Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

“САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ”

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

ПРИМЕНЕНИЕ МЕТОДОВ ПРОГРАММИРОВАНИЯ В ОГРАНИЧЕНИЯХ ДЛЯ ГЕНЕРАЦИИ КОНЕЧНО-АВТОМАТНЫХ МОДЕЛЕЙ КОНТРОЛЛЕРОВ В ЗАДАЧАХ ЛОГИЧЕСКОГО УПРАВЛЕНИЯ

Автор: *Чухарев Константин Игоревич* _____
(Подпись)

Направление подготовки (специальность): *15.03.06 Мехатроника и робототехника*

Квалификация: *бакалавр*

Руководитель: *Ульянцев В.И., канд.техн.наук* _____
(Подпись)

К защите допустить

Зав. Кафедрой: *Бобцов А.А., профессор, д.т.н* _____
(Подпись)

“____”_____ 2018 г.

Санкт-Петербург, 2018 г.

Студент: Чухарев К.И. **Группа:** Р3435 **Кафедра:** СУиИ **Факультет:** СУиР

Направленность (профиль), специализация: *15.03.06 Интеллектуальные технологии в робототехнике*

Консультант (ы):

а) *Чивилихин Д. С., канд. техн. наук* _____
(подпись)

ВКР принята “___”_____ 20___г.

Оригинальность ВКР _____%

ВКР выполнена с оценкой _____

Дата защиты “___”_____ 20___г.

Секретарь ГЭК _____ _____
(ФИО) (подпись)

Листов хранения _____

Демонстрационных материалов/Чертежей хранения _____

Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

“САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ”

УТВЕРЖДАЮ

Зав. кафедрой СУиИ

Бобцов А.А.

(ФИО) (подпись)

«____» «_____» 2018 г.

З А Д А Н И Е
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ

Студент Чухарев К.И. **Группа** P3435 **Кафедра** СУиИ **Факультет** СУиР

Руководитель Ульянцев Владимир Игоревич, канд. техн. наук, доцент кафедры КТ

1 Наименование темы: Применение методов программирования в ограничениях для генерации конечно-автоматных моделей контроллеров в задачах логического управления

Направление подготовки (специальность) 15.03.06 Мехатроника и робототехника

Направленность (профиль) Интеллектуальные технологии в робототехнике

Квалификация бакалавр

2 Срок сдачи студентом законченной работы «___» «_____» 2018 г.

3 Техническое задание и исходные данные к работе

В рамках работы требуется разработать методы генерации минимальных конечно-автоматных моделей контроллеров для задач логического управления. Требуется изучить существующие методы синтеза конечно-автоматных моделей, средства решения задач SAT и CSP. Провести сбор сценариев работы (примеров поведения) симуляционной модели контроллера, управляющего манипулятором, выполненной в виде функционального блока. Провести сравнение разработанных методов с существующими на собранных примерах поведения.

4 Содержание выпускной квалификационной работы (перечень подлежащих разработке вопросов)

а) Обзор предметной области

б) Разработка методов синтеза минимальных конечно-автоматных моделей контроллеров для задач логического управления

в) Реализация предложенных методов

г) Проведение экспериментального исследования – сбор примеров поведения контроллера, синтез конечно-автоматной модели, верификация и сравнение результатов

5 Перечень графического материала (с указанием обязательного материала)

Не предусмотрено

6 Исходные материалы и пособия

1. Vyatkin V. IEC 61499 as Enabler of Distributed and Intelligent Automation: State-of-the-Art Review // *IEEE Transactions on Industrial Informatics*, vol. 7, no. 4, pp. 768–781, 2011.
2. Chivilikhin D. et al. CSP-based inference of function block finite-state models from execution traces // *IEEE 15th International Conference on Industrial Informatics (INDIN)*, Emden, pp. 714–719, 2017.
3. Шалыто А.А. Логическое управление. Методы аппаратной и программной реализации. СПб.: Наука, 2000. 780 с.
4. Biere A., Heule M., van Maaren H. Handbook of satisfiability // *IOS press*, 2009. 980 p.

