Верификация 800 автоматных программ, построенных при помощи генетического программирования

Лукин М.А., Буздалов М.В., Шалыто А.А.
Университет ИТМО
197101, Санкт-Петербург,
Кронверкский пр., д. 49
Адреса электронной почты: {lukinma, mbuzdalov}@gmail.com, shalyto@mail.ifmo.ru

Аннотация—При создании критически важного программного обеспечения требуется верифицировать его соответствие ряду свойств. Это часто делается при помощи проверки моделей. Однако, создание верификационной модели для программы и анализ контрпримера – это непростая задача. Она может быть упрощена при помощи парадигмы автоматного программирования.

Существуют случаи, когда надо верифицировать множество программ, реализующих одинаковую функциональность и когда невозможно построить точную модель внешней по отношению к программе среды, достаточную для проведения верификации, в виде автомата. В статье представлен подход к верификации автоматных программ в таких случаях. В статье представлен пример, который основывается на 800 автоматных программах, которые решают простую задачу поиска пути. Результат применения подхода: для 231 программы доказано, что они работают правильно.

I. Введение

При создании критически важного программного обеспечения (ПО) требуется верифицировать его соответствие ряду свойств. Это часто делается при помощи проверки моделей [1]. Однако, создание верификационной модели для программы и анализ контрпримера — это непростая задача.

Парадигма автоматного программирования [2], [3] требует, чтобы логика программы была отделена от вычислений и описана в виде системы взаимодействующих автоматов. Программы, разработанные в рамках данной парадигмы, являются изоморфными своим моделям. Благодаря этому для автоматных программ сильно упрощается построение модели для верификации методом проверки моделей [4].

В некоторых случаях возможно применить методы поисковой инженерии программного обеспечения (англ. search-based software engineering), например, генетические алгоритмы, для построения автоматных программ [5]–[7]. Это может привести к необходимости проверить большое число программ, в основе которых может лежать очень сложная и запутанная логика. Кроме того, иногда невозможно построить такую конечноавтоматную модель внешней среды, которая позволила бы проверить требуемые свойства.

В данной статье представлен подход для верификации автоматных программ, при помощи которого можно осуществить верификацию в таких условиях. Мы проиллюстрируем этот подход на примере 800 автоматных программ для решения задачи поиска пути, построенных при помощи генетического алгоритма. Было доказано, что 231 из них работает правильно. Однако, как будет показано в работе, это не значит, что остальные программы некорректны.

II. Обзор работ по теме исследования

A. Парадигма автоматного программирования

Автоматное программирование [2], [3], [8], [9] — это парадигма программирования, которая предполагает проектировать и реализовывать программное обеспечение как систему взаимодействующих автоматизированных объектов управления [2]. Каждый из таких объектов состоит из управляющего конечного автомата (УКА), либо системы УКА, и объекта управления.

Главная идея автоматного программирования заключается в том, чтобы разделить управляющие состояния и вычислительные состояния. Число управляющих состояний достаточно мало. Каждое состояние качественно отличается от других и определяет действия. Число вычислительных состояний может быть очень велико (и даже бесконечно), они отличаются друг от друга количественно и определяют только результаты действий, но не сами действия.

Сложные программы могут быть спроектированы при помощи декомпозиции автоматов [2]. Так, например, можно спроектировать различные УКА, определяющие различные стратегии поведения системы, а УКА верхнего уровня будет решать, какую стратегию использовать.

Одним из главных преимуществ автоматного программирования является то, что автоматные программы могут быть эффективно верифицированы при помощи метода проверки моделей. Верификация ПО особенно важна в критически важном ПО [1]. Однако, ПО общего вида требует ручного построения модели, верификации данной модели, а затем преобразования

контрпримера обратно в термины исходного ПО. В автоматном программировании программы уже изоморфны своим моделям, что позволяет автоматизировать их верификацию [4], [7], [10].

B. Автоматное программирование в промышленности

Новый стандарт для распределенного контроля и автоматизации IEC 61499 облегчает использование автоматных моделей и автоматных программ для проектирования и разработки распределенных систем с существенно децентрализованной логикой [11], [12]. Возможность автоматической верификации при помощи проверки моделей крайне необходима [13].

Автоматные программы облегчают разработку корректных web-приложений [14] и некоторых видов встроенного ПО, такого как JavaCard [15].

