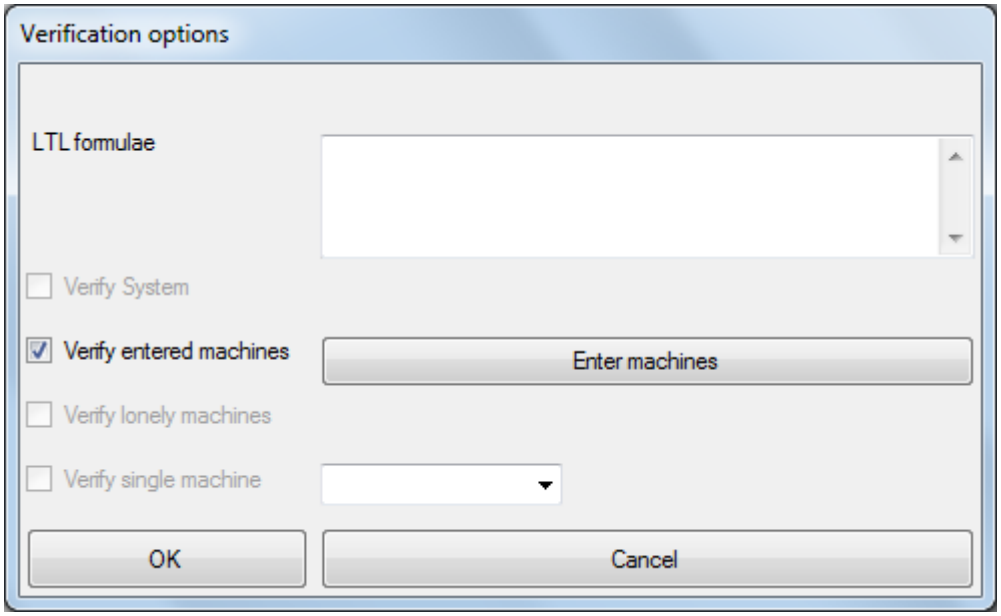

Рисунок 3.7.3.1. Настройки верификатора

Атомарные высказывания могут быть следующих видов:

1. `machineName.exx` – в автомат *machineName* пришло событие *exx*.
2. `machineName.sxx` – автомат *machineName* перешел в состояние *sxx*.
3. `machineName.Function` – автомат *machineName* вызвал выходное воздействие *Function*.
4. `machineName->nestedMachine` – автомат *machineName* дошёл до вложенного автомата *nestedMachine*.
5. `machineName||forkMachine` – автомат *machineName* запустил автомат *forkMachine*.
6. `machineName.variable OP value` сравнивает переменную или элемент массива экземпляра `machineName` автомата `MachineType` со значением value. `OP` – оператор сравнения (>, <, ==, !=, >=, <=).

Кнопка «*Enter machine*» открывает окно, в котором требуется ввести все запущенные внешней средой автоматы.

После верификации *SpinVeriff* записывает результат в файл `report.txt` в папке проекта и открывает текстовый редактор *Note Pad* с этим файлом.

Построенная модель находится в файле `spinModel.ltl` в папке проекта.

3.7.4. Архитектура *Stater*

Stater состоит из следующих компонентов (рисунок 3.7.4.1):

- `Core` – ядро системы.
- `Plug-ins` – расширения системы.
- `Plugin data` – данные, которые передаются из ядра в расширения и обратно.
- `Text processor` – библиотека для работы с текстом.

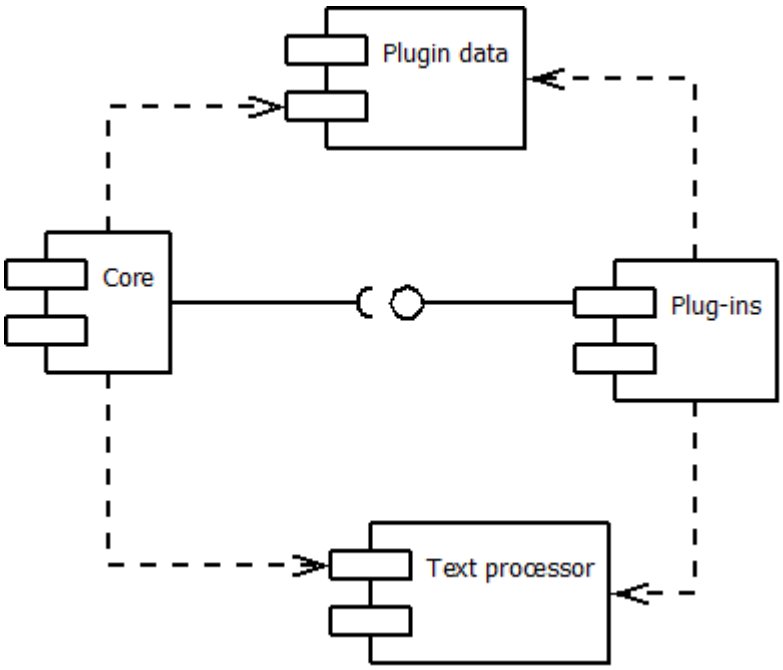

Рисунок 3.7.4.1. Диаграмма компонентов

Ядро системы содержит следующие компоненты:

- Оконная система.
- Система управления проектами.
- Система управления расширениями.
- Система команд.

Оконная система (рисунок 3.7.4.2) включает в себя всё, что относится к отображению диаграмм, а также диалоговые окна с настройками и свойствами элементов диаграмм. Она спроектирована так, чтобы можно было отображать любые диаграммы, однако на данный момент реализована только диаграмма переходов.

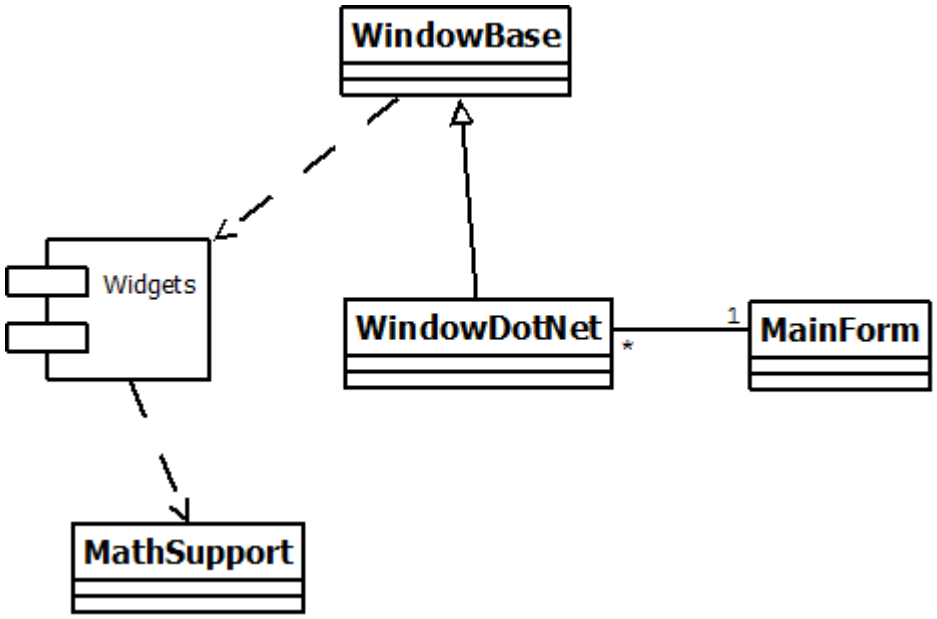

Рисунок 3.7.4.2. Оконная система

За основу отображения элементов диаграмм взята система *виджетов* [82].

Описание основных классов

- `WindowBase` – абстрактный класс, который содержит одну диаграмму и отвечает за её отображение.
- `WindowDotNet` – наследник `WindowBase`, который содержит реализацию для библиотеки *Windows Forms*.
- `Widget` – базовый класс для всех *виджетов*.
- `MathSupport` – математическая библиотека для графического интерфейса.
- `MainForm` – главное окно.

3.8. Верификация параллельной системы автоматов, управляющих гусеничным шасси

Продемонстрируем предложенный метод на примере верификации **системы** управляющих автоматов для гусеничного шасси.

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` (рисунок 3.8.1) управляют соответственно левым и правым двигателями.

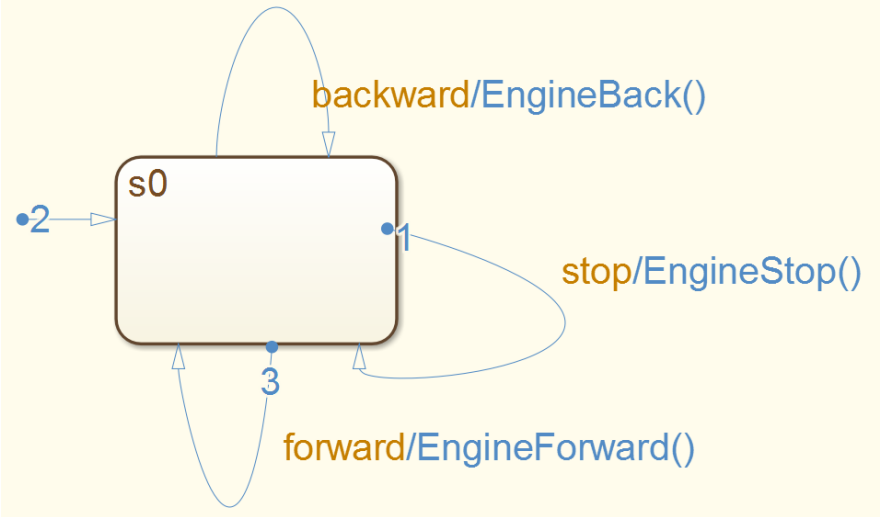

Рисунок 3.8.1. Граф переходов автоматного типа `AEngine`

Автомат типа `AManager` (рисунок 3.8.2) отправляет команды на управление двигателями в зависимости от команд для управления шасси.

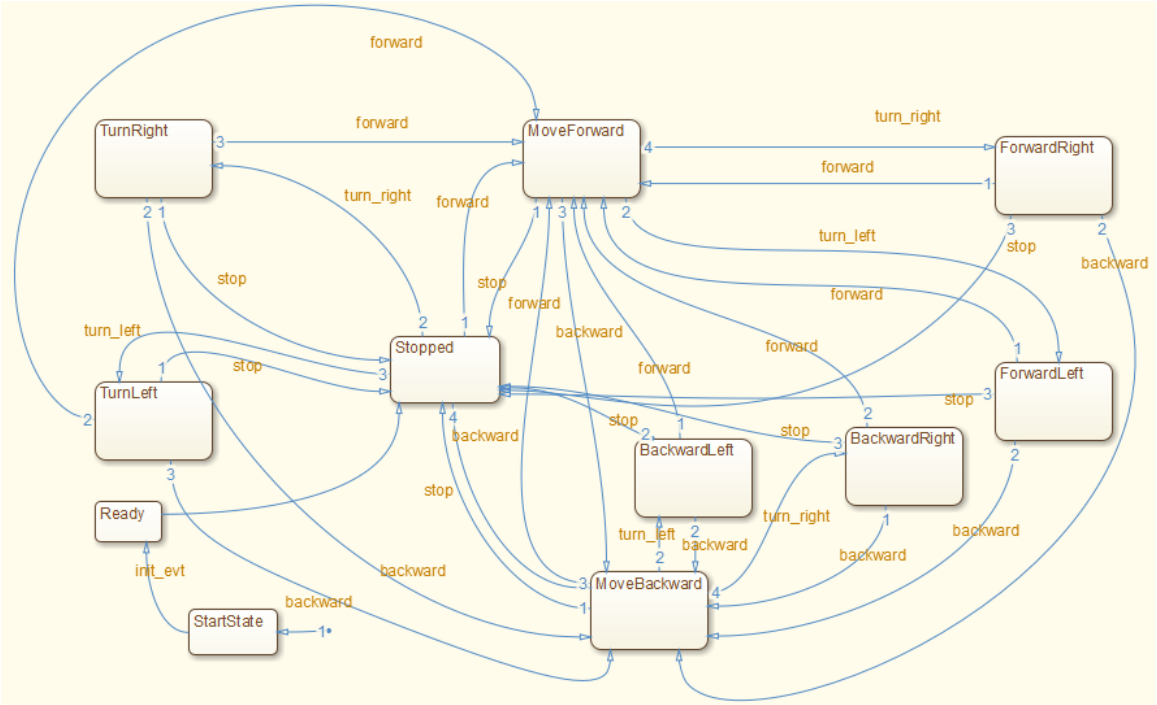

Рисунок 3.8.2. Граф переходов автоматного типа `AManager`

При входе в состояния этот автомат 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.`

Проверим свойство: «**В любой момент, если поступила команда «стоп», подается команда на остановку левого двигателя**». Отметим, что нет возможности проверить, что двигатель остановился, так как это

утверждение относится к аппаратной части, что в диссертации не рассматривается.

Формализуем указанное свойство. Высказывание «Поступила команда «стоп» 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)

Получаем следующую формулу:

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

Отметим, что в соответствии с синтаксисом языка *Promela*, оператор **G** (всегда) обозначается как `[]`, а оператор **F** (в будущем) – <>.

Запустив инструментальное средство *Stater* и выполнив верификацию, получим ответ, который означает, что верифицируемое свойство выполняется в построенной системе:

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

Выводы по главе 3

1. Построены математические модели управляющего автомата, системы вложенных автоматов и системы параллельно работающих автоматов.
2. Изложены предлагаемые алгоритмы построения *Spin*-моделей управляющего автомата, системы вложенных автоматов и системы параллельно работающих автоматов.
3. Изложено предлагаемое в работе расширение LTL-формул и их преобразование к *Spin*-моделям.

4. Изложен предлагаемый метод верификации автоматных программ.
5. Описаны разработанные автором инструментальные средства для поддержки предложенного метода.
6. Применение предложенного метода продемонстрировано на примере верификации системы автоматов, управляющих гусеничным шасси.

Глава 4. Внедрение

4.1. Инструментальное средство Converter

В рамках работ по государственному контракту [65 – 68] была проведена апробация инструментального средства *Converter*. Работоспособность этого средства проверялась на клиент-серверной модели банкомата. Эта модель состоит из двух **вложенных** автоматов: *AClient*, содержащий 13 состояний, и вложенный в него *AServer*, содержащий пять состояний. Были формально доказаны истинные свойства банкомата. Также были сформулированы свойства, которые не должны выполняться в банкомате, что и подтвердила верификация. Верификация модели банкомата описана в [5].

Также *Converter* и метод, на котором основано это средство, используется в учебном процессе. С 2012 года автором на кафедре «Компьютерные технологии» ведётся курс по верификации программного обеспечения. В рамках данного курса студенты в частности проводят верификацию автоматных программ, выложенных на сайте http://is.ifmo.ru. В 2012 году магистрант В. Ульянцев обнаружил ошибку в программе, реализующей управление часами с будильником, которая до этого многократно проверялась [83].

4.2. Инструментальное средство Stater

Начиная с 2014 года *Stater* используется вместо *Converter* в учебном процессе на кафедре «Компьютерные технологии». Функциональность *Stater* позволяет упростить верификацию, так как в нём можно сначала строить системы автоматов, а после этого их верифицировать.

4.2.1. Загрузка из XML-файла

Инструментальное средство *Stater* было использовано при разработке программы *AntsysEditor* в ООО «СТЦ». *AntSysEditor* предназначен для редактирования информации о структуре антенн, которая хранится в XML-файлах в следующем формате:

1. Внешний тег <antsys> содержит все остальные и сообщает номер версии соответствующей программы, с помощью которой данный XML-файл был создан. Этот тэг только один в файле.
2. Второй по вложенности тег <kind> содержит общую информацию для всей антенны: Index, id, Kind, ASName, BegFreq, EndFreq, Roll, Pitch, Yaw, Latitude, Longitude и Height. Бывает только один в файле.
3. Самым глубоким по вложенности тегом, является тег <ae>, который содержит информацию непосредственно о конкретном антенном элементе: id, x, y, z и phase. Число таких тегов равно числу используемых антенных элементов.

Верификация осуществлялась для автомата ALoader, который реализует следующий алгоритм обработки XML-файлов данного формата:

1. Если приходит тег <kind>, то автомат переходит в состояние _Kind_.
2. Далее обрабатываются все поля тэга. Например, если тэг содержит параметр Id, то вызывается функция SetId, которая устанавливает значение Id, и автомат переходит в состояние _Kind_
3. Если приходит тег <ae>, то вызывается функция создания нового объекта NewAE, и автомат переходит в состояние AE.
4. В состоянии AE обрабатываются поля, соответствующие данному тегу. Например: приходит параметр id, вызывается функция SetAEid, которая устанавливает значение id у объекта, и автомат переходит в состояние AE.
5. Если автомат находится в состоянии AE и приходит закрывающий тег </ae>, то вызывается функция добавления элемента AddAE, и автомат переходит в состояние _Kind_.

Автомат ALoader приведён на рисунке 4.2.1.1.