КАЛЕНДАРНЫЙ ПЛАН

№№ п/п	Наименование этапов выпускной квалификационной работы	Срок выполнения этапов работы	Отметка о выполнении, подпись руков.
1	Обзор предметной области		
2	Разработка методов синтеза минимальных конечно-автоматных моделей контроллеров для задач логического управления		
3	Реализация предложенных методов		
4	Проведение экспериментального исследования – сбор примеров поведения контроллера, синтез конечно-автоматной модели, верификация и сравнение результатов		

8 Дата выдачи задания «____» «_____» 20 18 г.

Руководитель _____
(подпись)

Задание принял к исполнению _____ «____» «_____» 20 18 г.
(подпись)

Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

“САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ”

АННОТАЦИЯ

ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

Студент: Чухарев Константин Игоревич

Наименование темы ВКР: Применение методов программирования в ограничениях для генерации конечно-автоматных моделей контроллеров в задачах логического управления

Наименование организации, где выполнена ВКР: Университет ИТМО

ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

1 Цель исследования: разработка методов генерации минимальных конечно-автоматных моделей контроллеров по примерам поведения.

2 Задачи, решаемые в ВКР: 1) расширение существующего двухэтапного метода fbCSP, а именно разработка метода генерации базовой конечно-автоматной модели на основе сведения к задаче SAT, а также разработка точного метода минимизации охранных условий на основе сведения к задаче CSP; 2) разработка совмещенного одноэтапного метода генерации минимальной конечно-автоматной модели на основе сведения к задаче SAT

3 Число источников, использованных при составлении обзора: 25

4 Полное число источников, использованных в работе: 30

5 В том числе источников по годам

Отечественных			**Иностранных**		
Последние 5 лет	От 5 до 10 лет	Более 10 лет	Последние 5 лет	От 5 до 10 лет	Более 10 лет
2	2	4	8	6	8

6 Использование информационных ресурсов Internet: да, две ссылки на сторонние электронные ресурсы, одна ссылка на собственную программную реализацию.

7 Использование современных пакетов компьютерных программ и технологий: пакет утилит по автоматической визуализации графов graphviz (главы 2 и 3), среда разработки распределенных систем с использованием стандарта IEC 61499 nxtSTUDIO (глава 3), SAT-решатели glucose, cadical, lingeling, CSP-решатели Minizinc, OR-Tools.

8 Краткая характеристика полученных результатов: в ходе работы были предложены и реализованы методы построения минимальных конечно-автоматных моделей контроллеров по примерам поведения. Проведено экспериментальное исследование, подтвердившее работоспособность реализованных методов.

9 Полученные гранты, при выполнении работы: _____

(Название гранта)

10 Наличие публикаций и выступлений на конференциях по теме выпускной работы: да.

1 Чухарев К.И., Чивилихин Д.С., Ульянцев В.И. Построение минимальных конечно-автоматных моделей функциональных блоков по обучающим примерам // Сборник тезисов докладов конгресса молодых ученых. Электронное издание [Электронный ресурс]. - Режим доступа: http://openbooks.ifmo.ru/ru/file/8061/8061.pdf, своб.

Выпускник: Чухарев Константин Игоревич _____

(подпись)

Руководитель: Ульянцев Владимир Игоревич _____

(подпись)

“_____” _____ 20_18_г.

Содержание

Перв. примен.
Справ. №
Подп. и дата
Инв. № дубл.
Взам. инв. №
Подп. и дата
Инв. № подл.

Изм	Лист	№ докум.	Подп.	Дата	КСУИ.111.3435.001 ПЗ		
Разраб.		Чухарев К.И.			Применение методов программирования в ограничениях для генерации конечно-автоматных моделей контроллеров в задачах логического управления Пояснительная записка	Лит.	Лист 4	Листов 44
Проверил		Ульянцев В.И.				Д		
Н. контр.		Ведяков А.А.			Университет ИТМО кафедра СУиИ, гр. 3435			
Утв.		Бобцов А.А.						

Копировал Формат А4

Введение

Современные технологические процессы в своём большинстве частично или полностью *автоматизированы* – функции, ранее выполняемые человеком, переданы приборам и автоматам. Процесс автоматизации организован с помощью автоматизированных систем управления технологическим процессом (АСУТП), обладающих трехуровневой структурой [1]. В частности, на среднем уровне располагаются *регуляторы* и *программируемые логические контроллеры* (ПЛК), осуществляющие непосредственное *управление* оборудованием нижнего уровня, например, датчиками и исполнительными механизмами. Для полноты картины, контроллеры объединены промышленной сетью и взаимодействуют посредством системы SCADA (*Supervisory Control And Data Acquisition* – диспетчерское управление и сбор данных), а контроль за нижними уровнями осуществляется операторами через HMI (*Human Machine Interface* –человеко-машинный интерфейс).