C. Генерация автоматных программ по спецификации

Задача построения различных типов автоматов, таких как детерминированные конечные автоматы (ДКА) и детерминированные конечные преобразователи (finite-state tranducers, FST) привлекла значительное число исследований. В работе [16] для построения ДКА из множества двоичных строк был использован эволюционный алгоритм. Авторы предложили алгоритм для автоматического назначения допускающих состояний ДКА оптимальным способом по заданным входным строкам, что существенно уменьшает пространство поиска и ускоряет построение автоматов. Те же авторы использовали рандомизированный локальный поиск (randomized local search) для обучения детерминированных конечных преобразователей по тестовым примерам [17]. Они определили, что наилучшим способом определить меру схожести строк для ее использования в функции приспособленности является расстояние Левенштейна [18]. В работе [19] управляющие алгоритмы, основанные на детерминированных конечных преобразователях, были «выращены» для задачи состязания за ресурсы.

По тематике управляющих конечных автоматов известно существенно меньше публикаций. В работе [5] УКА выращивались с помощью генетических алгоритмов для решения задачи поиска пути. Авторы работы [6] разработали генетический алгоритм для построения контроллеров, основанных на УКА, с дискретными и непрерывными выходными воздействиями для решения задачи управления беспилотным самолетом. В работе [20] был предложен подход для построения УКА, основанный на муравьиных алгоритмах. Наконец, в работе [7] был предложен генетический алгоритм для построения УКА по тестовым сценариям и темпоральным формулам.

III. Формулировка задачи

В работе [5] было построено решение в виде конечных автоматов для задачи поиска пути при помощи генетического алгоритма. Точнее говоря, решалась задача поиска пути с неполной информацией. Эта задача описана ниже.

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 он имеет очень ограниченный датчик, который позволяет агенту только определять, столкнулся ли он с препятствием и идти вдоль границы препятствия. В работе [21] было показано, что существует ряд алгоритмов (наиболее известных как BUG-1 и BUG-2), которые всегда работают конечное время и находят цель, если она достижима, и сообщают о том, что она недостижима, если она недостижима. В работе [22] идеи алгоритмов семейства BUG были расширены для случая сенсоров, ограниченных по расстоянию.

В работе [5] рассмотрена дискретная версия этой задачи, которая больше подходит для экспериментов с построением автоматных программ. В этой задаче поле представляет собой бесконечную клетчатую сетку. Каждая клетка 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. Агент имеет $O(1)$ дополнительной памяти. В алгоритмах BUG-1 и BUG-2 дополнительная память используется для того, чтобы сохранить состояние (координаты и направление) агента в некоторый момент времени. Логика агента закодирована в автомате согласно парадигме автоматного программирования.

A. 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_i, Y_j, где $i, j \in \{-1, 0, 1\}, i \neq j$ — координаты смежных по ребру или вершине клеток (функции от X_a, Y_a, D_a, даны для простоты);
- O — есть ли препятствие в смежной клетке спереди от агента.

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

- “can move forward”: x1 = not O;
- “is move forward cool”: x2 = $(dist(X_j, Y_j, X_t, Y_t) < dist(X_a, Y_a, X_t, Y_t))$;

- “is at finish”: x3 = $(X_a = X_t$ and $Y_a = Y_t)$;
- “is at saved”: x4 = $(X_a = X_s$ and $Y_a = Y_s$ and $D_a = D_s)$;
- “is better than saved”: x5 = $(dist(X_a, Y_a, X_t, Y_t) < dist(X_s, Y_s, X_t, Y_t))$;

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

B. 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.

C. 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”, и цель недостижима, и агент не посетил все клетки, граничащие с диагонально-связным препятствием, внутри которого находится цель.
8) Агент выполняет действие “report unreachable”, и 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 [5] было сказано, что 800 ИКА 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, 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 [5] формальное доказательство их корректности отсутствует.

IV. Предлагаемый подход к верификации

Возможно ли написать темпоральные формулы, которые эквивалентны следующим утверждениям: «если цель достижима, то агент когда-нибудь её достигнет и выполнит действие „report reached“» и «если цель недостижима, то агент посетит все достижимые клетки, которые граничат с диагонально-связной компонентой связности препятствий, внутри которой находится цель, и после этого когда-нибудь выполнит действие „report unreachable“»?