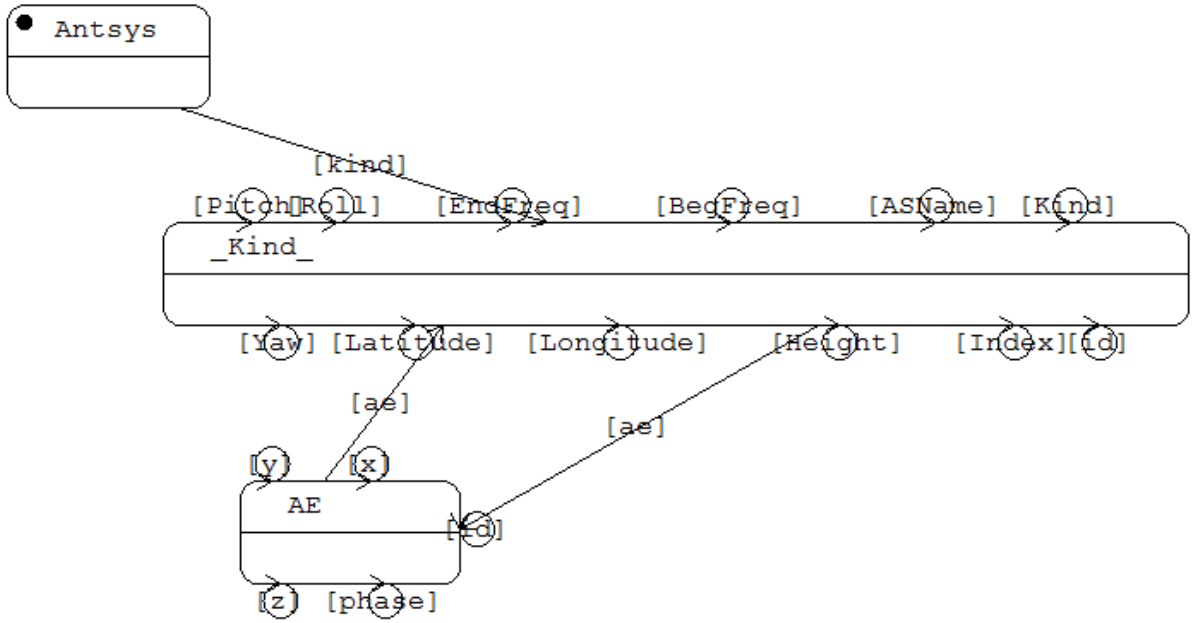

Рисунок 4.2.1.1. Автомат `ALoader`

Приведём LTL-спецификацию для этого автомата.

Формула *f0*. «Нельзя обрабатывать тег <ae>, пока нет тега <kind>».

```
(<> {loader.AE}) -> (!{loader.AE} U {loader.kind})
```

Формула *f1*. «Если объект `AE` создан, то нельзя обрабатывать следующий тег до добавления объекта».

```
[](({loader._Kind_} && {loader.ae}) -> (!{loader._Kind_} U {loader.ae}))
```

Формула *f2*. «Если обрабатываем значение x объекта `AE`, то обязательно вызовется функция установки x». Можно написать аналогичные формулы для каждого элемента каждого тега.

```
[](({loader.AE} && {loader.x}) -> ({loader.SetAEX} U {loader.AE}))
```

После формализации спецификации была выполнена верификация. Построенная при помощи *Stater* *Spin*-модель и вывод *Spin* приведены в *Приложении А*. Из изложенного следует, что в рассмотренной задаче верифицируемые свойства выполнены.

4.2.2. Игра *Turtleball*

При помощи инструментального средства *Starter* проверифицирована также логика в игре «*Turtleball HD*». Смысл игры состоит в том, что прямоугольному полю, ограниченному стенами, катятся шарики, а игрок должен при помощи стен отсекать части поля от шариков. Стена растёт от места, по которуму кликнул игрок, в две стороны (либо вертикально, либо горизонтально), пока не дорастёт до другой стены. Шарик ударился о строящуюся стену, то игрок теряет «жизнь», а стена уничтожается. Замкнутая стенами часть поля, в которой нет шариков, заполняется. Цель игрока состоит в том, чтобы заполнить 70% поля.

Автомат `AWall` описывает логику пересечения стен с шариками:

1. Изначально автомат находится в состоянии `Start`.
2. В зависимости от того, как растёт стена, вертикально или горизонтально, автомат переходит в состояние `GrowVertical` или `GrowHorizontal`.
3. Если автомат находится в одном из предыдущих состояний и ему приходит событие пересечения стены с шариком, то выполняется функция `KillAll`, и автомат переходит в состояние `Kill`. Если пришло событие, что стена установилась, то автомат переходит в состояние `Wall`.
4. В состоянии `Kill`, если приходит событие `reset`, то стена сбрасывается.

   Автомат `AWall` приведён на рисунке 4.2.2.1.

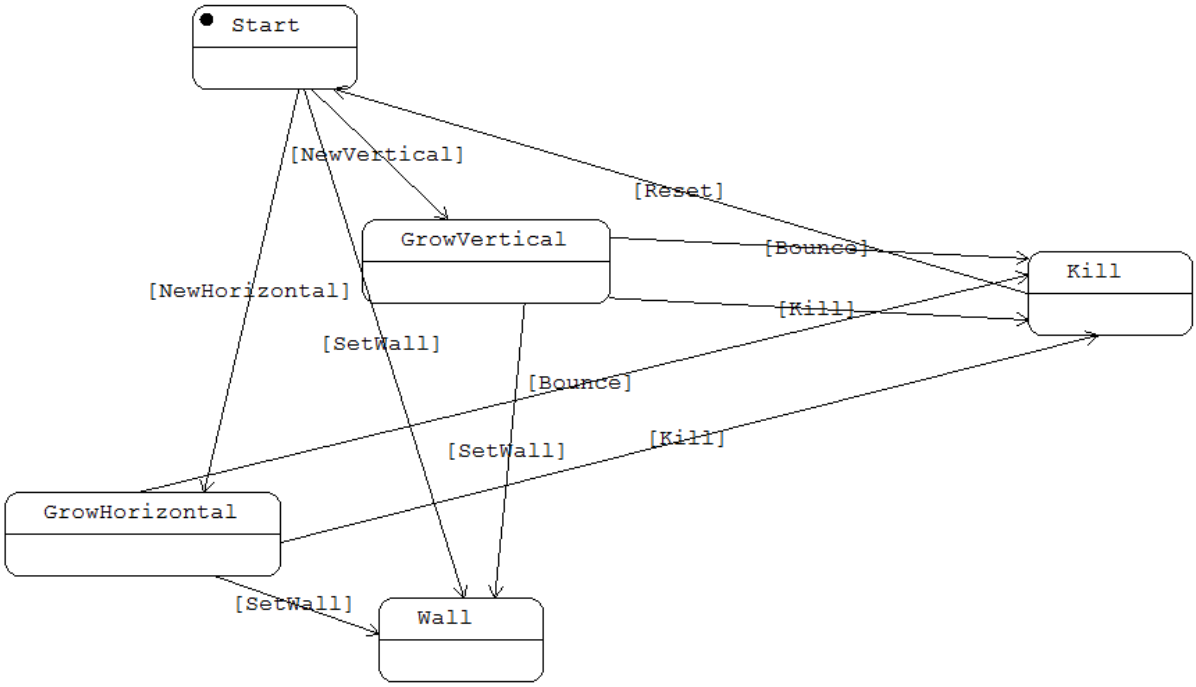

Рисунок 4.2.2.1. Автомат `AWall`

Приведём LTL-спецификацию для этого автомата

Формула *f0*. «Если шарик ударился о стену, то она когда-то начнёт строиться заново».

```
[] (({awall.Kill}) -> (<> {awall.Start}))
```

Формула *f1*. «Если шарик не ударился о стену, то она построится».

```
[](((!(<> {awall._Kill_})) -> (<> {awall.Wall}))
```

Автомат `APlatform` описывает логику роста обеих стен при нажатии на экран:

1. Изначально автомат находится в состоянии `Start`. Когда игрок кликает на экран, автомат переходит в состояние роста двух стен `Grow2Walls`.
2. Из состояния `Grow2Walls`, если ничего не произошло и приходит событие обновления, автомат переходит в состояние `Grow2Walls`, вызвав функции `DoGrowWallOne` и `DoGrowWallOther`. Если приходит событие остановки роста стены `StopOneSide` или `StopOtherSide`, то автомат переходит в состояние `GrowOtherWall` или `GrowOneWall`, соответственно.

3. Из состояния `GrowOtherWall` при событии обновления автомат переходит в состояние `GrowOtherWall`, вызвав функцию `DoGrowOtherWall`. Если приходит событие `StopOtherSide`, то автомат переходит в начальное состояние `Start`.
4. Из состояния `GrowOneWall` при событии обновления автомат переходит в состояние `GrowOneWall`, вызвав функцию `DoGrowOneWall`. Если приходит событие `StopOneSide`, то автомат переходит в начальное состояние `Start`.

Автомат `APlatform` приведён на рисунке 4.2.2.2.

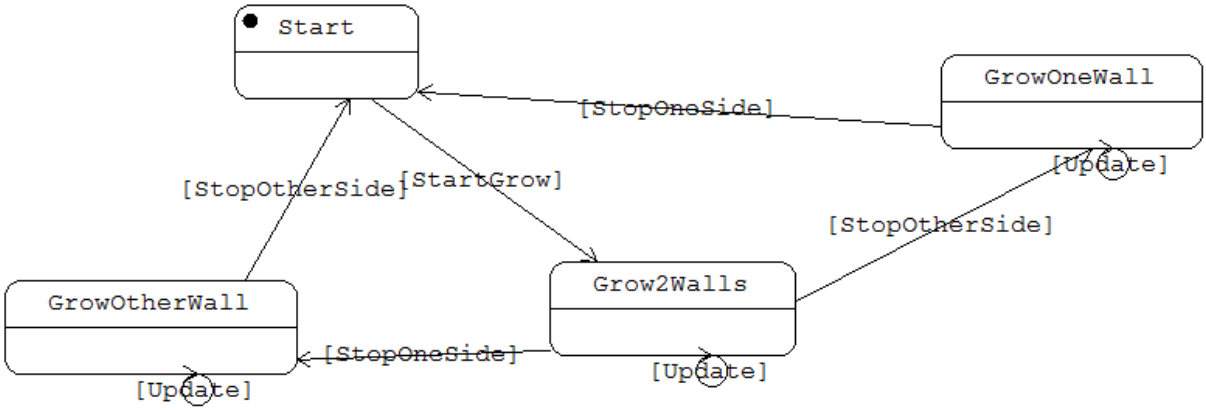

Рисунок 4.2.2.2. Автомат `APlatform`

Приведём LTL-спецификацию для этого автомата.

Формула *f0*. «Стены растут вместе либо бесконечно, либо будут расти до остановки роста одной из них».

```
[](({aplatform.Grow2Walls}) -> (({aplatform.Grow2Walls} U
({aplatform.GrowOneWall} || {aplatform.GrowOtherWall})) ||
[]({aplatform.Grow2Walls})))
```

Формула *f1*. «Если остановилась стена *One*, то далее будет расти только стена *Other*».

```
[](({aplatform.GrowOtherWall}) -> (({aplatform.GrowOtherWall} U
{aplatform.Start}) || []({aplatform.GrowOtherWall})))
```

Формула *f2*. «Если обе стены были когда-то остановлены, то автомат побывал в начальном состоянии».

```
((<> {aplatform.StopOneSide}) && (<> {aplatform.StopOtherSide}))
-> (<> {aplatform.Start})
```

Формула *f3*. «Если одна стена начала расти, то сначала росли обе».

```
(<> ({aplatform.GrowOtherWall} || {aplatform.GrowOneWall})) ->
(!({aplatform.GrowOtherWall} && {aplatform.GrowOneWall}) U
{aplatform.Grow2Walls})
```

После формализации спецификации была выполнена верификация. Посроенная при помощи *Stater Spin*-модель и вывод *Spin* приведены в *Приложении Б*. Из изложенного следует, что в рассмотренной задаче верифицируемые свойства выполнены.

4.2.3. Программа *Memory cards* для запоминания иностранных слов

Инструментальное средство *Stater* было также использовано автором для написания коммерческого программного обеспечения *Memory cards* для обучения иностранному языку. Оно доступно по адресу http://prognotes.ru/glubina/memory-cards/. Эта программа является электронной версией карточек для запоминания иностранных слов (на одной стороне карточки записано слово, а на другой перевод). Будем верифицировать автомат `Mnemo`, который реализует алгоритм показа карточек. Приведём описание этого алгоритма:

1. Вся колода карточек делится на несколько пачек (без участия автомата).
2. Пользователю показывается первая карточка функцией `GotoFirstCard` (автомат вызывает эту функцию, но она определена вне автомата).
3. Показ карточки осуществляется следующим образом:
    3.1. Показывается лицевая сторона карточки. По нажатию на кнопку «Перевернуть», карточка переворачивается, и показывается обратная сторона. После этого пользователь решает, правильно он вспомнил слово, или ошибся, и нажимает, соответственно кнопку «Верно» или «Неверно». После этого пользователю

показывается следующая карточка функцией GetNextCard (автомат вызывает эту функцию, но она определена вне автомата).

4. Для каждой пачки карточек:
   4.1. Первый проход: запоминание. Пачка перемешивается, и показываются по очереди все карточки из пачки. Когда пачка заканчивается, в автомат приходит событие `pack_is_over`. Определение конца пачки происходит вне автомата.
   4.2. Второй проход: повторение. Пачка перемешивается, и показываются все карточки по очереди. При этом если пользователь отметил, что он ошибся на некоторой карточке, то она возвращается обратно в пачку.
   4.3. Третий проход аналогичен второму, но все карточки показываются с обратной стороны.
   4.4. Осуществляется переход к следующей пачке.
5. Алгоритм заканчивает работу и переходит в начальное состояние, когда закончились пачки.

Автомат Mnemo приведён на рисунке 4.2.2.1.

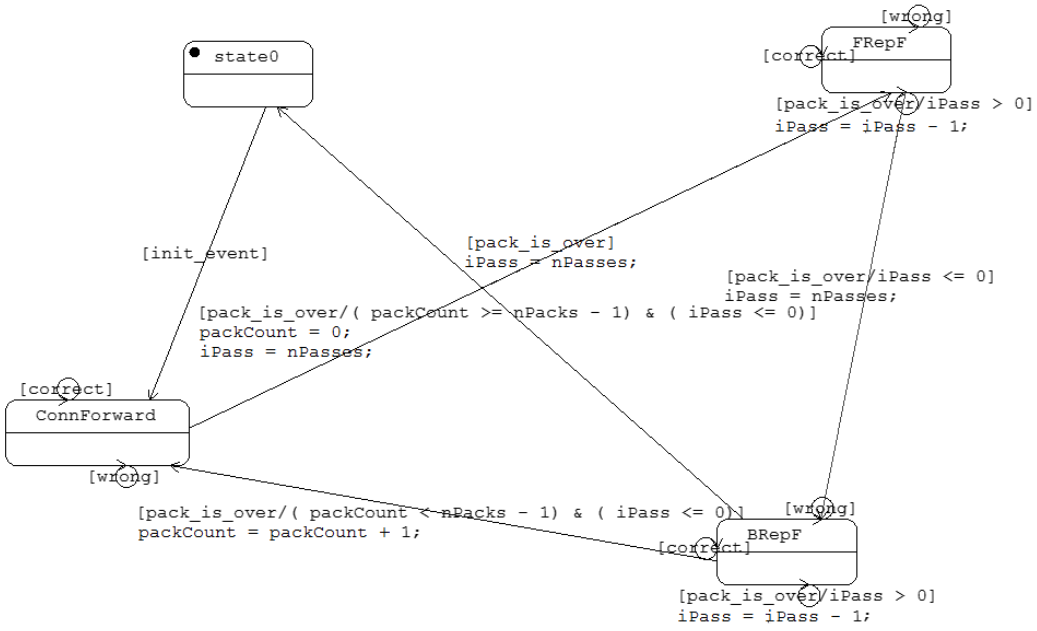

Рисунок 4.2.2.1. Автомат `Mnemo`

Автомат содержит следующие переменные:

- `param int8 nPacks = 0;`
- `param int8 nPasses = 4;`
- `int16 iPass = 0;`
- `int8 packCount = 0;`

Состояние `state0` – начальное состояние автомата. В этом состоянии автомат ждёт начала работы. Состояние `ConnForward` соответствует шагу 4.1 алгоритма. Состояние `FRepF` – шагу 4.2, а состояние `BRepF` – шагу 4.3. Переход из состояния `BRepF` в состояние `ConnForward` осуществляет шаг 4.4.

Приведём LTL-спецификацию для этого автомата.

Формула *f0*. «Если в процессе повторения пользователь отметил, что он не угадал слово на карточке, то эта карточка будет положена в оставшуюся часть пачки ещё раз до того, как будет выбрана следующая карточка из пачки». Повторение карточек происходит в состояниях `mnemo.FRepF` и `mnemo.BRepF`. «Пользователь отметил, что не угадал слово на карточке» – это событие `mnemo.wrong`. «Вернуть карточку в оставшуюся часть пачки» – `mnemo.ReturnCurCard`. «Выбрать следующую карточку» – `mnemo.GetNextCard`. В результате может быть записана следующая формула:

```
[] ( (({mnemo.FRepF} || {mnemo.BRepF}) && {mnemo.wrong} &&
(!({mnemo.ReturnCurCard})) && (!({mnemo.GetNextCard })))) ->
((!{mnemo.GetNextCard}) U {mnemo.ReturnCurCard}))
```

Формула *f1*. «Если пользователь отметил, что он угадал либо не угадал слово на карточке, то будет выдана следующая карточка:

```
[](({mnemo.wrong} || {mnemo.correct}) ->
(<>({mnemo.GetNextCard})))
```

Формула *f2*. «Алгоритм всегда сможет закончить работу и вернуться в состояние ожидания»:

```
[]<>{mnemo.state0}.
```

Эта формула **не должна выполняться**, так как вычисление конца пачки находится вне автомата, и если автомату не будет приходить событие `pack_is_over`, то он не сможет завершить работу. Это подтверждается результатом верификации (листинг 4.2.2.1), который выдаёт следующий **контрпример**: в автомат приходит событие `init_event`, автомат переходит в состояние `ConnForward`, после этого в бесконечном цикле в автомат приходит событие `correct`.