При разработке систем управления для мехатронных и роботизированных механизмов, а также для систем автоматизированного производства, зачастую пользуются *логическим управлением* [2-6] – особым видом управления, в котором контроллер выдает двоичные управляющие сигналы основываясь на двоичных сигналах от объекта управления. Логика управления реализуется на таких языках как последовательные функциональные схемы (SFC), релейно-контактные схемы (LD), функциональные блоковые диаграммы (FBD) и исполняется на программируемых логических контроллерах (ПЛК). В этом подходе все непрерывные процессы, моделируемые с помощью стандартного аппарата теории управления (например, ПИД-регуляторы), вынесены в объект управления, имеющий чисто логический интерфейс.

Разработка контроллеров в логическом управлении, особенно для сложных и ответственных систем, является нетривиальной задачей. При этом разработку логического контроллера можно свести к созданию *конечного автомата* или системы из нескольких взаимодействующих автоматов, обеспечивающих требуемое поведение системы. Такой подход применяется, например, в международном

стандарте распределенных систем управления и автоматизации *IEC 61499* [7]. Автоматизация разработки контроллеров может быть достигнута путем применения и адаптации существующих методов и подходов к синтезу конечно-автоматных моделей [8].

В данной работе предлагается метод, позволяющий *автоматически* генерировать управляющие алгоритмы логических контроллеров по примерам поведения. В отличие от существующих методов, предлагаемый метод является *точным* и генерируемые модели являются *минимальными*, что обеспечивает их *обобщение*. В частности, разработанный метод можно применять для автоматизации создания дискретных управляющих программ.

1 Обзор предметной области

В данной главе приведен обзор автоматизированных систем управления, программируемых логических контроллеров, логического управления, конечно-автоматных моделей, стандарта распределенных систем управления и автоматизации IEC 61499, задач и методов генерации конечно-автоматных моделей, задач SAT и CSP, а также существующего метода генерации конечно-автоматных моделей контроллеров по примерам поведения.

1.1 Автоматизация процессов

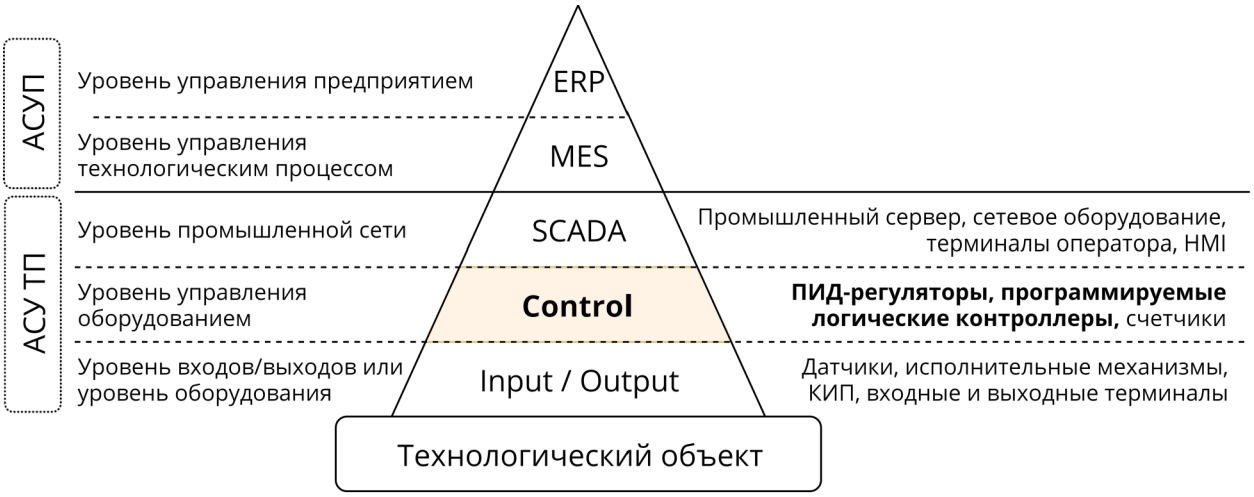

Рисунок 1 – Пирамида компьютерной автоматизации

Пирамида компьютерной автоматизации включает в себя пять уровней (рисунок 1) [1]. Два верхних уровня образуют автоматизированную систему управления предприятием (АСУП), а три нижних – автоматизированную систему управления технологическим процессом (АСУТП). Рассмотрим подробнее уровни АСУТП:

а) Нижний уровень – уровень оборудования или уровень входов-выходов. Здесь располагаются датчики (сенсоры), измерительные (КИП) и исполнительные (актюаторы) устройства, входные и выходные терминалы.

б) Средний уровень управления оборудованием включает в себя контроллеры (программируемые логические контроллеры; ПЛК), регуляторы, программируемые реле, счетчики. ПЛК управляют логикой управления и

обработки информации, получаемой от контрольно-измерительных приборов (КИП). Более подробно ПЛК описываются в разделе 1.2.

в) Верхний уровень промышленной сети состоит из промышленного сервера, сетевого оборудования и терминалов оператора. Здесь производится контроль за ходом выполнения технологического процесса, осуществляется связь с нижними уровнями. На этом уровне располагаются *SCADA* (система диспетчерского управления и сбора данных) и *HMI* (человеко-машинный интерфейс).

В данной работе рассматривается средний уровень, содержащий регуляторы и программируемые логические контроллеры.

1.2 Логическое управление на ПЛК

Программируемый логический контроллер (ПЛК) – промышленный компьютер, используемый для автоматизации технологических процессов. ПЛК состоит из процессорных модулей и модулей входов-выходов. К модулям входов-выходов подключаются непосредственно датчики и регулирующее оборудование, а в процессорные модули загружена логика автоматического регулирования и защитных блокировок. Частный случай управления, реализуемого с помощью ПЛК – *логическое управление*. Это особый вид управления, в котором контроллер оперирует логическими, то есть двоичными, управляющими сигналами.

Программная реализация управления требует не только оптимального потребления памяти и быстродействия, но и «понятности» и, самое главное, корректности, что связано с большой ответственностью разрабатываемых систем. Нетривиальный процесс разработки контроллера может быть сведен к построению конечного автомата или системы из нескольких взаимодействующих автоматов, обеспечивающих требуемое поведение системы. Применение конечных автоматов позволяет удобно визуализировать алгоритмы, решая проблему «понятности». Проверка корректности создаваемого логического контроллера традиционно производится путем симуляционного тестирования – для этого создается компьютерная симуляционная модель, например, в среде Matlab. Однако тестиро-

вание покрывает лишь ограниченный набор поведений, и потому его результаты не могут свидетельствовать об отсутствии ошибок. Поэтому тестирование необходимо дополнять формальной верификацией, позволяющей проверить все пространство состояний. Однако для применения формальной верификации требуется формальная модель системы. Среди математических объектов, используемых для формальной верификации систем, можно выделить конечные автоматы. Поэтому методы автоматической генерации конечных автоматов по примерам поведения могут быть использованы, в частности, для автоматического получения формальных моделей, что решает также проблему проверки корректности. Подход с использованием конечных автоматов для описания систем применяется, например, в международном стандарте распределенных систем управления и автоматизации IEC 61499, подробно описываемом в разделе 1.4.

1.3 Конечно-автоматные модели

В данном разделе рассмотрены различные виды конечно-автоматных моделей: детерминированные конечные автоматы (ДКА) и их расширенные версии – автоматы Мили и Мура, управляющие автоматы и диаграммы управления выполнением программы *(Execution Control Chart; ECC)*, использующиеся в IEC 61499.

Детерминированный конечный автомат (ДКА) – кортеж $(Q, q_s, \Sigma, \delta, F)$, где Q – множество состояний, $q_s \in Q$ – начальное состояние, Σ – множество входных символов (алфавит), $\delta : Q \times \Sigma \to Q$ – функция переходов, $F \subseteq Q$ – множество допускающих состояний.