Первая сложность состоит в том, что даже простые свойства поля, такие как существование пути между текущим положением агента и целью очень сложно, если вообще возможно записать при помощи LTL-формул. Другая проблема состоит в том, что много разных алгоритмов, таких как эквиваленты BUG-1 и BUG-2, должны удовлетворять этим формулам. Однако идеи доказательств для разных алгоритмов требуют разных инвариантов, и все эти инварианты (и, вероятно, многие другие, которые требуются для ещё не обнаруженных среди построенных автоматов алгоритмов) должны быть в LTL-спецификации. Наконец, если верификационная система должна строить контрпример в виде клетчатого поля, потребуется строить всё более и более крупные поля при верификации корректных агентов, пока не закончится память.

В настоящей работе предлагается другой подход, который является менее общим и не позволяет верифицировать сразу все автоматы одновременно, но имеет приемлемую сложность и достаточен для доказательства корректности многих автоматов, если они реализуют одинаковый алгоритм. Подход состоит из следующих этапов:

1) Построить гипотезу об алгоритме, который реализует серия автоматов.
2) Построить модель (возможно, неполную и недетерминированную) агента и внешней среды и множество формул, которые вместе с моделью будут использованы верификатором для доказательства, что данный автомат соответствует гипотезе.
3) Формально доказать, что если автомат соответствует гипотезе, то он корректен.
4) Запустить верификатор на всех автоматах, используя модель и формулы из шага 2. Все автоматы, которые успешно прошли верификацию, корректны.

В разделе V этот подход будет реализован для верификации 800 автоматов, решающих задачу поиска пути, описанную в разделе III.

V. Применение подхода для задачи поиска пути

A. Гипотеза

Далее под препятствием будет пониматься диагонально-связная компонента связности одноклеточных препятствий.

После предварительных экспериментов с несколькими агентами на разных полях была выдвинута гипотеза, что, в основном, они реализуют алгоритм BUG-2 [21] с небольшим, но заметным отклонением от него: при использовании манхэттенского расстояния кратчайший путь между двумя клетками не один, а их множество, и множество клеток, составляющих эти пути, представляет собой прямоугольник. Агент может двигаться по любому из этих путей.

Другими словами, такие агенты двигаются по кратчайшему пути к цели, пока это возможно. Когда агент упирается в препятствие, есть две возможности. Во-первых, если возможно повернуть и продолжить двигаться по кратчайшей линии к цели (не натыкаясь на препятствия), то агент может это сделать. Во-вторых, он может перейти в режим обхода препятствия: он обходит вдоль границы препятствия по часовой или против часовой стрелки (в зависимости от реализации агента), пока он не достигнет состояния, когда можно оторваться от препятствия, то есть продолжить движение по кратчайшей линии к цели или начать обходить другое препятствие. Во время обхода агент ищет ближайшую клетку к цели вокруг границы препятствия.

B. Модель

Модель предполагает, что вся работа агента разделена на ходы. За один ход управляющий автомат агента делает один переход, агент совершает одно из возможных действий, описанных в III-B, и происходит пересчёт состояния системы.

Первая часть модели — это конечное описание части поля, которая напрямую влияет на следующий ход агента. Она состоит из следующих компонентов:

- Информация о наличии препятствий в восьми соседних клетках;
- Направление агента (вверх, вниз, вправо или влево).
- Пеленг на цель (четыре направления: вверх, вниз, вправо, влево, четыре диагональных направления: вверх-вправо, вниз-вправо, вниз-влево, вверх-влево и одно значение, означающее, что агент находится в клетке с целью, — всего девять вариантов).
- относительное положение текущего состояния агента по отношению к сохранённому состоянию. За это отвечают следующие переменные:
  - IS_AT_SAVED — состояние агента идентично сохранённому (по совместительству входная переменная автомата);
  - IS_BETTER_THAN_SAVED — агент ближе к цели, чем был в сохранённой клетке (по совместительству входная переменная автомата);
  - InSavedCell — агент находится в сохранённой клетке;
  - leftSaved — агент покидал сохранённую клетку после последнего сохранения;
  - SameProfileAsSaved — профиль агента (см. ниже) идентичен сохранённому;
  - SameRot — профиль агента идентичен сохранённому с точностью до поворота.

Первые три компонента сгруппированы в структуру, которая называется профиль. Также в профиль добавлена информация о текущем и последнем действиях агента и информация о том, делает ли агент в данный момент свой ход и закончен ли пересчёт поля после хода агента, которая нужна для обеспечения атомарности модели относительно LTL-формул. Код профиля находится в Приложении. Четвёртая компонента хранится в глобальных переменных.