Листинг 4.2.2.1. Результат верификации свойства `[]<>{mnemo.state0}`

```
There are errors. Counterexample:
mnemosource sent _init_event_
mnemosource sent _init_event_
mnemo.nPacks = 255
mnemo.nPasses = 255
mnemo.iPass = 0
mnemo.packCount = 0
mnemo.state = Mnemo.state0
mnemo. event_happened: _init_event_
mnemo.LoadPack()
mnemo.GotoFirstCard()
mnemosource sent _init_event_
mnemo.nPacks = 255
mnemo.nPasses = 255
mnemo.iPass = 0
mnemo.packCount = 0
mnemo.state = Mnemo.ConnForward
mnemosource sent _correct_
mnemo.nPacks = 255
mnemo.nPasses = 255
```

```
mnemo.iPass = 0
mnemo.packCount = 0
mnemo.state = Mnemo.ConnForward
mnemo. event_happened: _correct_
mnemo.GetNextCard()
<<<<<START OF CYCLE>>>>>
mnemosource sent _correct_
mnemo.nPacks = 255
mnemo.nPasses = 255
mnemo.iPass = 0
mnemo.packCount = 0
mnemo.state = Mnemo.ConnForward
mnemo. event_happened: _correct_
mnemo.GetNextCard()
```

Формулы *f3 – f5*. Заменим формулу *f2* тремя другими формулами *f3 – f5*, которые должны выполняться:

f3: `[](({mnemo.ConnForward} && {mnemo.pack_is_over}) -> (<> {mnemo.FRepF}))`

f4: `[](({mnemo.FRepF } && {mnemo.pack_is_over} && (mnemo.iPass <= 0)) -> (<> {mnemo.BRepF}))`

f5: `[](({mnemo.BRepF } && {mnemo.pack_is_over} && (mnemo.packCount >= mnemo.nPacks - 1) && (mnemo.iPass <= 0)) -> (<> {mnemo.state0}))`

Формула *f6*. «Во время шагов 4.2 – 5 алгоритма значение `iPass` не превосходит `nPasses`».

```
[](({mnemo.FRepF} || {mnemo.BRepF}) -> mnemo.iPass <=
mnemo.nPasses)
```

После формализации спецификации была выполнена верификация. Из изложенного следует, что в рассмотренной задаче верифицируемые свойства выполнены.

Выводы по главе 4

1. В рамках работы по государственному контракту [65 – 69] была проведена апробация инструментального средства *Converter* на модели банкомата, содержащего два вложенных автомата с 13 и пятью состояниями. Были формально верифицированы как истинные свойства банкомата, так и построены контрпримеры для ложных свойств.
2. *Converter* и *Stater* используются в учебном процессе. С 2012 года автором на кафедре «Компьютерные технологии» ведётся курс по верификации программного обеспечения. В рамках его студенты провели верификацию автоматных программ, выложенных на сайте http://is.ifmo.ru. В одной из программ была найдена ошибка, несмотря на то, что она многократно проверялась до этого.
3. Инструментальное средство *Stater* было использовано в ООО «СТЦ» для верификации части программы *AntsysEditor*, позволяющей обрабатывать XML-файлы, которые содержат информацию о структуре антенн. Внедрение подтверждается актом о внедрении. Построенная при помощи *Stater* модель и вывод *Spin* приведены в Приложении А.
4. Инструментальное средство *Starter* было использовано для верификации логики в игре «*Turtleball HD*», что также подтверждается актом о внедрении. Построенная при помощи *Stater* модель и вывод *Spin* приведены в Приложении Б.

Глава 5. Подход к верификации программ в случае, когда модель внешней среды нельзя представить в виде автомата

В рассмотренных выше примерах, в частности, в модели управления гусеничным шасси, внешняя среда, которая формирует входные воздействия, из-за своей простоты отдельно не выделялась и задавалась при помощи LTL-формул.

В более сложных случаях внешняя среда может быть задана автоматами. Однако известны задачи (одна из них рассмотрена в настоящей главе), в которых и такое задание внешней среды невозможно, а без её учёта задача верификации не решается.

В данной главе в качестве внешней среды рассматривается клетчатое поле неограниченных размеров. При этом задача состоит в поиске пути агентом, содержащим *O(1)* памяти.

В работе [84] с помощью генетического программирования и частичной коэволюции с тестами было найдено 800 решений в виде одиночных конечных автоматов для задачи поиска пути агентом с *O(1)* памяти. Однако справедливость этих решений без верификации остаётся под сомнением. В настоящей главе доказывается корректность некоторых из полученных решений. Сформулируем решаемую задачу.

5.1. Формулировка задачи

Рассмотрим двумерное поле с препятствиями конечного размера. Цель – это точка, которая не принадлежит ни одному препятствию. Агент – это робот точечного размера, и который должен добраться до цели. Агент знает свои координаты, координаты цели, но имеет *O(1)* дополнительной памяти, поэтому он не может хранить карту поля с препятствиями. Также агент имеет датчик с малой функциональнотью, который позволяет агенту определять, столкнулся ли он с препятствием, и идёт ли он вдоль границы препятствия. В работе [85] было показано, что существует ряд алгоритмов (наиболее известные – *BUG-1* и *BUG-2*), которые всегда работают конечное время и находят цель, если она достижима, и сообщают, если она недостижима. В

работе [86] идеи алгоритмов семейства *BUG* были расширены для случая сенсоров, срабатывающих при достижением агента некоторого расстояния до препятствия.

В работе [84] рассмотрена дискретная версия этой задачи (рисунок 5.1.1), которая больше подходит для экспериментов с построением автоматных программ. В этой задаче поле представляет собой бесконечную клетчатую сетку. Каждая клетка поля либо свободна, либо содержит препятствие. Каждая связанная (по ребру или вершине) группа препятствий имеет конечный размер. Одна из свободных клеток назначается целью. Агент занимает целую клетку. Местоположение агента определяется декартовыми координатами и направлением (вверх, вниз, вправо, влево). Назовём соседней клетку, которая граничит по ребру с текущей. В алгоритмах *BUG-1* и *BUG-2* память объёмом *O(1)* используется для того, чтобы сохранить состояние (координаты и направление) агента в некоторый момент времени. Логика агента закодирована в автомате согласно парадигме автоматного программирования.

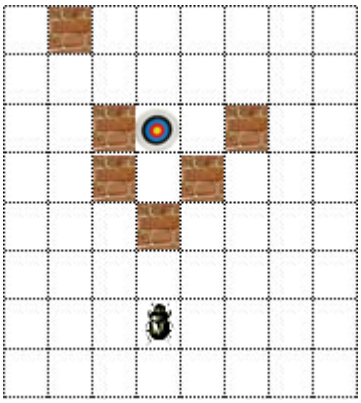

Рисунок 5.1.1. Пример поля для задачи о поиске пути («жук» – агент, «мишень» – цель, «кирпичи» – препятствия)

5.1.1. Входные данные

Входные данные, доступные агенту, следующие:

- X_t, Y_t – координаты цели;
- X_a, Y_a, D_a – координаты и направление агента;

- X_s, Y_s, D_s – координаты и направление сохранённой позиции;
- X_r, Y_r – координаты смежной клетки спереди (функции от X_a, Y_a, D_a, даны для простоты);
- O – наличие препятствия в смежной клетке спереди от агента.

Эти данные преобразованы в следующие логические переменные, которые непосредственно подаются на вход управляющему автомату агента:

- «can move forward»: $x_1 = not\ O$;
- «is move forward cool»: $x_2 = (dist(X_r, Y_r, X_t, Y_t) < dist(X_a, Y_a, X_t, Y_t))$;
- «is at finish»: $x_3 = ((X_a = X_t) \wedge (Y_a = Y_t))$;
- «is at saved»: $x_4 = ((X_a = X_s) \wedge (Y_a = Y_s) \wedge (D_a = D_s))$;
- «is better than saved»: $x_5 = (dist(X_a, Y_a, X_t, Y_t) < dist(X_s, Y_s, X_t, Y_t))$;

  где $dist(X_1, Y_1, X_2, Y_2) = |X_1 - X_2| + |Y_1 - Y_2|$.

5.1.2. Выходные воздействия

Основываясь на входных данных, агент может сделать одно из следующих действий:

- «move forward»: сделать шаг вперёд на соседнюю клетку;
- «rotate positive»: повернуть на 90 градусов по часовой стрелке;
- «rotate negative»: повернуть на 90 градусов против часовой стрелки;
- «report reached»: завершить работу и вернуть ответ о том, что цель достигнута;
- «report unreachable»: завершить работу и вернуть ответ о том, что цель недостижима;
- «save position»: сохранить текущие координаты и направление в память;
- «do nothing»: ничего не делать.

5.1.3. Возможные условия завершения

Результат работы агента может быть один из следующих:

1. Агент сделал шаг внутрь препятствия. В этом случае считается, что агент завершился аварийно.

2. Агент зацикливается.
3. Агент всё время движется в сторону от препятствия.
4. Агент выполняет действие «report reached» и не находится в целевой клетке.
5. Агент выполняет действие «report reached» и находится в целевой клетке.
6. Агент выполняет действие «report unreachable», и цель достижима.
7. Агент выполняет действие «report unreachable», цель недостижима, и агент не посетил все клетки, граничащие с диагонально-связным препятствием, внутри которого находится цель.
8. Агент выполняет действие «report unreachable», цель недостижима, и агент посетил все вышеуказанные клетки.

Только случаи 5 и 8 являются корректным завершением работы агента. Случай 7 является некорректным завершением работы, так как если агент не посетил все клетки, граничащие с диагонально-связным препятствием, внутри которого находится цель, то он не мог достоверно выяснить, что цель недостижима.

В работе [84] было сказано, что 800 автоматов были построены при помощи генетического программирования и частичной коэволюции с тестами. Все эти автоматы имеют четыре либо пять состояний. Каждый из них прошёл более 10^4 тестов, в которых были случайно сгенерированные поля размера 20×20. При этом были и достижимые, и недостижимые цели, поля были и с границей, состоящей из препятствий, и без такой границы.

Все автоматы корректно завершили работу на всех построенных тестах, однако, это не означает, что они корректны. В работе [84] формальное доказательство их корректности отсутствует.

5.2. Предлагаемый подход к верификации

Корректность работы агента следует из двух утверждений: «Для любого поля если цель достижима, то агент за конечное время достигнет её и

выполнит действие “report reached”» и «Для любого поля если цель недостижима, то агент за конечное время выполнит действие “report unreachable”». Сложность верификации состоит в том, что по условию задачи размер поля не ограничен. Автору неизвестно, как неограниченное поле закодировать при помощи темпоральных формул или автомата с конечным числом состояний.

Для решения этой задачи автором был предложен подход, состоящий из следующих шагов:

1. Построить гипотезу об алгоритме, который реализует программа.
2. Построить модель (возможно, неполную и недетерминированную) программы и внешней среды и множество формул, которые вместе с моделью будут использованы верификатором для доказательства, что данная программа соответствует гипотезе.
3. Формально доказать как теорему, что если программа соответствует гипотезе, то она корректна.
4. Запустить верификатор на всех автоматах, используя модель и формулы из шага 2. Программа, которая успешно прошла верификацию, корректна.

В настоящей главе предложенный подход будет шаг за шагом проиллюстрирован на примере верификации 800 автоматных программ, каждая из которых решает задачу, описанную в разделе 5.1.

5.3. Гипотеза

Первый шаг состоит в построении гипотезы. По результатам тестовых запусков построенных автоматных программ была сформулирована гипотеза о том, что агенты реализуют модификацию алгоритма *BUG-2* [85] с учетом того, что поле клетчатое.

Основная идея алгоритма *BUG-2* состоит в следующем. Агенты двигаются по кратчайшей линии к цели, пока это возможно. Когда агент упирается в препятствие, имеются два варианта. Если возможно повернуть и

продолжить движение по кратчайшей линии к цели (не натыкаясь на препятствия), то агент может это сделать. Кроме того, он может перейти в режим обхода препятствия. Когда можно продолжить движение к цели (оторваться от препятствия), агент это делает. Условие, при котором можно продолжить движение к цели: агент ближе к цели, чем точка соприкосновения с препятствием и можно идти в сторону цели.

5.4. Построение модели

Второй шаг состоит в построении модели и спецификации.

В предлагаемой модели для описания текущего состояния системы вводится *профиль* агента. Профиль состоит из следующих элементов:

- текущее действие;
- последнее совершённое действие;
- направление;
- соседние клетки;
- направление на цель;
- флаг, который определяет, обходит ли агент в данный момент препятствие;
- специальные флаги для обеспечения атомарности перехода из одного состояния всей системы в другое.

В листинге 5.4.1 приведена реализация профиля на языке *Promela*.

Листинг 5.4.1. Профиль агента

```
typedef Profile
{
    unsigned action : 3;//текущее действие
    unsigned lastAction : 3;//последнее действие
    unsigned direction : 2;//направление
    bit brick[9];//соседние клетки
    unsigned target : 4; //направление на цель
    bool moving;
    bool afterMoving;
```

}

Предикат `moving` означает, что все действия по пересчёту хода агента завершены. Мы считаем, что пока `moving = true` и `action ≠ NoAction`, агент совершает свой ход.

Предикат `afterMoving` означает, что ход агента закончен. При этом `action = NoAction`.

В модели хранится текущий профиль и профиль последнего сохранения агентом своего состояния.

Также в модель входят конечный автомат системы управления агентом и внешняя среда, в которой находится агент. Профиль, внешняя среда и LTL-формулы были написаны вручную. Во всех моделях эти части являются одинаковыми. Модели различаются только той частью, которая эквивалентна конечному автомату. В дальнейшем, слово «модель» будет относиться ко всем построенным моделям, если это не оговорено иначе. Эта часть была сгенерирована при помощи инструментального средства *Stater*.

5.5. LTL-спецификация

Спецификация состоит из двух наборов LTL-формул: для обходов препятствий по часовой стрелке и против часовой стрелки. Агент удовлетворяет спецификации, если он удовлетворяет хотя бы одному набору формул. Заметим, что если агент обходит препятствия, как по часовой стрелке, так и против, то он может быть корректным, но не будет удовлетворять спецификации, однако среди построенных алгоритмов таких нет. Поэтому, включать такой случай в спецификацию нет необходимости.

Отметим, что создать модель, полностью соответствующую задаче, не удалось. Этому есть несколько причин. Во-первых, поле, по которому передвигается агент, может быть бесконечным. Во-вторых, сохранение даже части поля размером более *O(1)* в памяти модели делает её слишком большого размера для верификации. Построенная модель не хранит в себе всё поле, и каждый раз, когда агент делает шаг вперёд, недетерминированно генерирует

часть поля вокруг агента. Из этого следует, что помимо всех вариантов поля, которые бывают, верификатор создаёт ещё невозможные случаи. Основными случаями, невозможными в исходной задаче, но встречающимися в модели, являются следующие:

- Изменяющееся поле – если агент сделал шаг вперёд, то информация о трёх клетках, которые были сзади агента, не будет сохранена. При возвращении агента эти клетки будут сгенерированы заново.
- Бесконечно большие препятствия.
- Бесконечно далёкая цель. Для экономии размера модели не хранятся ни координаты агента, ни координаты цели, ни даже расстояние до цели. Хранится только направление на цель и два флага: «находится ли агент в сохранённой клетке» и «ближе ли агент к цели, чем сохранённая клетка». Эти значения каждый ход пересчитываются. В случаях, когда информации не хватает для пересчёта, значения выбираются недетерминированно. Поэтому цель может быть впереди агента на протяжении бесконечного числа шагов.
- «Блуждающая цель». Так как информация о цели в некоторых случаях пересчитывается недетерминированно, то цель в построенной модели может менять своё местонахождение.
- «Блуждающая сохранённая клетка». Местоположение сохранённой клетки в модели не хранится. Поэтому оно может измениться так же, как и поле.

Таким образом, LTL-формулы должны быть не только такими, что алгоритм, который им удовлетворяет, является корректным алгоритмом поиска пути, но и такими, чтобы корректные алгоритмы удовлетворяли этим формулам и в невозможных для задачи поиска пути, но возможных для построенной модели, случаях.

Приведём набор LTL-формул для спецификации обходов против часовой стрелки.

Формулы *f0 – f3*

Группа формул, утверждающих, что в случае, как на рисунке 5.5.1, агент должен сделать шаг вперёд (сонаправленно с текущим направлением), возможно несколько раз повернувшись.

Рисунок 5.5.1. Обход № 1

В листингах 5.5.1 – 5.5.4 приведены формулы из этой группы, смысл которых описывается ниже. Все формулы группы используют макрос `o2`:

```
#define o2  ((p.brick[4] == 1) && (p.brick[2] == 0))
```

Листинг 5.5.1. Формула *f0*

```
#define dt_pos_base_up (o2 && (p.direction == DirUp) && detourWall && p.moving)

#define dt_pos_act_up (( ((p.action == Forward && p.direction == DirUp) ||
((p.action == TargetAchieved) || (p.action == TargetUnreachable))) && p.moving
) || breakaway)

#define detour_pos_up (dt_pos_base_up -> (dt_rot2 U dt_pos_act_up))

ltl f0 {[] detour_pos_up}
```

Листинг 5.5.2. Формула *f1*

```
#define dt_pos_base_right (o2 && (p.direction == DirRight) && detourWall &&
p.moving)

#define dt_pos_act_right (( ((p.action == Forward && p.direction == DirRight)
|| ((p.action == TargetAchieved) || (p.action == TargetUnreachable))) &&
p.moving ) || breakaway)

#define detour_pos_right (dt_pos_base_right -> (dt_rot2 U dt_pos_act_right))

ltl f1 {[] detour_pos_right}
```

Листинг 5.5.3. Формула *f2*

```
#define dt_pos_base_down (o2 && (p.direction == DirDown) && detourWall &&
p.moving)

#define dt_pos_act_down (( ((p.action == Forward && p.direction == DirDown) ||
((p.action == TargetAchieved) || (p.action == TargetUnreachable))) && p.moving
) || breakaway)
```

```
#define detour_pos_down (dt_pos_base_down -> (dt_rot2 U dt_pos_act_down))

ltl f2 {[] detour_pos_down}
```

Листинг 5.5.4. Формула *f3*

```
#define dt_pos_base_left (o2 && (p.direction == DirLeft) && detourWall &&
p.moving)

#define dt_pos_act_left (( ((p.action == Forward && p.direction == DirLeft) ||
((p.action == TargetAchieved) || (p.action == TargetUnreachable))) && p.moving
) || breakaway)

#define detour_pos_left (dt_pos_base_left -> (dt_rot2 U dt_pos_act_left))

ltl f3 {[] detour_pos_left}
```

Объясним смысл формулы *f0*.

`o2 == ((p.brick[4] == 1) && (p.brick[2] == 0))`: препятствие слева имеется, а спереди нет.

`dt_pos_base_up == (o2 && (p.direction == DirUp) && detourWall && p.moving)`: выполняется `o2`, у агента направление вверх, он обходит препятствие, и все действия по пересчёту шага завершены.

`dt_rot2 == ( ((p.action == RotLeft) || (p.action == RotRight) || (p.action == SavePosition) || (p.action == NoAction)) || !p.moving)`: если все действия по пересчёту шага завершены, то агент либо крутится, либо сохраняет позицию, либо ничего не делает.

`dt_pos_act_up == (( ((p.action == Forward && p.direction == DirUp) || ((p.action == TargetAchieved) || (p.action == TargetUnreachable))) && p.moving ) || breakaway)`: агент движется вперёд и направлен вверх, либо сообщил, что цель достигнута или не достигнута, либо агент оторвался от препятствия и все действия по пересчёту шага завершены.

`detour_pos_up == (dt_pos_base_up -> (dt_rot2 U dt_pos_act_up))`: если случилось `dt_pos_base_up`, то когда-нибудь выполнится `dt_pos_act_up`, а до тех пор будет верно `dt_rot2`.

Таким образом, всегда верно следующее: если слева есть препятствие, а спереди нет, агент направлен вверх и находится в состоянии обхода препятствия и все действия по пересчёту шага завершены, то когда-нибудь

либо агент подвинется наверх, либо сообщит о том, что цель достигнута или недостижима, либо оторвётся от препятствия. При этом все действия по пересчёту шага будут завершены, а до тех пор будет выполняться следующее: если все действия по пересчёту шага завершены, то агент либо крутится, либо сохраняется, либо ничего не делает.

Формулы *f1* – *f3* аналогичны *f0*, но утверждают про направления в другие стороны.

Формулы *f4* – *f7*

Данная группа формул утверждает, что в ситуации, как на рисунке 5.5.2, агент должен повернуться направо.

Рисунок 5.5.2. Обход № 2

Формулы из этой группы приведены в листингах 5.5.5 – 5.5.8. Все формулы группы используют макрос `o3`:

```
#define o3  ((p.brick[4] == 1) && (p.brick[2] == 1))
```

Смысл этих формул описан ниже.

Листинг 5.5.5. Формула *f4*

```
#define dt2_pos_base_up (o3 && (p.action != Forward) && (p.direction == DirUp)
&& detourWall && p.moving)

#define dt2_pos_act_up ( (p.direction == DirRight) || ((p.action ==
TargetAchieved) || (p.action == TargetUnreachable)) )

#define detour2_pos_up (dt2_pos_base_up -> (dt_rot2 U dt2_pos_act_up))

ltl f4 {[] detour2_pos_up}
```

Листинг 5.5.6. Формула *f5*

```
#define dt2_pos_base_right (o3 && (p.action != Forward) && (p.direction ==
DirRight) && detourWall && p.moving)

#define dt2_pos_act_right ( (p.direction == DirDown) || ((p.action ==
TargetAchieved) || (p.action == TargetUnreachable)) )
```

```
#define detour2_pos_right (dt2_pos_base_right -> (dt_rot2 U dt2_pos_act_right))

ltl f5 {[] detour2_pos_right}
```

Листинг 5.5.7. Формула *f6*

```
#define dt2_pos_base_down (o3 && (p.action != Forward) && (p.direction == DirDown) && detourWall && p.moving)

#define dt2_pos_act_down ( (p.direction == DirLeft) || ((p.action == TargetAchieved) || (p.action == TargetUnreachable)) )

#define detour2_pos_down (dt2_pos_base_down -> (dt_rot2 U dt2_pos_act_down))

ltl f6 {[] detour2_pos_down}
```

Листинг 5.5.8. Формула *f7*

```
#define dt2_pos_base_left (o3 && (p.action != Forward) && (p.direction == DirLeft) && detourWall && p.moving)

#define dt2_pos_act_left ( (p.direction == DirUp) || ((p.action == TargetAchieved) || (p.action == TargetUnreachable)) )

#define detour2_pos_left (dt2_pos_base_left -> (dt_rot2 U dt2_pos_act_left))

ltl f7 {[] detour2_pos_left}
```

Объясним значение формулы *f4*.

`o3 == ((p.brick[4] == 1) && (p.brick[2] == 1))`: слева и спереди есть препятствие.

`dt2_pos_base_up == (o3 && (p.action != Forward) && (p.direction == DirUp) && detourWall && p.moving)`: выполняется `o3`, текущее действие – любое другое действие, отличное от движения вперёд, направление вверх, агент находится в состоянии обхода и все действия по пересчёту шага завершены.

`dt2_pos_act_up == ( (p.direction == DirRight) || ((p.action == TargetAchieved) || (p.action == TargetUnreachable)) )`: агент направлен вправо или сообщил, что цель достигнута или недостижима.

`detour2_pos_up == (dt2_pos_base_up -> (dt_rot2 U dt2_pos_act_up))`: если случилось `dt2_pos_base_up`, то когда-нибудь выполнится `dt2_pos_act_up`, и до тех пор будет верно `dt_rot2`.

Таким образом, всегда верно следующее: если система (автомат и поле) пришла в такое состояние, что есть препятствие и слева, и спереди, агент

направлен вверх и находится в состоянии обхода, и все действия по пересчёту шага завершены, то когда-нибудь агент повернётся так, что будет направлен направо или сообщит о том, что цель недостижима, а до тех пор будет верно следующее: если все действия по пересчёту шага завершены, то агент либо крутится, либо сохраняется, либо ничего не делает.

Формулы *f5 – f7* аналогичны *f4*, но утверждают про направления в другие стороны.

Формулы *f8 – f11*

Данная группа формул утверждает, что если профиль агента как на рисунке 5.5.3 и цель спереди от агента и агент не находится в состоянии обхода, то агент повернёт направо.

Рисунок 5.5.3. Обход № 3

В листингах 5.5.9 –5.5.12 приведены формулы *f8 – f11*. Все формулы группы используют макросы `o0` и `no_obs`:

```
#define o0 ((p.brick[2] == 1) && IS_MOVE_FORWARD_COOL && (detourWall == false))
#define no_obs (IS_MOVE_FORWARD_COOL && CAN_MOVE_FORWARD)
```

Листинг 5.5.9. Формула *f8*

```
#define dt_begin_base_up (o0 && (p.direction == DirUp) && p.afterMoving)
#define dt_begin_act_up ((p.direction == DirRight) || \
        (p.action == TargetUnreachable) || no_obs)
#define detour_begin_up (dt_begin_base_up -> (dt_rot2 U dt_begin_act_up))
ltl f8 {[] detour_begin_up}
```

Листинг 5.1.3.10. Формула *f9*

```
#define dt_begin_base_right (o0 && (p.direction == DirRight) && p.afterMoving)
#define dt_begin_act_right ((p.direction == DirDown) || \
        (p.action == TargetUnreachable) || no_obs)
```

```
#define detour_begin_right (dt_begin_base_right -> \
        (dt_rot2 U dt_begin_act_right))
ltl f9 {[] detour_begin_right}
```

Листинг 5.5.11. Формула *f10*

```
#define dt_begin_base_down (o0 && (p.direction == DirDown) && p.afterMoving)
#define dt_begin_act_down ((p.direction == DirLeft) || (p.action == TargetUnreachable) || no_obs)
#define detour_begin_down (dt_begin_base_down -> (dt_rot2 U dt_begin_act_down))
ltl f10 {[] detour_begin_down}
```

Листинг 5.5.12. Формула *f11*

```
#define dt_begin_base_left (o0 && (p.direction == DirLeft) && p.afterMoving)
#define dt_begin_act_left ((p.direction == DirUp) || (p.action == TargetUnreachable) || no_obs)
#define detour_begin_left (dt_begin_base_left -> (dt_rot2 U dt_begin_act_left))
ltl f11 {[] detour_begin_left}
```

Объясним значение формулы *f8*:

`o0` – прямо перед агентом препятствие (нельзя идти вперёд) и движение вперёд приближает к цели и агент не находится в состоянии обхода.

`no_obs` – движение вперёд приближает к цели и можно двигаться вперёд (перед агентом нет препятствия).

`dt_begin_base_up` – верно `o0` и агент направлен вверх и все действия по пересчёту шага завершены.

`dt_begin_act_up` – агент направлен вправо или агент сообщает, что цель недостижима или `no_obs`.

`detour_begin_up` – если случилось `dt_begin_base_up`, то в будущем случится **`dt_begin_act_up`**, а до тех пор будет верно `dt_rot2` (см. описание формулы *f0*).

Таким образом, формула означает, что в любой момент если прямо перед агентом препятствие (нельзя идти вперёд), движение вперёд приближает к цели, агент не находится в состоянии обхода, агент направлен вверх и все действия по пересчёту шага завершены, то в будущем произойдёт следующее:

агент направлен вправо или агент сообщает, что цель недостижима или движение вперёд приближает к цели и можно двигаться вперёд (перед агентом нет препятствия), а пока оно не случилось, будет верно следующее: если все действия по пересчёту шага завершены, то агент либо крутится, либо сохраняется, либо ничего не делает.

Из изложенного следует, что если агент упирается в препятствие, и цель где-то спереди, то либо он начинает обход, либо он находит направление, которое тоже приближает к цели, либо сообщает о том, что цель недостижима.

Формулы *f9 – f11* аналогичны *f8*, но утверждают про направления в другие стороны.

Формулы *f12 – f15*

Данная группа формул утверждает, что если агент находится в состоянии обхода и его профиль такой, как на рисунке 5.5.4, то агент сначала сделает шаг вперёд, а затем шаг налево.

Рисунок 5.5.4. Обход № 4

В листингах 5.5.13 – 5.5.16 приведены формулы *f12 – f15*. Все формулы группы используют макрос `o4`:

```
#define o4 ((p.brick[2] == 0) && (p.brick[1] == 0) && (p.brick[4] == 1))
```

Листинг 5.5.13. Формула *f12*

```
#define dt3_base_up (o4 && (p.target != 0) && (p.direction == DirUp) &&
p.afterMoving && detourWall && (!(IS_MOVE_FORWARD_COOL && ((InSavedCell &&
!leftSaved) || IS_BETTER_THAN_SAVED))))

#define dt3_act1_up (((p.direction == DirUp) && (p.action == Forward)) ||
breakaway)

#define dt3_act2_up (((p.direction == DirLeft) && (p.action == Forward)) ||
breakaway)
```

```
#define detour3_up (dt3_base_up -> ((dt_rot2 U dt3_act1_up) && (dt3_act1_up -> (dt_rot2 U dt3_act2_up))))

ltl f12 {[] detour3_up}
```

Листинг 5.5.14. Формула *f13*

```
#define dt3_base_right (o4 && (p.target != 0) && (p.direction == DirRight) && p.afterMoving && detourWall && (!(IS_MOVE_FORWARD_COOL && ((InSavedCell && !leftSaved) || IS_BETTER_THAN_SAVED))))

#define dt3_act1_right (((p.direction == DirRight) && (p.action == Forward)) || breakaway)

#define dt3_act2_right (((p.direction == DirUp) && (p.action == Forward)) || breakaway)

#define detour3_right (dt3_base_right -> ((dt_rot2 U dt3_act1_right) && (dt3_act1_right -> (dt_rot2 U dt3_act2_right))))

ltl f13 {[] detour3_right}
```

Листинг 5.5.15. Формула *f14*

```
#define dt3_base_down (o4 && (p.target != 0) && (p.direction == DirDown) && p.afterMoving && detourWall && (!(IS_MOVE_FORWARD_COOL && ((InSavedCell && !leftSaved) || IS_BETTER_THAN_SAVED))))

#define dt3_act1_down (((p.direction == DirDown) && (p.action == Forward)) || breakaway)

#define dt3_act2_down (((p.direction == DirRight) && (p.action == Forward)) || breakaway)

#define detour3_down (dt3_base_down -> ((dt_rot2 U dt3_act1_down) && (dt3_act1_down -> (dt_rot2 U dt3_act2_down))))

ltl f14 {[] detour3_down}
```

Листинг 5.5.16. Формула *f15*

```
#define dt3_base_left (o4 && (p.target != 0) && (p.direction == DirLeft) && p.afterMoving && detourWall && (!(IS_MOVE_FORWARD_COOL && ((InSavedCell && !leftSaved) || IS_BETTER_THAN_SAVED))))

#define dt3_act1_left (((p.direction == DirLeft) && (p.action == Forward)) || breakaway)

#define dt3_act2_left (((p.direction == DirDown) && (p.action == Forward)) || breakaway)

#define detour3_left (dt3_base_left -> ((dt_rot2 U dt3_act1_left) && (dt3_act1_left -> (dt_rot2 U dt3_act2_left))))

ltl f15 {[] detour3_left}
```

Объясним значение формулы *f12*:

`o4` – слева есть препятствие, а спереди и слева-спереди препятствия нет.

`dt3_base_up` – верно `o4`, агент не находится в клетке с целью, и направлен вверх, ход завершён, агент находится в состоянии обхода, неверно, что движение вперёд не приближает к цели и либо агент находится ближе к цели, чем сохранённая клетка, либо агент находится в сохранённой клетке и не покидал её с момента сохранения.

`dt3_act1_up` – агент оторвался от препятствия, либо он направлен вверх и совершает ход вперёд.

`dt3_act2_up` – агент оторвался от препятствия, либо он направлен влево и совершает ход вперёд.

`detour3_up` – если условие `dt3_base_up` выполнено, то в будущем случится `dt3_act1_up`, а до тех пор будет верно `dt_rot2`, а затем выполнится `dt3_act2_up`, а между `dt3_act1_up` и `dt3_act2_up` будет верно `dt_rot2`.

Таким образом, формула означает, что в ситуации, как на рисунке 5.5.4, агент должен сделать шаг вверх, а затем шаг влево, возможно, поворачиваясь в разные стороны, либо оторваться от препятствия.

Формулы *f13 – f15* аналогичны *f12*, но утверждают про направления в другие стороны.

Формула *f16*

Формула *f16* приведена в листинге 5.5.17.

Листинг 5.5.17. Формула *f16*

```
#define notF ((IS_AT_FINISH == false) && p.moving)

#define Reach (p.action == TargetAchieved)

#define Unreach (p.action == TargetUnreachable)

#define A ((p.brick[2] == 1))

#define C_up ((p.direction == DirUp) && (p.brick[2] == 0) && ( (p.target == 1)
|| (p.target == 2) || (p.target == 3) ))
```

```
#define C_right ((p.direction == DirRight) && (p.brick[2] == 0) && ( (p.target == 3) || (p.target == 5) || (p.target == 8) ))

#define C_down ((p.direction == DirDown) && (p.brick[2] == 0) && ( (p.target == 6) || (p.target == 7) || (p.target == 8) ))

#define C_left ((p.direction == DirLeft) && (p.brick[2] == 0) && ( (p.target == 1) || (p.target == 4) || (p.target == 6) ))

#define C (C_up || C_right || C_down || C_left)

ltl f16 {[] ((notF && !Reach && !Unreach) -> (<>((A || C) && p.moving)))}
```

Объясним формулу *f16*:

notF: агент еще не достиг цели.

Reach: агент сообщил, что он добрался до цели.

Unreach: агент сообщил, что цель недостижима.

A: впереди препятствие.

C_up: агент направлен вверх, впереди препятствия нет и цель сверху, слева-сверх или справа-сверху (вперёд идти "хорошо").

C_right: аналогично, но направо.

C_down: аналогично, но вниз.

C_left: аналогично, но налево.

C: перед агентом препятствия нет и вперёд идти "хорошо" (шаг вперёд приближает к цели).

Таким образом, всегда будет верно, что если агент не достиг цели, не сообщил о достижении цели или не сообщил о недостижимости цели, то он обязательно достигнет такого состояния, когда все действия по пересчёту шага завершены и либо перед ним препятствие, либо препятствия нет, и шаг вперёд приближает к цели.

Формула *f17*

Эта формула утверждает, что агент не сделает шаг внутрь препятствия. Она приведена в листинге 5.5.18.