Другая часть модели описывает пересчёт её состояния. Действие «do nothing» не изменяет эти переменные. Действия «report reached» и «report unreachable» меняют только переменные, несущие в себе информацию о действии агента и для обеспечения атомарности. Действие «save position» влечёт за собой выставление в значение «истина» переменных IS_AT_SAVED, InSavedCell, SameProfileAsSaved, SameRot и в значение «ложь» переменных IS_BETTER_THAN_SAVED и leftSaved. Повороты меняют направление агента и ведут к повороту массива переменных, несущих информацию о препятствиях вокруг агента (так как в модели они заданы относительно направления агента).

Единственное «проблемное» действие — это «move forward», так как в модели хранится информация только о соседних клетках. Поэтому, информация о новых соседних клетках неизвестна и её приходится недетерминированно генерировать, а информация о соседних клетках, которые были позади агента, забывается. Кроме того, в некоторых случаях информация о положении цели относительно агента изменяется недетерминированно. Например, если цель была впереди агента, то после шага вперёд либо она осталась впереди, либо агент попал в клетку с целью. Если цель была впереди-слева, то она после шага вперёд может либо остаться впереди-слева, либо стать строго слева.

Для того чтобы уменьшить число ложных случаев, при пересчёте каждого действия «save position» копия профиля агента сохраняется в отдельной переменной. Эта переменная используется для пересчёта действия «move forward»: текущее и сохранённое состояния агента могут совпадать только если их профили совпадают; агент может находиться в той же клетке, что и

сохранённая позиция, только если профили совпадают с точностью до поворота.

Глобальный флаг detourWall используется для того, чтобы отслеживать, находится ли агент в состоянии обхода препятствия. На агента значение этой переменной не влияет, оно используется только для спецификации. В процессе обработки каждого действия этот флаг устанавливается или очищается с использованием определённых эвристик.

Переменные CAN_MOVE_FORWARD, IS_MOVE_FORWARD_COOL, IS_AT_FINISH, IS_AT_SAVED и IS_BETTER_THAN_SAVED пересчитываются каждый ход агента. Они являются входными данными для управляющего автомата агента и используются в LTL-формулах.

Модель реализована на языке Promela верификатора Spin [23]. Общая для всех построенных программ часть модели была написана вручную. Часть модели, которая зависит от конкретной программы, была сгенерирована при помощи инструмента Stater.[1]

C. Недостатки модели

Так как модель программ является конечным автоматом и частично недетерминирована, то в ней возможны случаи, которых не бывает в реальной задаче. Перечислим некоторые из них:

- 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. Такие препятствия запрещены условием задачи (алгоритмы семейства BUG не работают при бесконечно больших препятствиях). В этом случае агент может никогда не завершить свою работу.
- Бесконечно далёкая цель. Невозможно хранить в памяти модели расстояние до цели, так как оно может быть сколь угодно большим. Поэтому в построенной модели направление на цель в некоторых случаях меняется недетерминированно.
- «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.
- Цель находится в клетке с препятствием. Технически этот случай совпадает со случаем недостижимой цели и обычно обрабатывается агентом корректно, однако он запрещён условием задачи.

Все эти невозможные случаи никогда не появлялись в ходе построения автоматов и никогда не появятся в реальном мире. Это значит, что корректная программа может не удовлетворять спецификации, если спецификация не учитывает эти случаи.

D. LTL-Формулы

Спецификация состоит из двух наборов LTL-формул: для обходов препятствий по часовой стрелки и против часовой стрелки. Агент удовлетворяет спецификации, если он удовлетворяет ровно одному набору формул. Так как наборы формул противоречат друг другу, то агент не может удовлетворять сразу двум наборам формул. Заметим, что если агент обходит препятствия как по часовой стрелки, так и против, то он может быть корректным, но не будет удовлетворять спецификации, однако похоже, что среди построенных алгоритмов таких нет. Поэтому случай не был включён в спецификацию.

В Приложении представлен набор формул для обхода по часовой стрелке. Набор формул для обхода против часовой стрелки может быть построен из этого набора при помощи их «отражения».

Формулы f0–f3 означают, что если агент обходит препятствие и справа от агента находится препятствие, то агент сделает шаг вперёд в том же направлении или сообщит, что цель недостижима, или оторвётся от препятствия. До одного из этих действий агент может только поворачиваться, сохранять состояние, либо ничего не делать.

Формулы f4–f7 означают, что если агент обходит препятствие и справа и спереди от агента находится препятствие, то агент сделает следующее: повернётся налево, либо сообщит, что цель недостижима. До одного из этих действий агент может только поворачиваться, сохранять состояние, либо ничего не делать.

Формулы f8–f11 означают, что если спереди от агента находится препятствие, то агент сделает следующее: либо повернётся налево (в этом случае агент начнёт обходить препятствие), либо сообщит, что цель недостижима, либо повернёт в направлении цели. До одного из этих действий агент может только поворачиваться, сохранять состояние, либо ничего не делать.

Формулы f12–f15 означают, что если агент обходит препятствие и справа от агента находится препятствие, а спереди и справа-спереди препятствий нет, то агент должен сделать шаг вперёд, а затем шаг вправо, возможно, поворачиваясь в разные стороны или сохраняясь, либо оторваться от препятствия.

Формула f16 означает, что если агент не достиг цели, не сообщил о достижении цели или не сообщил о недостижимости цели, то он обязательно достигнет такого состояния, когда либо перед ним препятствие, либо препятствия нет, и шаг вперёд приближает к цели.

Формула f17 означает, что агент никогда не войдёт внутрь препятствия.

Формула f18 означает, что если агент достигнет цели, то он сообщит, что цель достигнута.

[1]Stater доступен по адресу: https://yadi.sk/d/clWWtMrIYhQZJ.

Формулы f19–f26 означают, что агент всегда идёт в сторону цели, когда это возможно.

Формула f27 означает, что агент может сохраниться, только если он находится в сохранённой клетке, либо ближе к цели, чем сохранённая клетка.

Формула f28 означает, что если агент никогда не достигнет цели, то он когда-нибудь сообщит, что цель недостижима либо будет одна из следующих ситуаций:

- агент будет сохранять состояние бесконечно много раз;
- агент будет бесконечно долго идти в сторону цели;
- агент будет бесконечно долго обходить препятствие.

Формула f29 означает, что если агент находится ближе к цели, чем последняя сохранённая клетка, то агент когда-нибудь сохранится ближе к цели, чем последнее сохранение.

Формула f30 означает, что если агент сообщил, что цель достигнута, то цель была достигнута.

E. Доказательство корректности автоматов

В этом разделе будет доказано, что каждый автомат, который удовлетворяет спецификации (набору LTL-формул f0–f30), решает задачу. Это утверждение следует из теорем 1–4, которые будут доказаны ниже.

Теорема 1: Каждый агент, который удовлетворяет LTL-формулам f0–f30, никогда не заходит внутрь препятствия.

Proof: Справедливость теоремы следует из формулы f17. ■

Лемма 1: Если существует путь к цели, то в любой момент времени верно, что агент когда-нибудь достигнет цели или когда-нибудь сохранит состояние ближе к цели, чем предыдущее сохранение.

Proof: 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 положения. Если путь в сторону цели свободен, то он пойдёт в сторону цели и будет находиться ближе, чем сохранённая клетка. Если агент наткнётся на препятствие, то он в этот момент либо ближе, чем сохранённая клетка, либо в ней находится. Как только агент наткнулся на 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).

Из формул f0–f3, f12–f15 и функции CalcEndDetour следует, что агент может оторваться от препятствия, если и только если он находится либо в сохранённой клетке, либо ближе, чем последнее сохранение и он направлен в сторону цели. Следовательно, на следующем шаге, после того, как агент оторвётся от препятствия, он будет ближе, чем последнее сохранение.

Следовательно, он либо сохранится ближе последнего сохранения, либо дойдёт до цели. И так далее. ■

Теорема 2: Если существует путь к цели, то агент, который удовлетворяет формулам f0–f30, когда-нибудь достигнет цели.

Proof: Из леммы 1 следует, что мы можем построить последовательность из сохранённых клеток, каждая из которых будет ближе к цели, чем предыдущая. Так как расстояние до цели — это целое число, то эта последовательность конечна. Следовательно, после последнего сохранения агент достигнет цели. Из формул f18, f30 следует, что после этого через конечное число шагов агент завершит работу и сообщит, что цель достигнута. ■

Теорема 3: Если пути к цели не существует, то агент, который удовлетворяет формулам f0–f30, когда-нибудь сообщит, что цель недостижима.

Proof: Отсутствие пути равносильно тому, что есть препятствие, во время обхода которого не существует условий, при которых можно от него оторваться. Из формулы f29 аналогично лемме 1 следует, что можно построить последовательность из сохранённых клеток, каждая из которых ближе к цели, чем предыдущая. Эта последовательность конечна.