Листинг 5.5.18. Формула *f17*

```
#define my_rot ( ((p.action == RotLeft) || (p.action == RotRight)) )
ltl f17 {[] ( ((p.brick[2] == 1) && p.afterMoving) ->
        (my_rot V (!(p.moving && (p.action == Forward)))))}
```

Объясним её значение: `my_rot` означает, что агент делает поворот направо или налево. Буквой «*V*» в *Spin* обозначается темпоральный оператор **R** (Release).

Таким образом, если прямо впереди агента находится препятствие, то всегда будет верно, что агент не сделает шаг вперёд до тех пор, пока не повернётся.

Формула *f18*

Эта формула, приведённая в листинге 5.5.19, утверждает, что если агент достиг цели, то он когда-нибудь выполнит действие «report reached».

Листинг 5.5.19. Формула *f18*

```
ltl f18 {[] (IS_AT_FINISH -> (<>(p.action == TargetAchieved)))}
```

Формулы *f19 – f26*

Эти формулы, приведённые в листинге 5.5.20, утверждают, что агент будет идти прямо к цели всегда, когда алгоритм *BUG-2* этого требует.

Листинг 5.5.20. Формулы *f19 – f26*

```
#define ActUp ((p.direction == DirUp) && (p.action == Forward))
#define ActRight ((p.direction == DirRight) && (p.action == Forward))
#define ActDown ((p.direction == DirDown) && (p.action == Forward))
#define ActLeft ((p.direction == DirLeft) && (p.action == Forward))
#define NotGoUp ((!p.moving) || (!ActUp))
#define NotGoDown ((!p.moving) || (!ActDown))
#define NotGoLeft ((!p.moving) || (!ActLeft))
#define NotGoRight ((!p.moving) || (!ActRight))

ltl f19 {[] (((p.target == 1) && (p.brick[2]\
```

```
    != 0) && !ActDown && !Reached &&\
    p.afterMoving) -> ((NotGoDown && NotGoRight)\
    U ((p.action == TargetUnreachable) || ActUp\
    || ActLeft || detourWall)))}

ltl f20 {[] (((p.target == 2) && (p.brick[2]\
    != 0) && !ActDown && !Reached &&\
    p.afterMoving) -> (NotGoDown U ((p.action ==\
    TargetUnreachable) || ActUp || detourWall)))}

ltl f21 {[] (((p.target == 3) && (p.brick[2]!=0)\
    && !ActDown && !Reached && p.afterMoving) ->\
    ((NotGoDown && NotGoLeft) U ((p.action ==\
    TargetUnreachable) || ActUp || ActRight ||\
    detourWall)))}

ltl f22 {[] (((p.target == 4) && (p.brick[2]!=0)\
    && !ActRight && !Reached && p.afterMoving)->\
    (NotGoRight U ((p.action==TargetUnreachable)\
    || ActLeft || detourWall)) )}

ltl f23 {[] (((p.target == 5) && (p.brick[2]!=0)\
    && !ActLeft && !Reached && p.afterMoving) ->\
    (NotGoLeft U ((p.action==TargetUnreachable)\
    || ActRight || detourWall)) )}

ltl f24 {[] (((p.target == 6) && (p.brick[2]!=0)\
    && !ActUp && !ActRight && !Reached &&\
    p.afterMoving) -> ((NotGoUp && NotGoRight) U\
    ((p.action==TargetUnreachable) || ActDown ||\
    ActLeft || detourWall)))}
```

```
ltl f25 {[] (((p.target==7) && (p.brick[2]!=0)&&\
    !ActUp&&!Reached&&p.afterMoving) -> (NotGoUp\
    U ((p.action == TargetUnreachable)\
    || ActDown || detourWall)) )}

ltl f26 {[] (((p.target == 8) && (p.brick[2]\
    != 0) && !ActUp && !ActLeft && !Reached &&
    p.afterMoving) -> ((NotGoUp && NotGoLeft) U\
    ((p.action == TargetUnreachable) || ActDown\
    || ActRight || detourWall)))}
```

Объясним формулу *f19*:

`ActDown`: агент направлен вниз и его текущее действие – движение вперёд (агент движется вниз).

`NotGoDown`: или действия по пересчёту хода агента не закончились, или он не движется вниз.

`ActRight`: агент движется направо.

`NotGoRight`: или действия по пересчёту хода агента не закончились, или он не движется направо.

Таким образом, всегда верно следующее: если цель сверху-слева, цель не достигнута и агент не движется вниз, то когда-нибудь или агент сообщит о недостижимости цели или входит наверх или сходит налево или начнёт обход, а до этого будет верно, что агент не движется вниз и не движется вправо.

Остальные формулы аналогичны *f19*, но про другие направления пеленга на цель.

Формула *f27*

Эта формула означает, что агент может сохранять свою позицию только если он находится в сохранённой клетке, либо ближе к цели, чем сохранённая клетка. Она приведена в листинге 5.5.21.

Листинг 5.5.21. Формула *f27*

```
ltl f27 {[] ((p.action == SavePosition) ->
 (InSavedCell || IS_BETTER_THAN_SAVED))}
```

Формула *f28*

Эта формула означает, что всегда будет верно следующее: если агент никогда не достигнет цели, то обязательно случится один из вариантов:

- агент сообщит о том, что цель недостижима;
- агент будет бесконечно много раз сохранять позицию и бесконечно много раз выходить из сохранённой клетки (следовательно, будет идти в сторону бесконечно далёкой цели);
- всегда будет верно, что если он идёт вперёд, то это движение его приближает к цели;
- с некоторого места он будет бесконечно долго в состоянии обхода и никогда попадёт в ранее сохранённую клетку.

  Она приведена в листинге 5.5.22.

Листинг 5.5.22. Формула *f28*

```
ltl f28 {[] ((!(<>(p.target == 0))) -> <>((p.action == TargetUnreachable) ||\
    (([]<> (p.action == SavePosition))&& ([]<> !InSavedCell)) || \
    ([]((p.action == Forward) -> IS_MOVE_FORWARD_COOL)) || \
    (<>[](detourWall && !InSavedCell))))}
```

Формула *f29*

Формула *f29* означает, что если агент находится ближе к цели, чем сохранённая клетка, то он либо достигнет цели, либо сохранит позицию ближе, чем сохранённая клетка. Она приведена в листинге 5.5.23.

Листинг 5.5.23. Формула *f29*

```
ltl f29 {[] (IS_BETTER_THAN_SAVED ->
<>((p.action==TargetAchieved) || ((p.action
    == SavePosition)&&(IS_BETTER_THAN_SAVED))))}
```

Формула *f30*

Эта формула означает, что если агент выполнил действие «report reached», то он действительно достиг цели. Она приведена в листинге 5.5.23.

Листинг 5.5.23. Формула *f30*

```
ltl f30 {[] ((p.action == TargetAchieved) -> Reached)}
```

5.6. Доказательство корректности алгоритмов

Третий шаг состоит в том, чтобы доказать, что если алгоритм удовлетворяет спецификации, то он является алгоритмом поиска пути – удовлетворяет следующим утверждениям:

1. Алгоритм работает конечное время.
2. Агент корректно ходит по полю и не натыкается на препятствия.
3. Если существует путь к цели, то он будет найден.
4. Если пути не существует, то агент вернёт ответ о том, что цель недостижима.

Вначале докажем утверждение 3.

Лемма 5.5.1. Если путь к цели существует, то всегда своём пути агент либо дойдёт до цели, либо сохранится ближе, чем предыдущее сохранение.

Из формул *f19 – f26* следует, что до тех пор, пока агент не наткнулся на препятствие, он будет идти в сторону цели. Агент стартует из сохранённого состояния. Если путь в сторону цели свободен, то агент пойдёт в это сторону и будет находиться ближе, чем сохранённая клетка. Если агент упрётся в препятствие, то он в этот момент либо ближе, чем сохранённая клетка, либо в ней находится. Как только агент упёрся в препятствие, он начинает его обход либо по часовой стрелке, либо против (формулы *f12 – f15*).

Формулы *f0 – f15* утверждают, что если агент находится в состоянии обхода, то он будет обходить препятствие правильно. Так как агент обходит препятствие, то он будет в некоторый момент ближе, чем последняя сохранённая клетка. Следовательно (формула *f29*), агент сохранится ближе к цели, чем предыдущее сохранение, либо дойдёт до цели (в этом случае лемма

верна). Оторваться от препятствия агент может, только если он находится либо в «сохранённой» клетке, либо ближе, чем последнее сохранение и только если он совершает шаг в сторону цели. Следовательно, на следующем шаге, после того, как агент оторвётся от препятствия, он будет ближе, чем последнее сохранение. Поэтому он либо сохранится ближе последнего сохранения, либо дойдёт до цели и т. д.

Из леммы 5.5.1 следует, что можно построить последовательность из сохранённых клеток, каждая из которых ближе к цели, чем предыдущая. Так как расстояние до цели – это целое число, то эта последовательность конечная, и после последней сохранённой клетки агент дойдёт до цели. Следовательно, если путь существует, то агент за конечное число шагов дойдёт до цели. Из формул (*f18*, *f30*) следует, что после этого агент за конечное число шагов завершит работу и вернёт ответ, что цель достигнута.

Докажем утверждение 4.

Утверждение, что пути не существует равносильно тому, что есть препятствие, во время обхода которого не существует условий, при которых можно от него оторваться. Из формулы *f29* аналогично доказательству леммы 5.5.1 следует, что можно построить последовательность из сохранённых клеток, каждая из которых ближе предыдущей. Эта последовательность конечная.

Посмотрим на последнюю сохранённую клетку. Так как она последняя, то после неё агент никогда не подходил к препятствию ближе, чем расположена эта клетка.

Следовательно, это сохранение было сделано во время обхода препятствия, иначе, агент должен сделать шаг в сторону препятствия и стать ближе. Таким образом, у агента на всём пути после этой клетки не было возможности оторваться от этого препятствия, иначе, он бы стал ближе к цели, чем последняя сохранённая клетка.

Следовательно, весь остальной путь агента является обходом этого препятствия. Из формулы *f30* следует, что агент не вернёт результат, свидетельствующий о том, что путь найден. Из формулы *f28* следует, что агент либо вернёт результат, что цель недостижима, либо в будущем будет верно одно из следующих утверждений:

- можно будет построить бесконечную последовательность из сохранённых клеток, каждая из которых ближе предыдущей (в случае конечного расстояния до цели это невозможно);
- каждый следующий шаг агента будет в сторону цели (в случае конечного расстояния до цели это невозможно);
- весь следующий путь агент будет обходить препятствие и никогда не попадёт в сохранённую клетку (в случае конечного препятствия это невозможно).

Следовательно, агент завершит работу и вернёт результат, что цель недостижима.

Из справедливости утверждений 3 и 4 следует справедливость утверждения 1.

Докажем теперь утверждение 2. Формула *f17* утверждает, что агент никогда не войдёт внутрь препятствия, следовательно, агенты, удовлетворяющие этой формуле, удовлетворяют и утверждению 2.

5.7. Результаты верификации

Четвёртый шаг состоит в проведении верификации, которая проводилась следующим образом: каждый из автоматов, записанный текстом в некотором формате, автоматически преобразовывался в формат *Stater*. Далее при помощи *Stater* для каждого автомата была построена *Spin*-модель и выполнена верификация.

По результатам верификации оказалось, что из 800 автоматов только 231 удовлетворяет спецификации. Таким образом, для 231 программы удалось доказать корректность работы. Вопрос о корректности остальных автоматов

остается открытым. Возможно, для доказательства их корректности требуется строить другие гипотезы.

Выводы по главе 5

Предложен подход к верификации программ, внешняя среда которых непредставима в виде автоматов, который рассмотрен на примере использования в качестве внешней среды неограниченного клетчатого поля. При этом была выполнена верификация 800 решений задачи поиска пути агентом с *O(1)* памяти на неограниченном клетчатом поле. В результате верификации удалось доказать корректность 231 программы. Статья с описанием предложенного подхода принята на конференцию IBM Haifa Verification Conference 2014 [87].

Заключение

В работе получены следующие научные результаты:

- обоснован выбор математических моделей управляющих автоматов;
- разработаны алгоритмы построения *Spin*-моделей для одного автомата, системы вложенных автоматов (иерархического автомата) и системы параллельно работающих иерархических автоматов;
- разработан метод верификации автоматных программ: построение *Spin*-модели, преобразование *расширенных* LTL-формул в *Spin*-формулы, преобразование контрпримера в путь в системе автоматов;
- разработан подход для верификации программ в случаях, когда модель внешней среды нельзя представить в виде автомата, например, для клетчатого поля неограниченных размеров.

Также автором разработаны инструментальные средства для верификации автоматных программ разных типов. Результаты работы внедрены в практику разработки ПО и в учебный процесс на кафедре «Компьютерные технологии» Университета ИТМО. Акты о внедрении и использовании имеются.

Источники

1. *Кларк Э. М., Грамберг О., Пелед Д.* Верификация моделей программ: Model Checking. М.: МЦНМО, 2002. 416 с.
2. *Синицын С. В., Налютин Н. Ю.* Верификация программного обеспечения. М.: БИНОМ, 2008. 368 с.
3. *Pnueli A.* The temporal logic of programs / 18th IEEE Symposium on Foundations of Computer Science. 1977. Pp. 46–57. http://www.inf.ethz.ch/personal/kroening/classes/fv/f2007/readings/focs77.pdf
4. *Карпов Ю. Г.* Model Checking: верификация параллельных и распределенных программных систем. СПб.: БХВ-Петербург, 2010. 560 с.
5. *Вельдер С.Э., Лукин М.А., Шалыто А.А., Яминов Б.Р.* Верификация автоматных программ. СПб.: Наука, 2011. 244 с. (http://is.ifmo.ru/verification/velder_verification_posobie_nauka.pdf).
6. *Mikk E., Lakhnech Y., Siegel M. , Holzmann G. J.* Implementing Statecharts in Promela/SPIN. Proceedings of WIFT'98, 1998.
7. *Latella D., Majzik I., Massink M.* Automatic verification of a behavioral subset UML statechart diagrams using the SPIN model-checker // Formal Aspects of Computing 11, 1999. Pp. 637–664.

   http://home.mit.bme.hu/~majzik/publicat/fac99.pdf
8. *Lilius, J., Paltor I. P.* Formalising UML State Machines for Model Checking, in: R. B. France and B. Rumpe, editors / Proceedings of 2nd International Conference UML. 1999. LNCS: Volume 1723. Pp. 430–445.
9. *Eschbah R.* A verification approach for distributed abstract state machines. // PSI '02, 2001. Pp. 109-115. http://dl.acm.org/citation.cfm?id=705973
10. *Shaffer T., Knapp A., Merz S.* Model checking UML state machines and collaborations // Electronic notes in theoretical computer science 47, 2001. Pp. 1-13.
11. *Tiwari A.* Formal semantics and analysis methods for Simulink Stateflow models. Technical report. SRI International. 2002. http://www.csl.sri.com/~tiwari/~stateflow.html
12. *Pingree P. J., Mikk E., Holzmann G. J., Smith M. H., Dams D.* Validation of mission critical software design and implementation using model checking [spacecraft] / Proceedings of 5th International Workshop, DAS 2002, Princeton, NJ, USA, 2002.
13. *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

14. *Gnesi S., Mazzanti F.* On the fly model checking of communicating UML state machines. 2004. http://fmt.isti.cnr.it/WEBPAPER/onthefly-SERA04.pdf

15. *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

16. *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

17. *Кузьмин Е. В., Соколов В. А.* О верификации «автоматных программ». Актуальные проблемы математики и информатики. / Сборник статей к 20-летию факультета ИВТ ЯрГУ им. П. Г. Демидова, Ярославль: ЯрГУ, 2006. С. 27 – 32.

18. *Кузьмин Е. В., Соколов В. А.* О некоторых подходах к верификации автоматных программ. / Сборник докладов семинара Go4IT – шаг к новым технологиям Интернета. Москва, Институт системного программирования, 2007. С. 43 – 48.

19. *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

20. *Лукин М. А.* Верификация автоматный программ. Бакалаврская работа. СПбГУ ИТМО, 2007. 32 с. http://is.ifmo.ru/papers/_lukin_bachelor.pdf

21. *Яминов Б. Р.* Автоматизация верификации автоматных UniMod-моделей на основе инструментального средства Bogor. Бакалаврская работа. СПбГУ ИТМО, 2007. 46 с. http://is.ifmo.ru/papers/_jaminov_bachelor.pdf

22. *Ma G.* Model checking support for CoreASM: model checking distributed abstract state machines using Spin. 2007. 120 p. http://summit.sfu.ca/item/8056

23. *David A., Moller O., Yi W.* Formal Verification of UML Statecharts with Real-time Extensions. Formal Methods. 2006. Vol. 3. Pp. 15–22.

24. *Кубасов С.В., Соколов В.А.* Синхронная модель автоматной программы // Моделирование и анализ информационных систем. 2007. № 1, с. 11–18

25. *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

26. *Кузьмин Е. В., Соколов В. А.* Моделирование, спецификация и верификация автоматных программ // Программирование. 2008. № 1.

27. *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

28. *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

29. *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

30. *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

31. *Prashanth C.M., Shet K. C.* Efficient Algorithms for Verification of UML Statechart Models. // Journal of Software. 2009. Issue 3. Pp 175–182.

32. *Лукин М. А.* Верификация визуальных автоматных программ с использованием инструментального средства Spin. СПбГУ ИТМО, 2009. 50 с. http://is.ifmo.ru/papers/_lukin_master.pdf

33. *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

34. *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

35. *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

36. *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

37. *Шошмина И. В., Карпов Ю. Г.* Технология проектирования и верификации распределенных бортовых систем / Материалы международного семинара PSSV. Казань. 2010.