Рассмотрим последнюю сохранённую клетку. Обозначим её как c_e. Так как она последняя, то после неё агент никогда не подходил к препятствию ближе, чем c_e. Иначе, по формуле f29 агент должен будет сохраниться ближе, чем c_e, а она ближайшая к цели. Это сохранение было сделано во время обхода препятствия, иначе агент должен сделать шаг в сторону препятствия и стать ближе. Следовательно, у агента на всём пути после c_e не было возможности оторваться от этого препятствия, иначе он бы стал ближе к цели, чем c_e. Следовательно, весь остальной путь агента является обходом этого препятствия. По формуле f30, агент не вернёт результат, что путь найден. Из формулы f28 следует, что агент либо вернёт результат, что цель недостижима, либо в будущем будет верно одно из следующих утверждений:

1) можно будет построить бесконечную последовательность из сохранённых клеток, каждая из которых ближе предыдущей (в случае конечного расстояния до цели это невозможно);
2) каждый следующий шаг агента будет в сторону цели (в случае конечного расстояния до цели это невозможно);
3) 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. ■

Теорема 4: Агент, который удовлетворяет LTL-формулам f0–f30, заканчивает работу за конечное время.

Proof: Так как цель может быть либо достижима (случай теоремы 2), либо недостижима (случай теоремы 3), и в обоих случаях агент завершает работу за конечное время, утверждение этой теоремы верно. ■

F. Результаты верификации

Было построено 1600 моделей для верификации, а именно по две модели для каждого автомата из работы [5]: одна модель использует формулы «по часовой стрелке», а вторая — «против часовой стрелки». Каждая модель на языке Promela была транслирована при помощи Spin в программу-верификатор на языке C, которая компилировалась при помощи GCC. После этого получившийся исполнимый файл верификатора запускался для каждой из 31 LTL-формулы, пока либо не проходились все формулы, либо верификация не выдавала ошибку. Весь процесс, запущенный в 16 потоков (каждый верификатор был однопоточным и запускался в своём потоке), занял примерно двое суток на 32-ядерном сервере с процессорами AMD Opteron™ 6272. Все 32 ядра процессора не получилось загрузить, так как оперативной памяти сервера хватило только на 16 потоков. Итого, примерно, 32 дня процессорного времени.

В результате верификации 231 автомат удовлетворил одной из двух построенных спецификаций (для обхода по часовой и против часовой стрелки) и 569 автоматов не удовлетворили ни одной. Как и ожидалось, не было ни одного автомата, который бы удовлетворил сразу обеим спецификациям.

Про автоматы, которые не удовлетворили ни одной из спецификаций, нельзя утверждать, что они некорректные. Они могут реализовывать другой алгоритм, для которого требуется другой набор LTL-формул. Действительно, некоторые из автоматов реализуют алгоритм, аналогичный BUG-1, а не BUG-2.

Для повторения эксперимента архив со всем моделями на Promela и скриптами можно загрузить по следующему адресу[2].

VI. Заключение

Был представлен подход, который может быть использован для верификации программ, в условиях отсутствия точной верификационной модели внешней среды, необходимой для проверки требуемых свойств. Этот подход включает в себя построение гипотезы о том, как верифицируемая программа работает, построение таких модели и спецификации, которые утверждают эту гипотезу и доказательства, что если программа удовлетворяет гипотезе, то она работает правильно.

[2]https://yadi.sk/d/-orvfVKnYhRFc

Этот подход был проиллюстрирован на примере задачи поиска пути, в которой построение контрпримера влечёт за собой построение неограниченных по размеру структур. Из работы [5] были получены 800 программ в виде автоматов, которые предположительно решают эту задачу. Построив гипотезу об их работе, модель и спецификацию, получилось доказать, что 231 из этих программ работает правильно. Однако, вопрос о корректности остальных программ остаётся открытым. Возможно, для доказательства требуется построить другие гипотезы.

Список литературы

[1] P. J. Pingree, E. Mikk, G. J. Holzmann, M. H. Smith, and D. Dams, "Validation of mission critical software design and implementation using model checking," p. 12, 2002. [Online]. Available: http://spinroot.com/gerard/pdf/02-1911.pdf

[2] N. Polikarpova and A. Shalyto, Automata-based Programming, 2nd Edition (in Russian). Piter, 2011.

[3] A. Shalyto, "Logic control and reactive systems: Algorithmization and programming," Automation and Remote Control, vol. 62, no. 1, pp. 1–29, 2001.