38. *Вяткин В. В., Дубинин В. Н.* Верификация приложений IEC 61499 на основе метода Model checking. // Известия высших учебных заведений. Поволжский район. Технические науки. 2011. Вып. 3 (19), с. 44 – 55. http://cyberleninka.ru/article/n/verifikatsiya-prilozheniy-iec-61499-na-osnove-metoda-model-checking

39. *Miyazawa A., Cavalcanti A.* Refinement-based verification of sequential implementations of Stateflow charts // Formal aspects of computing. 2011. Issuie 6.

40. *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.

41. D. Harel. Statecharts: A visual formalism for complex systems // Sci. Comput. Program., 1987. Issue 8. 231–274.

42. *Шалыто А. А.* Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. 628 с. http://is.ifmo.ru/books/switch/1

43. *Cardei I., Jha R., Cardei M.* Hierarchical architecture for real-time adaptive resource management. Secaucus, NJ, USA: Springer-Verlag, 2000.

44. *Кузьмин Е. В.* Иерархическая модель автоматных программ. // Моделирование и анализ информационных систем. Ярославль: ЯрГУ. Т. 13, №1 (2006) с. 26 – 34.

45. *Поликарпова Н. И., Шалыто А. А.* Автоматное программирование. СПб.: Питер, 2010. 168 с. http://is.ifmo.ru/books/_book.pdf

46. *Вельдер С.Э., Лукин М.А., Шалыто А.А., Яминов Б.Р.* Верификация автоматных программ. Учебное пособие. СПб.: НИУ ИТМО, 2011. 242 с. http://is.ifmo.ru/verification/velder_verification_posobie.pdf

47. *Lukin M., Velder S., Shalyto A., Yaminov B*. Verification of automata-based programs. Монография на английском (рукопись). СПб.: НИУ ИТМО, 2013. 104 с. http://is.ifmo.ru/verification/3_3_37_54_UP_Software_verification.pdf

48. *Хопкрофт Дж., Мотвани Р., Ульман Дж.* Введение в теорию автоматов, языков и вычислений. М.: Вильямс, 2001. 528 с.

49. *Mealy G. H.* A Method to Synthesizing Sequential Circuits // Bell Systems Technical Journal. AT&T, 1955. Vol. 5 (34). Pp. 1045–1079.

50. *Moore E*. Gedanken-experiments on sequential machines // Annals of mathematics studies, Princeton, N.J.: Princeton University Press, 1956. Vol. 34. Automata studies. Pp. 129–153. http://www.ams.org/journals/proc/1961-012-04/S0002-9939-1961-0177048-3/#sthash.PWMBmNS0.dpuf
51. *Руководство по Matlab*. Распространение событий для синхронизации состояний. http://www.mathworks.com/help/stateflow/ug/broadcasting-events-to-synchronize-states.html
52. *Biere A., Cimatti A., Clarke E., Strichman O., Zhu Y.* Bounded Model Checking // Advances in Computers. 2003. Volume 58. Pp. 117–148. http://fmv.jku.at/papers/BiereCimattiClarkeStrichmanZhu-Advances-58-2003-preprint.pdf
53. *Описание Therac-25*. http://ru.wikipedia.org/wiki/Therac-25
54. *S. Kane, E. Liberman, T. DiViesti, F. Click*. Toyota Sudden Unintended Acceleration. Safety Research & Strategies, Inc, 2010.

    http://www.safetyresearch.net/Library/ToyotaSUA020510FINAL.pdf
55. *Блумберг*. http://www.bloomberg.com/news/2012-10-17/knight-capital-reports-net-loss-as-software-error-takes-toll-1-.html
56. *Umrigar Z., Pitchumani V.* Formal verification of a real-time hardware design / Proceedings of the 20th Design Automation Conference, 1983. http://portal.acm.org/ft_gateway.cfm?id=800667&type=pdf&CFID=112534228&CFTOKEN=12780503
57. *Chlipala A*. Certified Programming with Dependent Types. Cambridge: The MIT Press, 2013. 424 p.
58. Официальная страница продукта Valgrind. http://www.valgrind.org/
59. Официальная страница компании Coverity. http://www.coverity.com/
60. Официальная страница продукта PVS-Studio. http://www.viva64.com/ru/pvs-studio/
61. *Dynamic program analysis* – Wikipedia. http://en.wikipedia.org/wiki/Dynamic_program_analysis
62. *Глухих М. И., Ицыксон В. М.* Программная инженерия. Обеспечения качества программных средств методами статического анализа : учеб. пособие // СПб.: Изд-во Политехн. ун-та, 2011. 150 с.
63. *Java Pathfinder homepage*. http://javapathfinder.sourceforge.net/
64. *CBMC homepage*. http://www.cprover.org/cbmc/
65. *Разработка* технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода. Промежуточный отчёт по I этапу «Выбор направления исследований и базовых методов». http://is.ifmo.ru/verification/_2007_01_report-verification.pdf
66. *Разработка* технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода. Промежуточный отчёт по II этапу «Теоретические исследования

поставленных перед НИР задач». http://is.ifmo.ru/verification/_2007_02_report-verification.pdf
67. *Разработка* технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода. Промежуточный отчёт по III этапу «Экспериментальные исследования поставленных перед НИР задач». http://is.ifmo.ru/verification/_2007_03_report-verification.pdf
68. *Разработка* технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода. Заключительный отчёт по IV этапу «Обобщение и оценка результатов исследований». http://is.ifmo.ru/verification/_2007_04_report-verification.pdf
69. *Разработка* технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода. Отчёт о патентных исследованиях. http://is.ifmo.ru/verification/_2007_01_patent-verification.pdf
70. *Официальная страница инструментального средства Uppaal*. http://www.uppaal.org/
71. *Бейзер Б.* Тестирование черного ящика. Технологии функционального тестирования программного обеспечения и систем. СПб.: Питер, 2004. 165 с.
72. *Promela reference* – ltl. http://www.spinroot.com/spin/Man/ltl.html
73. *Spin homepage*. http://spinroot.com
74. *Шошмина И. В., Карпов Ю. Г.* Введение в язык Promela и систему комплексной верификации Spin. Учебное пособие. СПбГПУ. 2010. 111 с.
75. *Лукин М. А., Шалыто А. А.* Разработка и автоматическая верификация распределенных автоматных программ // Информационно-управляющие системы. 2013. № 5 (66). С. 43 – 50.
76. *Лукин М. А.* Верификация параллельных автоматных программ // Научно-технический вестник СПбГУ ИТМО. 2014. № 1 (89). С. 145–162.
77. *Dijkstra E.W.* Guarded commands, non-determinacy and formal derivation of programs // CACM. 1975. Vol. 8. Pp. 453–457 http://www.cs.virginia.edu/~weimer/615/reading/DijkstraGC.pdf
78. ISO/IEC 9899:2011. Information technology – Programming languages – C, 2011. 683 p.
79. *Promela reference* – printf. http://spinroot.com/spin/Man/printf.html
80. *Promela reference* – Run-Time Options http://spinroot.com/spin/Man/Spin.html
81. Unimod home page. http://unimod.sourceforge.net/
82. *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб: Питер, 2001. 368 с.

83. *Ульянцев В*. Отчёт о верификации программы управления часами с будильником. is.ifmo.ru/verification/2013/alarm_clock_verification.pdf
84. *Buzdalov, M., Sokolov, A*. Evolving EFSMs Solving a Path-Planning Problem by Genetic Programming / Proceedings of Genetic and Evolutionary Computation Conference Companion. Pp. 591–594 (2012).
85. *Lumelsky V., Stepanov A*. Path planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape // Algorithmica, 1987. Vol. 2. Pp. 403–430.
86. Kamon I., Rivlin E., Rimon E. A new range-sensor based globally convergent navigation algorithm for mobile robots / Proceedings of IEEE International Conference on Robotics and Automation, no. 1, 1996. Pp. 429–435.
87. *Lukin M. A, Buzdalov M. V, Shalyto A. A*. Formal Verification of 800 Genetically Constructed Automata Programs: A Case Study / Proceedings of the 10th IBM Haifa Verification Conference, 2014. LNCS 8855. Pp. 165–170.

Приложение А. Модель и вывод *Spin* при верификации программы *AntSysEditor*

Модель

```
#define ALoader_Antsys 0

#define
ALoader_____________________________Kind_____________________________ 1

#define ALoader_AE 2


#define kind 1

#define Index 2

#define id 3

#define Kind 4

#define ASName 5

#define BegFreq 6

#define EndFreq 7

#define Roll 8

#define Pitch 9

#define Yaw 10

#define Latitude 11

#define Longitude 12

#define Height 13

#define ae 14

#define x 15

#define y 16

#define z 17

#define phase 18


#define ALoader_SetIndex 1

#define ALoader_SetId 2
```

```
#define ALoader_SetKind 3
#define ALoader_SetASName 4
#define ALoader_SetBegFreq 5
#define ALoader_SetEndFreq 6
#define ALoader_SetRoll 7
#define ALoader_SetPitch 8
#define ALoader_SetYaw 9
#define ALoader_SetLatitude 10
#define ALoader_SetLongitude 11
#define ALoader_SetHeight 12
#define ALoader_AddAE 13
#define ALoader_NewAE 14
#define ALoader_SetAEid 15
#define ALoader_SetAEX 16
#define ALoader_SetAEY 17
#define ALoader_SetAEZ 18
#define ALoader_SetAEPhase 19
typedef ALoaderData
{
     byte state;
     byte curEvent;
     byte ID;
     byte functionCall;
     byte nestedMachine;
     byte forkMachine;
     bool started;
     bool finished;
}
```

```
inline random(var, nBits)
{
     int index = 0;
     var = 0;
     do
          :: index < nBits ->
               index = index + 1;
               var = var * 2;
               if
                    :: var = var + 1;
                    :: var = var + 0;
               fi;
          :: else -> break;
     od;
}

#define p0 (loader.state == ALoader_AE)
#define p1 (loader.state == ALoader_AE)
#define p2 (loader.curEvent == kind)
chan loader_ch = [0] of {int}
inline ALoader(machine, evt)
{
     byte sendEvt;
     if
          ::(machine.state == ALoader_Antsys) ->
               printf("machine%d.state = ALoader.Antsys \n",
machine.ID);
```

```
                if
                     ::((evt == kind)) ->
                          machine.state =
ALoader_____________________Kind_____________________;
                          atomic
                          {
                               printf("machine%d. event_happened:
_kind_ \n", machine.ID);
                               machine.curEvent = kind;
                          }
                          //Code
                          //Code
                     :: else ->
                     skip;
                fi;
          ::(machine.state ==
ALoader_____________________Kind_____________________) -
>
                printf("machine%d.state =
ALoader._____________________Kind_____________________
\n", machine.ID);
                if
                     ::((evt == Index)) ->
                          machine.state =
ALoader_____________________Kind_____________________;
                          atomic
                          {
                               printf("machine%d. event_happened:
_Index_ \n", machine.ID);
                               machine.curEvent = Index;
```

```
                    }
                    //Code
                    //Code
                    atomic
                    {
                        printf("machine%d.SetIndex()\n",
machine.ID);
                        machine.functionCall = 1;
                    }
                ::((evt == id)) ->
                    machine.state =
ALoader_____________________Kind_____________________;
                    atomic
                    {
                        printf("machine%d. event_happened:
_id_ \n", machine.ID);
                        machine.curEvent = id;
                    }
                    //Code
                    //Code
                    atomic
                    {
                        printf("machine%d.SetId()\n",
machine.ID);
                        machine.functionCall = 2;
                    }
                ::((evt == Kind)) ->
                    machine.state =
ALoader_____________________Kind_____________________;
```

```
                        atomic
                        {
                             printf("machine%d. event_happened:
_Kind_ \n", machine.ID);
                             machine.curEvent = Kind;
                        }
                        //Code
                        //Code
                        atomic
                        {
                             printf("machine%d.SetKind()\n",
machine.ID);
                             machine.functionCall = 3;
                        }
                   ::((evt == ASName)) ->
                        machine.state =
ALoader_____________________Kind_____________________;
                        atomic
                        {
                             printf("machine%d. event_happened:
_ASName_ \n", machine.ID);
                             machine.curEvent = ASName;
                        }
                        //Code
                        //Code
                        atomic
                        {
                             printf("machine%d.SetASName()\n",
machine.ID);
```

```
                    machine.functionCall = 4;
                }
            ::((evt == BegFreq)) ->
                machine.state =
ALoader_____________________Kind_____________________;
                atomic
                {
                    printf("machine%d. event_happened:
_BegFreq_ \n", machine.ID);
                    machine.curEvent = BegFreq;
                }
                //Code
                //Code
                atomic
                {
                    printf("machine%d.SetBegFreq()\n",
machine.ID);
                    machine.functionCall = 5;
                }
            ::((evt == EndFreq)) ->
                machine.state =
ALoader_____________________Kind_____________________;
                atomic
                {
                    printf("machine%d. event_happened:
_EndFreq_ \n", machine.ID);
                    machine.curEvent = EndFreq;
                }
                //Code
```

```
                        //Code
                        atomic
                        {
                             printf("machine%d.SetEndFreq()\n",
machine.ID);
                             machine.functionCall = 6;
                        }
                    ::((evt == Roll)) ->
                        machine.state =
ALoader_____________________Kind_____________________;
                        atomic
                        {
                             printf("machine%d. event_happened:
_Roll_ \n", machine.ID);
                             machine.curEvent = Roll;
                        }
                        //Code
                        //Code
                        atomic
                        {
                             printf("machine%d.SetRoll()\n",
machine.ID);
                             machine.functionCall = 7;
                        }
                    ::((evt == Pitch)) ->
                        machine.state =
ALoader_____________________Kind_____________________;
                        atomic
                        {
```

```
                              printf("machine%d. event_happened:
_Pitch_ \n", machine.ID);
                              machine.curEvent = Pitch;
                         }
                         //Code
                         //Code
                         atomic
                         {
                              printf("machine%d.SetPitch()\n",
machine.ID);
                              machine.functionCall = 8;
                         }
                    ::((evt == Yaw)) ->
                         machine.state =
ALoader_____________________Kind_____________________;
                         atomic
                         {
                              printf("machine%d. event_happened:
_Yaw_ \n", machine.ID);
                              machine.curEvent = Yaw;
                         }
                         //Code
                         //Code
                         atomic
                         {
                              printf("machine%d.SetYaw()\n",
machine.ID);
                              machine.functionCall = 9;
                         }
```

```
                    ::((evt == Latitude)) ->
                         machine.state =
ALoader_____________________Kind_____________________;
                         atomic
                         {
                              printf("machine%d. event_happened:
_Latitude_ \n", machine.ID);
                              machine.curEvent = Latitude;
                         }
                         //Code
                         //Code
                         atomic
                         {
                              printf("machine%d.SetLatitude()\n",
machine.ID);
                              machine.functionCall = 10;
                         }
                    ::((evt == Longitude)) ->
                         machine.state =
ALoader_____________________Kind_____________________;
                         atomic
                         {
                              printf("machine%d. event_happened:
_Longitude_ \n", machine.ID);
                              machine.curEvent = Longitude;
                         }
                         //Code
                         //Code
                         atomic
```

```
                    {
                        printf("machine%d.SetLongitude()\n",
machine.ID);
                        machine.functionCall = 11;
                    }
                ::((evt == Height)) ->
                    machine.state =
ALoader_____________________Kind_____________________;
                    atomic
                    {
                        printf("machine%d. event_happened:
_Height_ \n", machine.ID);
                        machine.curEvent = Height;
                    }
                    //Code
                    //Code
                    atomic
                    {
                        printf("machine%d.SetHeight()\n",
machine.ID);
                        machine.functionCall = 12;
                    }
                ::((evt == ae)) ->
                    machine.state = ALoader_AE;
                    atomic
                    {
                        printf("machine%d. event_happened:
_ae_ \n", machine.ID);
                        machine.curEvent = ae;
```

```
                    }
                    //Code
                    //Code
                    atomic
                    {
                         printf("machine%d.NewAE()\n",
machine.ID);
                         machine.functionCall = 14;
                    }
               :: else ->
                    skip;
          fi;
     ::(machine.state == ALoader_AE) ->
          printf("machine%d.state = ALoader.AE \n",
machine.ID);
          if
               ::((evt == ae)) ->
                    machine.state =
ALoader_____________________Kind_____________________;
                    atomic
                    {
                         printf("machine%d. event_happened:
_ae_ \n", machine.ID);
                         machine.curEvent = ae;
                    }
                    //Code
                    //Code
                    atomic
                    {
```

```
                              printf("machine%d.AddAE()\n",
machine.ID);
                              machine.functionCall = 13;
                         }
                    ::((evt == id)) ->
                         machine.state = ALoader_AE;
                         atomic
                         {
                              printf("machine%d. event_happened:
_id_ \n", machine.ID);
                              machine.curEvent = id;
                         }
                         //Code
                         //Code
                         atomic
                         {
                              printf("machine%d.SetAEid()\n",
machine.ID);
                              machine.functionCall = 15;
                         }
                    ::((evt == x)) ->
                         machine.state = ALoader_AE;
                         atomic
                         {
                              printf("machine%d. event_happened:
_x_ \n", machine.ID);
                              machine.curEvent = x;
                         }
                         //Code
```

```
                    //Code
                    atomic
                    {
                        printf("machine%d.SetAEX()\n",
machine.ID);
                        machine.functionCall = 16;
                    }
                ::((evt == y)) ->
                    machine.state = ALoader_AE;
                    atomic
                    {
                        printf("machine%d. event_happened:
_y_ \n", machine.ID);
                        machine.curEvent = y;
                    }
                    //Code
                    //Code
                    atomic
                    {
                        printf("machine%d.SetAEY()\n",
machine.ID);
                        machine.functionCall = 17;
                    }
                ::((evt == z)) ->
                    machine.state = ALoader_AE;
                    atomic
                    {
                        printf("machine%d. event_happened:
_z_ \n", machine.ID);
```

```
                              machine.curEvent = z;
                         }
                         //Code
                         //Code
                         atomic
                         {
                              printf("machine%d.SetAEZ()\n",
machine.ID);
                              machine.functionCall = 18;
                         }
                    ::((evt == phase)) ->
                         machine.state = ALoader_AE;
                         atomic
                         {
                              printf("machine%d. event_happened:
_phase_ \n", machine.ID);
                              machine.curEvent = phase;
                         }
                         //Code
                         //Code
                         atomic
                         {
                              printf("machine%d.SetAEPhase()\n",
machine.ID);
                              machine.functionCall = 19;
                         }
                    :: else ->
                         skip;
               fi;
```

```
     fi;
}

inline loaderVolatileChange()
{
     int r;
     int ind = 0;
}

inline loaderParamChange()
{
     int r;
     int ind = 0;
}

ALoaderData loader;



proctype loaderProc ()
{
     byte newEvt;
     loader.started= true;
     loader_ch ? newEvt;
     loaderParamChange();
     do
          :: loader.finished == false ->
               loader_ch ? newEvt;
               ALoader(loader, newEvt);
```

```
          :: else -> skip;
     od;
}
proctype loaderEventSource ()
{
     byte newEvt;
     do
          :: (loader.state == ALoader_Antsys) ->
          if
               :: (1) -> newEvt = kind;
                    loaderVolatileChange();
                    atomic
                    {
                         loader_ch ! newEvt;
                         printf("loadersource sent _kind_\n");
                    }
          fi;
          :: (loader.state ==
ALoader_____________________________Kind_____________________________) -
>
          if
               :: (1) -> newEvt = Index;
                    loaderVolatileChange();
                    atomic
                    {
                         loader_ch ! newEvt;
                         printf("loadersource sent _Index_\n");
                    }
               :: (1) -> newEvt = id;
```

```
        loaderVolatileChange();
        atomic
        {
            loader_ch ! newEvt;
            printf("loadersource sent _id_\n");
        }
    :: (1) -> newEvt = Kind;
        loaderVolatileChange();
        atomic
        {
            loader_ch ! newEvt;
            printf("loadersource sent _Kind_\n");
        }
    :: (1) -> newEvt = ASName;
        loaderVolatileChange();
        atomic
        {
            loader_ch ! newEvt;
            printf("loadersource sent _ASName_\n");
        }
    :: (1) -> newEvt = BegFreq;
        loaderVolatileChange();
        atomic
        {
            loader_ch ! newEvt;
            printf("loadersource sent _BegFreq_\n");
        }
    :: (1) -> newEvt = EndFreq;
```

```
        loaderVolatileChange();
        atomic
        {
            loader_ch ! newEvt;
            printf("loadersource sent _EndFreq_\n");
        }
    :: (1) -> newEvt = Roll;
        loaderVolatileChange();
        atomic
        {
            loader_ch ! newEvt;
            printf("loadersource sent _Roll_\n");
        }
    :: (1) -> newEvt = Pitch;
        loaderVolatileChange();
        atomic
        {
            loader_ch ! newEvt;
            printf("loadersource sent _Pitch_\n");
        }
    :: (1) -> newEvt = Yaw;
        loaderVolatileChange();
        atomic
        {
            loader_ch ! newEvt;
            printf("loadersource sent _Yaw_\n");
        }
    :: (1) -> newEvt = Latitude;
```

```
                    loaderVolatileChange();
                    atomic
                    {
                         loader_ch ! newEvt;
                         printf("loadersource sent _Latitude_\n");
                    }
               :: (1) -> newEvt = Longitude;
                    loaderVolatileChange();
                    atomic
                    {
                         loader_ch ! newEvt;
                         printf("loadersource sent
_Longitude_\n");
                    }
               :: (1) -> newEvt = Height;
                    loaderVolatileChange();
                    atomic
                    {
                         loader_ch ! newEvt;
                         printf("loadersource sent _Height_\n");
                    }
               :: (1) -> newEvt = ae;
                    loaderVolatileChange();
                    atomic
                    {
                         loader_ch ! newEvt;
                         printf("loadersource sent _ae_\n");
                    }
```

```
fi;
:: (loader.state == ALoader_AE) ->
if
     :: (1) -> newEvt = ae;
          loaderVolatileChange();
          atomic
          {
               loader_ch ! newEvt;
               printf("loadersource sent _ae_\n");
          }
     :: (1) -> newEvt = id;
          loaderVolatileChange();
          atomic
          {
               loader_ch ! newEvt;
               printf("loadersource sent _id_\n");
          }
     :: (1) -> newEvt = x;
          loaderVolatileChange();
          atomic
          {
               loader_ch ! newEvt;
               printf("loadersource sent _x_\n");
          }
     :: (1) -> newEvt = y;
          loaderVolatileChange();
          atomic
          {
```

```
                    loader_ch ! newEvt;
                    printf("loadersource sent _y_\n");
               }
          :: (1) -> newEvt = z;
               loaderVolatileChange();
               atomic
               {
                    loader_ch ! newEvt;
                    printf("loadersource sent _z_\n");
               }
          :: (1) -> newEvt = phase;
               loaderVolatileChange();
               atomic
               {
                    loader_ch ! newEvt;
                    printf("loadersource sent _phase_\n");
               }
     fi;
     od;
}


init
{
     loader.ID = 0;
     loader.started = false;
     loader.finished = false;
     loader.state = ALoader_Antsys;
```

```
        run loaderProc();
        run loaderEventSource();
    }
```

Вывод верификатора *Spin*

```
ltl f0 {(<> p0 ) -> (!p1 U p2)}
ltl f0: (! (<> ((loader.state==2)))) || ((! ((loader.state==2))) U ((loader.curEvent==1)))
ltl f1: [] ((! (((loader.state==1)) && ((loader.curEvent==14)))) || ((! ((loader.state==1))) U ((loader.curEvent==14))))
ltl f2: [] ((! (((loader.state==2)) && ((loader.curEvent==15)))) || (((loader.functionCall==16)) U ((loader.state==2))))
  the model contains 3 never claims: f2, f1, f0
  only one claim is used in a verification run
  choose which one with ./pan -N name (defaults to -N f0)

(Spin Version 6.2.3 -- 24 October 2012)
    + Partial Order Reduction

Bit statespace search for:
    never claim             + (f0)
    assertion violations    + (if within scope of claim)
    acceptance   cycles     + (fairness disabled)
    invalid end states  - (disabled by never claim)

State-vector 224 byte, depth reached 58, errors: 0
       90 states, stored
        8 states, matched
       98 transitions (= stored+matched)
        0 atomic steps
```

```
hash factor: 1.49131e+006 (best if > 100.)

bits set per state: 3 (-k3)

Stats on memory usage (in Megabytes):
    0.021   equivalent memory usage for states (stored*(State-
vector + overhead))
   16.000   memory used for hash array (-w27)
   38.147   memory used for bit stack
  343.323   memory used for DFS stack (-m10000000)
  397.665   total actual memory usage

pan: elapsed time 0.001 seconds
(Spin Version 6.2.3 -- 24 October 2012)
    + Partial Order Reduction

Bit statespace search for:
    never claim             + (f1)
    assertion violations    + (if within scope of claim)
    acceptance   cycles     + (fairness disabled)
    invalid end states  - (disabled by never claim)

State-vector 224 byte, depth reached 5094, errors: 0
    35274 states, stored
    15475 states, matched
    50749 transitions (= stored+matched)
        0 atomic steps
```

```
hash factor: 3805 (best if > 100.)

bits set per state: 3 (-k3)

Stats on memory usage (in Megabytes):
    8.074   equivalent memory usage for states (stored*(State-
vector + overhead))
   16.000   memory used for hash array (-w27)
   38.147   memory used for bit stack
  343.323   memory used for DFS stack (-m10000000)
  398.153   total actual memory usage

pan: elapsed time 0.303 seconds
pan: rate 116415.84 states/second

(Spin Version 6.2.3 -- 24 October 2012)
    + Partial Order Reduction

Bit statespace search for:
    never claim             + (f2)
    assertion violations    + (if within scope of claim)
    acceptance   cycles     + (fairness disabled)
    invalid end states  - (disabled by never claim)

State-vector 224 byte, depth reached 5094, errors: 0
    35274 states, stored
    15475 states, matched
```

```
    50749 transitions (= stored+matched)
        0 atomic steps

hash factor: 3805 (best if > 100.)

bits set per state: 3 (-k3)

Stats on memory usage (in Megabytes):
    8.074   equivalent memory usage for states (stored*(State-
vector + overhead))
   16.000   memory used for hash array (-w27)
   38.147   memory used for bit stack
  343.323   memory used for DFS stack (-m10000000)
  398.153   total actual memory usage

pan: elapsed time 0.292 seconds
pan: rate 120801.37 states/second
```

Приложение Б. Модель и вывод *Spin* при верификации логики в игре *Tutrleball*

Модель

```
#define AWall_Start 0
#define AWall_GrowHorizontal 1
#define AWall_GrowVertical 2
#define AWall_Kill 3
#define AWall_Wall 4

#define APlatform_Start 0
#define APlatform_GrowOtherWall 1
```

```
#define APlatform_Grow2Walls 2
#define APlatform_GrowOneWall 3

#define Reset 1
#define NewHorizontal 2
#define NewVertical 3
#define SetWall 4
#define Bounce 5
#define Kill 6
#define StopOtherSide 7
#define StopOneSide 8
#define StartGrow 9
#define Update 10

#define AWall_KillAll 1
#define APlatform_DoGrowWallOther 2
#define APlatform_DoGrowWallOne 3
typedef AWallData
{
     byte state;
     byte curEvent;
     byte ID;
     byte functionCall;
     byte nestedMachine;
     byte forkMachine;
     bool started;
     bool finished;
}
```

```
typedef APlatformData
{
     byte state;
     byte curEvent;
     byte ID;
     byte functionCall;
     byte nestedMachine;
     byte forkMachine;
     bool started;
     bool finished;
}

inline random(var, nBits)
{
     int index = 0;
     var = 0;
     do
          :: index < nBits ->
               index = index + 1;
               var = var * 2;
               if
                    :: var = var + 1;
                    :: var = var + 0;
               fi;
          :: else -> break;
     od;
}
```

```
chan awall_ch = [0] of {int}
chan platform_ch = [0] of {int}
inline AWall(machine, evt)
{
     byte sendEvt;
     if
          ::(machine.state == AWall_Start) ->
               printf("machine%d.state    =    AWall.Start    \n",
machine.ID);
               if
                    ::((evt == NewHorizontal)) ->
                         machine.state = AWall_GrowHorizontal;
                         atomic
                         {
                              printf("machine%d.    event_happened:
_NewHorizontal_ \n", machine.ID);
                              machine.curEvent = NewHorizontal;
                         }
                         //Code
                         //Code
                   ::((evt == NewVertical)) ->
                         machine.state = AWall_GrowVertical;
                         atomic
                         {
                              printf("machine%d.    event_happened:
_NewVertical_ \n", machine.ID);
                              machine.curEvent = NewVertical;
                         }
                         //Code
```

```
                    //Code
                ::((evt == SetWall)) ->
                    machine.state = AWall_Wall;
                    atomic
                    {
                        printf("machine%d.   event_happened:
_SetWall_ \n", machine.ID);
                        machine.curEvent = SetWall;
                    }
                    //Code
                    //Code
                :: else ->
                    skip;
            fi;
        ::(machine.state == AWall_GrowHorizontal) ->
            printf("machine%d.state = AWall.GrowHorizontal \n",
machine.ID);
            if
                ::((evt == Bounce)) ->
                    machine.state = AWall_Kill;
                    atomic
                    {
                        printf("machine%d.   event_happened:
_Bounce_ \n", machine.ID);
                        machine.curEvent = Bounce;
                    }
                    //Code
                    //Code
                    atomic
```

```
                    {
                        printf("machine%d.KillAll()\n",
machine.ID);
                        machine.functionCall = 1;
                    }
                ::((evt == Kill)) ->
                    machine.state = AWall_Kill;
                    atomic
                    {
                        printf("machine%d.   event_happened:
_Kill_ \n", machine.ID);
                        machine.curEvent = Kill;
                    }
                    //Code
                    //Code
                    atomic
                    {
                        printf("machine%d.KillAll()\n",
machine.ID);
                        machine.functionCall = 1;
                    }
                ::((evt == SetWall)) ->
                    machine.state = AWall_Wall;
                    atomic
                    {
                        printf("machine%d.   event_happened:
_SetWall_ \n", machine.ID);
                        machine.curEvent = SetWall;
                    }
```

```
                    //Code
                    //Code
                :: else ->
                    skip;
            fi;
        ::(machine.state == AWall_GrowVertical) ->
            printf("machine%d.state  =  AWall.GrowVertical  \n",
machine.ID);
            if
                ::((evt == Bounce)) ->
                    machine.state = AWall_Kill;
                    atomic
                    {
                        printf("machine%d.   event_happened:
_Bounce_ \n", machine.ID);
                        machine.curEvent = Bounce;
                    }
                    //Code
                    //Code
                    atomic
                    {
                        printf("machine%d.KillAll()\n",
machine.ID);
                        machine.functionCall = 1;
                    }
                ::((evt == Kill)) ->
                    machine.state = AWall_Kill;
                    atomic
                    {
```

```
                              printf("machine%d.    event_happened:
_Kill_ \n", machine.ID);
                              machine.curEvent = Kill;
                         }
                         //Code
                         //Code
                         atomic
                         {
                              printf("machine%d.KillAll()\n",
machine.ID);
                              machine.functionCall = 1;
                         }
                    ::((evt == SetWall)) ->
                         machine.state = AWall_Wall;
                         atomic
                         {
                              printf("machine%d.    event_happened:
_SetWall_ \n", machine.ID);
                              machine.curEvent = SetWall;
                         }
                         //Code
                         //Code
                    :: else ->
                         skip;
               fi;
          ::(machine.state == AWall_Kill) ->
               printf("machine%d.state    =    AWall.Kill    \n",
machine.ID);
               if
```

```
                    ::((evt == Reset)) ->
                         machine.state = AWall_Start;
                         atomic
                         {
                              printf("machine%d.   event_happened:
_Reset_ \n", machine.ID);
                              machine.curEvent = Reset;
                         }
                         //Code
                         //Code
                    :: else ->
                    skip;
               fi;
          ::(machine.state == AWall_Wall) ->
               printf("machine%d.state    =    AWall.Wall    \n",
machine.ID);
               if
                    :: else ->
                    skip;
               fi;
     fi;
}

inline APlatform(machine, evt)
{
     byte sendEvt;
     if
          ::(machine.state == APlatform_Start) ->
```

```
                printf("machine%d.state   =   APlatform.Start   \n",
machine.ID);
                if
                     ::((evt == StartGrow)) ->
                          machine.state = APlatform_Grow2Walls;
                          atomic
                          {
                               printf("machine%d.    event_happened:
_StartGrow_ \n", machine.ID);
                               machine.curEvent = StartGrow;
                          }
                          //Code
                          //Code
                     :: else ->
                     skip;
                fi;
           ::(machine.state == APlatform_GrowOtherWall) ->
                printf("machine%d.state  =  APlatform.GrowOtherWall
\n", machine.ID);
                if
                     ::((evt == StopOtherSide)) ->
                          machine.state = APlatform_Start;
                          atomic
                          {
                               printf("machine%d.    event_happened:
_StopOtherSide_ \n", machine.ID);
                               machine.curEvent = StopOtherSide;
                          }
                          //Code
```

```
                    //Code
                ::((evt == Update)) ->
                    machine.state = APlatform_GrowOtherWall;
                    atomic
                    {
                        printf("machine%d.   event_happened:
_Update_ \n", machine.ID);
                        machine.curEvent = Update;
                    }
                    //Code
                    //Code
                    atomic
                    {

    printf("machine%d.DoGrowWallOther()\n", machine.ID);
                        machine.functionCall = 2;
                    }
                :: else ->
                    skip;
            fi;
        ::(machine.state == APlatform_Grow2Walls) ->
            printf("machine%d.state = APlatform.Grow2Walls \n",
machine.ID);
            if
                ::((evt == Update)) ->
                    machine.state = APlatform_Grow2Walls;
                    atomic
                    {
```

```
                              printf("machine%d.    event_happened:
_Update_ \n", machine.ID);
                              machine.curEvent = Update;
                         }
                         //Code
                         //Code
                         atomic
                         {

     printf("machine%d.DoGrowWallOne()\n", machine.ID);
                              machine.functionCall = 3;
                         }
                         atomic
                         {

     printf("machine%d.DoGrowWallOther()\n", machine.ID);
                              machine.functionCall = 2;
                         }
                    ::((evt == StopOneSide)) ->
                         machine.state = APlatform_GrowOtherWall;
                         atomic
                         {
                              printf("machine%d.    event_happened:
_StopOneSide_ \n", machine.ID);
                              machine.curEvent = StopOneSide;
                         }
                         //Code
                         //Code
                    ::((evt == StopOtherSide)) ->
```

```
                        machine.state = APlatform_GrowOneWall;
                        atomic
                        {
                             printf("machine%d.   event_happened:
_StopOtherSide_ \n", machine.ID);
                             machine.curEvent = StopOtherSide;
                        }
                        //Code
                        //Code
                   :: else ->
                        skip;
              fi;
         ::(machine.state == APlatform_GrowOneWall) ->
              printf("machine%d.state   =   APlatform.GrowOneWall
\n", machine.ID);
              if
                   ::((evt == StopOneSide)) ->
                        machine.state = APlatform_Start;
                        atomic
                        {
                             printf("machine%d.   event_happened:
_StopOneSide_ \n", machine.ID);
                             machine.curEvent = StopOneSide;
                        }
                        //Code
                        //Code
                   ::((evt == Update)) ->
                        machine.state = APlatform_GrowOneWall;
                        atomic
```

```
                        {
                            printf("machine%d.    event_happened:
_Update_ \n", machine.ID);
                            machine.curEvent = Update;
                        }
                        //Code
                        //Code
                        atomic
                        {

     printf("machine%d.DoGrowWallOne()\n", machine.ID);
                            machine.functionCall = 3;
                        }
                    :: else ->
                    skip;
                fi;
     fi;
}

inline awallVolatileChange()
{
     int r;
     int ind = 0;
}

inline awallParamChange()
{
     int r;
     int ind = 0;
```

```
}

inline platformVolatileChange()
{
     int r;
     int ind = 0;
}

inline platformParamChange()
{
     int r;
     int ind = 0;
}

AWallData awall;
APlatformData platform;



proctype awallProc ()
{
     byte newEvt;
     awall.started= true;
     awall_ch ? newEvt;
     awallParamChange();
     do
          :: awall.finished == false ->
               awall_ch ? newEvt;
               AWall(awall, newEvt);
```

```
          :: else -> skip;
     od;
}
proctype awallEventSource ()
{
     byte newEvt;
     do
          :: (awall.state == AWall_Start) ->
          if
               :: (1) -> newEvt = NewHorizontal;
                    awallVolatileChange();
                    platformVolatileChange();
                    atomic
                    {
                         awall_ch ! newEvt;
                         printf("awallsource                    sent
_NewHorizontal_\n");
                    }
               :: (1) -> newEvt = NewVertical;
                    awallVolatileChange();
                    platformVolatileChange();
                    atomic
                    {
                         awall_ch ! newEvt;
                         printf("awallsource                    sent
_NewVertical_\n");
                    }
               :: (1) -> newEvt = SetWall;
                    awallVolatileChange();
```

```
                platformVolatileChange();
                atomic
                {
                     awall_ch ! newEvt;
                     printf("awallsource sent _SetWall_\n");
                }
        fi;
        :: (awall.state == AWall_GrowHorizontal) ->
        if
             :: (1) -> newEvt = Bounce;
                awallVolatileChange();
                platformVolatileChange();
                atomic
                {
                     awall_ch ! newEvt;
                     printf("awallsource sent _Bounce_\n");
                }
             :: (1) -> newEvt = Kill;
                awallVolatileChange();
                platformVolatileChange();
                atomic
                {
                     awall_ch ! newEvt;
                     printf("awallsource sent _Kill_\n");
                }
             :: (1) -> newEvt = SetWall;
                awallVolatileChange();
                platformVolatileChange();
```

```
            atomic
            {
                awall_ch ! newEvt;
                printf("awallsource sent _SetWall_\n");
            }
        fi;
        :: (awall.state == AWall_GrowVertical) ->
        if
            :: (1) -> newEvt = Bounce;
                awallVolatileChange();
                platformVolatileChange();
                atomic
                {
                    awall_ch ! newEvt;
                    printf("awallsource sent _Bounce_\n");
                }
            :: (1) -> newEvt = Kill;
                awallVolatileChange();
                platformVolatileChange();
                atomic
                {
                    awall_ch ! newEvt;
                    printf("awallsource sent _Kill_\n");
                }
            :: (1) -> newEvt = SetWall;
                awallVolatileChange();
                platformVolatileChange();
                atomic
```

```
                {
                    awall_ch ! newEvt;
                    printf("awallsource sent _SetWall_\n");
                }
        fi;
        :: (awall.state == AWall_Kill) ->
        if
            :: (1) -> newEvt = Reset;
                awallVolatileChange();
                platformVolatileChange();
                atomic
                {
                    awall_ch ! newEvt;
                    printf("awallsource sent _Reset_\n");
                }
        fi;
        :: (awall.state == AWall_Wall) ->
        skip;
        od;
}

proctype platformProc ()
{
        byte newEvt;
        platform.started= true;
        platform_ch ? newEvt;
        platformParamChange();
        do
```

```
                :: platform.finished == false ->
                     platform_ch ? newEvt;
                     APlatform(platform, newEvt);
                :: else -> skip;
          od;
     }
     proctype platformEventSource ()
     {
          byte newEvt;
          do
               :: (platform.state == APlatform_Start) ->
               if
                    :: (1) -> newEvt = StartGrow;
                         awallVolatileChange();
                         platformVolatileChange();
                         atomic
                         {
                              platform_ch ! newEvt;
                              printf("platformsource          sent
_StartGrow_\n");
                         }
               fi;
               :: (platform.state == APlatform_GrowOtherWall) ->
               if
                    :: (1) -> newEvt = StopOtherSide;
                         awallVolatileChange();
                         platformVolatileChange();
                         atomic
```

```
                    {
                        platform_ch ! newEvt;
                        printf("platformsource          sent
_StopOtherSide_\n");
                    }
                :: (1) -> newEvt = Update;
                    awallVolatileChange();
                    platformVolatileChange();
                    atomic
                    {
                        platform_ch ! newEvt;
                        printf("platformsource          sent
_Update_\n");
                    }
            fi;
            :: (platform.state == APlatform_Grow2Walls) ->
            if
                :: (1) -> newEvt = Update;
                    awallVolatileChange();
                    platformVolatileChange();
                    atomic
                    {
                        platform_ch ! newEvt;
                        printf("platformsource          sent
_Update_\n");
                    }
                :: (1) -> newEvt = StopOneSide;
                    awallVolatileChange();
                    platformVolatileChange();
```

```
                    atomic
                    {
                         platform_ch ! newEvt;
                         printf("platformsource          sent
_StopOneSide_\n");
                    }
               :: (1) -> newEvt = StopOtherSide;
                    awallVolatileChange();
                    platformVolatileChange();
                    atomic
                    {
                         platform_ch ! newEvt;
                         printf("platformsource          sent
_StopOtherSide_\n");
                    }
          fi;
          :: (platform.state == APlatform_GrowOneWall) ->
          if
               :: (1) -> newEvt = StopOneSide;
                    awallVolatileChange();
                    platformVolatileChange();
                    atomic
                    {
                         platform_ch ! newEvt;
                         printf("platformsource          sent
_StopOneSide_\n");
                    }
               :: (1) -> newEvt = Update;
                    awallVolatileChange();
```

```
                    platformVolatileChange();
                    atomic
                    {
                         platform_ch ! newEvt;
                         printf("platformsource          sent
_Update_\n");
                    }
               fi;
               od;
          }


          init
          {
               awall.ID = 0;
               awall.started = false;
               awall.finished = false;
               awall.state = AWall_Start;
               platform.ID = 1;
               platform.started = false;
               platform.finished = false;
               platform.state = APlatform_Start;
               run awallProc();
               run awallEventSource();
               run platformProc();
               run platformEventSource();
          }
```