[4] E. Kurbatsky, "Verification of automata-based programs," in Proceedings of the Second Spring Young Researchers Colloquium on Software Engineering, vol. 2, 2008, pp. 15–17.

[5] M. Buzdalov and A. Sokolov, "Evolving EFSMs Solving a Path-Planning Problem by Genetic Programming," in Proceedings of Genetic and Evolutionary Computation Conference Companion, 2012, pp. 591–594.

[6] A. Alexandrov, A. Sergushichev, S. Kazakov, and F. Tsarev, "Genetic algorithm for induction of finite automata with continuous and discrete output actions," in Proceedings of the 13th annual conference companion on Genetic and evolutionary computation, ser. GECCO '11. New York, NY, USA: ACM, 2011, pp. 775–778. [Online]. Available: http://doi.acm.org/10.1145/2001858.2002089

[7] F. Tsarev and K. Egorov, "Finite State Machine Induction Using Genetic Algorithm Based on Testing and Model Checking," in Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation, ser. GECCO '11. New York, NY, USA: ACM, 2011, pp. 759–762. [Online]. Available: http://doi.acm.org/10.1145/2001858.2002085

[8] V. Gurov, M. Mazin, A. Narvsky, and A. Shalyto, "Tools for support of automata-based programming," Programming and Computer Software, vol. 33, no. 6, pp. 343–355, 2007.

[9] A. Shalyto and N. Tukkel', "SWITCH Technology: An Automated Approach to Developing Software for Reactive Systems," Programming and Computer Software, vol. 27, no. 5, pp. 260–276, 2001. [Online]. Available: http://dx.doi.org/10.1023/A\%3A1012392927006

[10] E. V. Kuzmin and V. A. Sokolov, "Modeling, specification, and verification of automaton programs," Programming and Computer Software, vol. 34, no. 1, pp. 27–43, 2008.

[11] V. Dubinin, S. Patil, C. Pang, and V. Vyatkin, "Neutralizing semantic ambiguities of function block architecture by modeling with asm," in Proceedings of PSI 2014: Ershov Informatics Conference (to appear), 2014.

[12] C.-H. Yang, V. Vyatkin, and C. Pang, "Model-driven development of control software for distributed automation: a survey and an approach," IEEE Transactions on Systems, Man and Cybernetics, vol. 44, no. 3, pp. 292–305, 2014.

[13] C. Yang and V. Vyatkin, "Transformation of simulink models to iec 61499 function blocks for verification of distributed control systems," Control Engineering Practice, vol. 20, no. 12, pp. 1259–1269, 2012.

[14] A. Zakonov and A. Shalyto, "Automatic extraction and verification of state-models for web applications," in Informatics in Control, Automation and Robotics, ser. Lecture Notes in Electrical Engineering, 2012, vol. 133, pp. 157–160.

[15] A. Klebanov, “Automata-based programming technology extension for generation of jml annotated java card code,” in Proceedings of the Second Spring Young Researchers Colloquium on Software Engineering, vol. 1, 2008, pp. 41–44.
[16] S. M. Lucas and T. J. Reynolds, “Learning deterministic finite automata with a smart state labelling evolutionary algorithm,” IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 27, pp. 1063–1074, 2005.
[17] ——, “Learning finite-state transducers: Evolution versus heuristic state merging,” IEEE Transactions on Evolutionary Computation, vol. 11, no. 3, pp. 308–325, Jun. 2007. [Online]. Available: http://dx.doi.org/10.1109/TEVC.2006.880329
[18] V. Levenshtein, “Binary Codes Capable of Correcting Deletions, Insertions and Reversals,” Soviet Physics Doklady, vol. 10, p. 707, 1966.
[19] W. Spears and G. D., “Evolving finite-state machine strategies for protecting resources,” Proceedings of the International Symposium on Methodologies for Intelligeng Systems., pp. 166–175, 2000.
[20] D. Chivilikhin and V. Ulyantsev, “MuACOsm: A New Mutation-Based Ant Colony Optimization Algorithm for Learning Finite-State Machines,” in Proceedings of the fifteenth annual conference on Genetic and evolutionary computation, ser. GECCO ’13. New York, NY, USA: ACM, 2013, pp. 511–518. [Online]. Available: http://doi.acm.org/10.1145/2463372.2463440
[21] V. Lumelsky and A. Stepanov, “Path planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitraty shape,” Algorithmica, vol. 2, pp. 403–430, 1987.
[22] I. Kamon, E. Rivlin, and E. Rimon, “A new range-sensor based globally convergent navigation algorithm for mobile robots,” in Proceedings of IEEE International Conference on Robotics and Automation, no. 1, 1996, pp. 429–435.
[23] G. Holzmann, “The model checker SPIN,” IEEE Transactions on Software Engineering, vol. 23, no. 5, pp. 279–295, May 1997.

Приложение: LTL-спецификация

Спецификация приведена на языке Promela. В Promela приняты следующие обозначения темпоральных операторов: «[]» означает оператор «Globally», «<>» означает оператор «Future». В целях улучшения читаемости формул, для некоторых утверждений были определены макросы (в Promela макросы определяются так же, как в языке C), которые используются в формулах. Каждая формула начинается с ключевого слова «ltl», после него идёт название формулы, далее сама формула в фигурных скобках.

```
typedef Profile
{
    unsigned action: 3; //current action
    unsigned lastAction: 3; //the last action
    unsigned direction: 2; //direction
    bit brick [9]; //neighboring cells
    unsigned target: 4; //direction of the target
    bool moving;
    bool afterMoving;
}

#define dt_rot2 (((p.action == RotLeft) ||\
    (p.action == RotRight) || (p.action ==\
    SavePosition) || (p.action == NoAction)) ||\
    !p.moving)
#define o2 ((p.brick[5] == 1) &&\
    (p.brick[2] == 0))

#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))
//the same idea for _right, _down, _left

ltl f0 {[] detour_pos_up}
ltl f1 {[] detour_pos_right}
ltl f2 {[] detour_pos_down}
ltl f3 {[] detour_pos_left}

#define o3 ((p.brick[5] == 1) &&\
    (p.brick[2] == 1))

#define dt2_pos_base_up (o3 && (p.action !=\
    Forward) && (p.direction == DirUp) &&\
    detourWall && p.moving)
#define dt2_pos_act_up ((p.direction == DirLeft)\
    || ((p.action==TargetAchieved) || (p.action\
    == TargetUnreachable)))
#define detour2_pos_up (dt2_pos_base_up -> \
    (dt_rot2 U dt2_pos_act_up))
//the same idea for _right, _down, _left

ltl f4 {[] detour2_pos_up}
ltl f5 {[] detour2_pos_right}
ltl f6 {[] detour2_pos_down}
ltl f7 {[] detour2_pos_left}

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

#define dt_begin_base_up (o0 && (p.direction ==\
    DirUp) && p.afterMoving)
#define dt_begin_act_up ((p.direction==DirLeft)\
    || (p.action == TargetUnreachable) || no_obs)
#define detour_begin_up (dt_begin_base_up -> \
    (dt_rot2 U dt_begin_act_up))
//the same idea for _right, _down, _left

ltl f8 {[] detour_begin_up}
ltl f9 {[] detour_begin_right}
ltl f10 {[] detour_begin_down}
ltl f11 {[] detour_begin_left}

#define o4 ((p.brick[2] == 0) &&\
    (p.brick[3] == 0) && (p.brick[5] == 1))

#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 == DirRight)\
    && (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))))
//the same idea for _right, _down, _left

ltl f12 {[] detour3_up}
ltl f13 {[] detour3_right}
ltl f14 {[] detour3_down}
ltl f15 {[] detour3_left}

#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)))
// the same idea for C_right, C_down, C_left
#define C (C_up || C_right || C_down || C_left)

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

#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)))))}

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

#define ActUp     ((p.direction == DirUp) \
    && (p.action == Forward))
//the same idea for ActRight, ActDown, ActLeft
#define NotGoUp ((!p.moving) || (!ActUp))
//the same idea for NotGoDown, NotGoLeft,
//                  NotGoRight

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)))}
ltl f27 {[] ((p.action == SavePosition) -> \
    (InSavedCell || IS_BETTER_THAN_SAVED))}
ltl f28 {[] ((!(<>(p.target == 0))) ->\
    <>((p.action == TargetUnreachable) ||\
    (([]<> (p.action == SavePosition))\
    && ([]<> !InSavedCell)) || ([](((p.action\
    == Forward) -> IS_MOVE_FORWARD_COOL)) || \
    (<>[](detourWall && !InSavedCell)))))}
ltl f29 {[] (IS_BETTER_THAN_SAVED ->\
    <>((p.action==TargetAchieved) || ((p.action\
    == SavePosition)&&(IS_BETTER_THAN_SAVED))))}
ltl f30 {[] ((p.action == TargetAchieved) ->\
    Reached)}
```