Верификация автомата AWall

Формулы

```
#define p0 (platform.state == APlatform_Grow2Walls)
#define p1 (platform.state == APlatform_Grow2Walls)
#define p2 (platform.state == APlatform_GrowOneWall)
#define p3 (platform.state == APlatform_GrowOtherWall)
#define p4 (platform.state == APlatform_Grow2Walls)
#define p5 (platform.state == APlatform_GrowOneWall)
#define p6 (platform.state == APlatform_GrowOtherWall)
#define p7 (awall.state == AWall_Kill)
#define p8 (awall.curEvent == Bounce)
#define p9 (awall.state == AWall_Wall)
ltl f0 {[]((p0) -> (p1 U (p2 || p3)))}
ltl f1 {[]((p4 U (p5 || p6)))}
```

Вывод *Spin*

```
ltl f0: [] ((! ((awall.state==3))) || (<> ((awall.state==0))))
ltl f1: [] ((! (! (<> ((awall.state==3))))) || (<> ((awall.state==4))))
  the model contains 2 never claims: f1, f0
  only one claim is used in a verification run
  choose which one with ./pan -N name (defaults to -N f0)



(Spin Version 6.2.3 -- 24 October 2012)
    + Partial Order Reduction

Bit statespace search for:
    never claim             + (f0)
```

```
    assertion violations    + (if within scope of claim)
    acceptance   cycles     + (fairness disabled)
    invalid end states  - (disabled by never claim)

State-vector 144 byte, depth reached 404, errors: 0
     2412 states, stored (2804 visited)
     1808 states, matched
     4612 transitions (= visited+matched)
        0 atomic steps

hash factor: 47866.5 (best if > 100.)

bits set per state: 3 (-k3)

Stats on memory usage (in Megabytes):
    0.368    equivalent memory usage for states (stored*(State-
vector + overhead))
   16.000   memory used for hash array (-w27)
   38.147   memory used for bit stack
  343.323   memory used for DFS stack (-m10000000)
  397.665   total actual memory usage



pan: elapsed time 0.019 seconds
pan: rate 147578.95 states/second
```

```
(Spin Version 6.2.3 -- 24 October 2012)
    + Partial Order Reduction

Bit statespace search for:
    never claim             + (f1)
    assertion violations    + (if within scope of claim)
    acceptance   cycles     + (fairness disabled)
    invalid end states  - (disabled by never claim)

State-vector 144 byte, depth reached 404, errors: 0
     3365 states, stored (4710 visited)
     4404 states, matched
     9114 transitions (= visited+matched)
        0 atomic steps

hash factor: 28496.3 (best if > 100.)

bits set per state: 3 (-k3)

Stats on memory usage (in Megabytes):
    0.513    equivalent memory usage for states (stored*(State-
vector + overhead))
   16.000   memory used for hash array (-w27)
   38.147   memory used for bit stack
  343.323   memory used for DFS stack (-m10000000)
  397.665   total actual memory usage
```

```
pan: elapsed time 0.035 seconds
pan: rate 134571.43 states/second
```

Верификация автомата APlatform

Формулы:

```
#define p0 (aplatform.state == APlatform_Grow2Walls)
#define p1 (aplatform.state == APlatform_Grow2Walls)
#define p2 (aplatform.state == APlatform_GrowOneWall)
#define p3 (aplatform.state == APlatform_GrowOtherWall)
#define p4 (aplatform.state == APlatform_Grow2Walls)
#define p5 (aplatform.state == APlatform_GrowOtherWall)
#define p6 (aplatform.state == APlatform_GrowOtherWall)
#define p7 (aplatform.state == APlatform_Start)
#define p8 (aplatform.state == APlatform_GrowOtherWall)
#define p9 (aplatform.curEvent == StopOneSide)
#define p10 (aplatform.curEvent == StopOtherSide)
#define p11 (aplatform.state == APlatform_Start)
#define p12 (aplatform.state == APlatform_GrowOtherWall)
#define p13 (aplatform.state == APlatform_GrowOneWall)
#define p14 (aplatform.state == APlatform_GrowOtherWall)
#define p15 (aplatform.state == APlatform_GrowOneWall)
#define p16 (aplatform.state == APlatform_Grow2Walls)
#define p17 (aplatform.state == APlatform_GrowOtherWall)
#define p18 (aplatform.state == APlatform_GrowOneWall)
#define p19 (aplatform.state == APlatform_GrowOtherWall)
#define p20 (aplatform.state == APlatform_GrowOneWall)
#define p21 (aplatform.state == APlatform_Grow2Walls)
ltl f0 {[]((p0) -> ((p1 U (p2 || p3)) || [](p4)))}
```

```
ltl f1 {[]((p5) -> ((p6 U p7) || [](p8)))}

ltl f2 {((<> p9) && (<> p10)) -> (<> p11)}

ltl f3 {(<> (p12 || p13)) -> (!(p14 && p15) U p16)}

ltl f4 {(<> (p17 || p18)) -> (!(p19 && p20) U p21)}
```

Вывод *Spin*

```
ltl      f0:      []      ((!      ((aplatform.state==2)))      ||
((((aplatform.state==2))      U      (((aplatform.state==3))      ||
((aplatform.state==1)))) || ([] ((aplatform.state==2)))))

ltl      f1:      []      ((!      ((aplatform.state==1)))      ||
((((aplatform.state==1))   U   ((aplatform.state==0)))   ||   ([]
((aplatform.state==1)))))

ltl    f2:    (!    ((<>    ((aplatform.curEvent==8)))    &&    (<>
((aplatform.curEvent==7))))) || (<> ((aplatform.state==0)))

ltl      f3:      (!      (<>      (((aplatform.state==1))      ||
((aplatform.state==3)))))   ||   ((!   (((aplatform.state==1))   &&
((aplatform.state==3)))) U ((aplatform.state==2)))

ltl      f4:      (!      (<>      (((aplatform.state==1))      ||
((aplatform.state==3)))))   ||   ((!   (((aplatform.state==1))   &&
((aplatform.state==3)))) U ((aplatform.state==2)))
  the model contains 5 never claims: f4, f3, f2, f1, f0
  only one claim is used in a verification run
  choose which one with ./pan -N name (defaults to -N f0)



(Spin Version 6.2.3 -- 24 October 2012)
    + Partial Order Reduction


Bit statespace search for:
```

```
    never claim             + (f0)
    assertion violations    + (if within scope of claim)
    acceptance   cycles     + (fairness disabled)
    invalid end states  - (disabled by never claim)

State-vector 132 byte, depth reached 665, errors: 0
     3225 states, stored
     2090 states, matched
     5315 transitions (= stored+matched)
        0 atomic steps

hash factor: 41617.9 (best if > 100.)

bits set per state: 3 (-k3)

Stats on memory usage (in Megabytes):
    0.455    equivalent memory usage for states (stored*(State-
vector + overhead))
   16.000   memory used for hash array (-w27)
   38.147   memory used for bit stack
  343.323   memory used for DFS stack (-m10000000)
  397.665   total actual memory usage

pan: elapsed time 0.02 seconds
pan: rate    161250 states/second
(Spin Version 6.2.3 -- 24 October 2012)
    + Partial Order Reduction
```

```
Bit statespace search for:
    never claim             + (f1)
    assertion violations    + (if within scope of claim)
    acceptance   cycles     + (fairness disabled)
    invalid end states  - (disabled by never claim)

State-vector 132 byte, depth reached 665, errors: 0
     2965 states, stored
     1690 states, matched
     4655 transitions (= stored+matched)
        0 atomic steps

hash factor: 45267.4 (best if > 100.)

bits set per state: 3 (-k3)

Stats on memory usage (in Megabytes):
    0.418    equivalent memory usage for states (stored*(State-
vector + overhead))
   16.000   memory used for hash array (-w27)
   38.147   memory used for bit stack
  343.323   memory used for DFS stack (-m10000000)
  397.665   total actual memory usage
pan: elapsed time 0.019 seconds
pan: rate 156052.63 states/second
(Spin Version 6.2.3 -- 24 October 2012)
    + Partial Order Reduction
Bit statespace search for:
```

```
    never claim             + (f2)
    assertion violations    + (if within scope of claim)
    acceptance   cycles     + (fairness disabled)
    invalid end states  - (disabled by never claim)

State-vector 48 byte, depth reached 0, errors: 0
        1 states, stored
        0 states, matched
        1 transitions (= stored+matched)
        0 atomic steps
hash factor: 1.34218e+008 (best if > 100.)

bits set per state: 3 (-k3)

Stats on memory usage (in Megabytes):
    0.000    equivalent memory usage for states (stored*(State-
vector + overhead))
   16.000   memory used for hash array (-w27)
   38.147   memory used for bit stack
  343.323   memory used for DFS stack (-m10000000)
  397.665   total actual memory usage



pan: elapsed time 0 seconds


(Spin Version 6.2.3 -- 24 October 2012)
```

```
    + Partial Order Reduction

Bit statespace search for:
    never claim             + (f3)
    assertion violations    + (if within scope of claim)
    acceptance   cycles     + (fairness disabled)
    invalid end states  - (disabled by never claim)

State-vector 132 byte, depth reached 56, errors: 0
       47 states, stored
        7 states, matched
       54 transitions (= stored+matched)
        0 atomic steps

hash factor: 2.8557e+006 (best if > 100.)

bits set per state: 3 (-k3)

Stats on memory usage (in Megabytes):
    0.007    equivalent memory usage for states (stored*(State-
vector + overhead))
   16.000   memory used for hash array (-w27)
   38.147   memory used for bit stack
  343.323   memory used for DFS stack (-m10000000)
  397.665   total actual memory usage
```

```
pan: elapsed time 0 seconds


(Spin Version 6.2.3 -- 24 October 2012)
    + Partial Order Reduction

Bit statespace search for:
    never claim             + (f4)
    assertion violations    + (if within scope of claim)
    acceptance   cycles     + (fairness disabled)
    invalid end states  - (disabled by never claim)

State-vector 132 byte, depth reached 56, errors: 0
       47 states, stored
        7 states, matched
       54 transitions (= stored+matched)
        0 atomic steps

hash factor: 2.8557e+006 (best if > 100.)

bits set per state: 3 (-k3)

Stats on memory usage (in Megabytes):
    0.007    equivalent memory usage for states (stored*(State-
vector + overhead))
   16.000   memory used for hash array (-w27)
   38.147   memory used for bit stack
  343.323   memory used for DFS stack (-m10000000)
```

```
  397.665   total actual memory usage

pan: elapsed time 0 seconds
```