Автомат Мили – кортеж $(Q, q_s, \Sigma, Z, \delta, \lambda)$, где в дополнение к определению ДКА вводится алфавит выходных символов Z и соответствующая функция выходов $\lambda : Q \times \Sigma \to Z$. *Автомат Мура* отличается тем, что выходные символы расположены в состояниях, а не на переходах – функция выходов выглядит следующим образом: $\lambda : Q \to Z$. Алфавит входных символов Σ также называется множеством входных событий.

Управляющий автомат (в английской литературе – *Extended Finite State Machine; EFSM*) – расширенная модель автомата Мили или Мура с возможностью

обработки входных переменных. Переходы дополняются охранными условиями, представляющими собой булеву функцию от входных переменных [9].

Диаграмма управления выполнением программы (Execution Control Chart; ECC) – автомат, отвечающий за функционал базового блока в стандарте IEC 61499, и являющийся расширением автомата Мура. При смене состояния не только формируется выходное событие, но и выполняется *алгоритм*, изменяющий значения выходных или внутренних переменных.

1.4 Стандарт распределенных систем управления и автоматизации IEC 61499

Стандарт распределенных систем управления и автоматизации IEC 61499 [7] является языком описания систем управления при помощи функциональных блоков, совмещающих логику управления и обработки информации с конечными автоматами, являющимися базовыми элементами программ.

Системы управления разрабатываются в виде соединенных между собой функциональных блоков (ФБ). Каждый ФБ характеризуется своим *интерфейсом* – он определяет входные и выходные события и переменные (рисунок 2). Каждая переменная имеет тип, например логический, целочисленный или действительный. В данной работе рассматривается только логическое управление – такое управление, при котором используются только булевы переменные (принимающие одно из двух возможных значений: True («истина») или False («ложь»)).

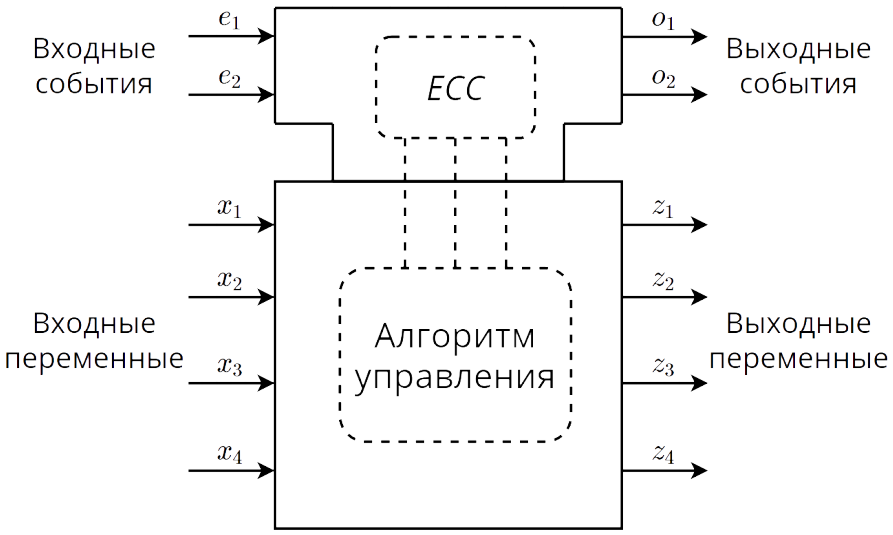

Рисунок 2 – Пример интерфейса функционального блока

Стандарт IEC 61499 описывает два основных типа функциональных блоков – составные и базовые. Составной ФБ является сетью других ФБ. Функционал базового ФБ определяется диаграммой управления выполнением программы (здесь и далее – *ECC*, рисунок 3), представляющей собой расширенный автомат Мура. Кроме входных и выходных переменных, являющихся входными и выходными алфавитами автомата, соответственно, функциональный блок может обладать внутренними переменными. В данной работе рассматривается класс ФБ только с входными и выходными переменными, без внутренних. Переходы *ECC* помечены так называемыми охранными условиями, представляющими собой булевы функции от входных, выходных или внутренних переменных. С каждым состоянием *ECC* ассоциировано «выходное действие» – пара из выходного события и *ECC*-алгоритма. Алгоритмы могут быть реализованы с использованием языков из стандарта *IEC 61131-3* (например, может быть использован язык *ST – Structured Text*) и представляют собой описание того, как изменяются значения выходных переменных при переходе в данное состояние автомата.

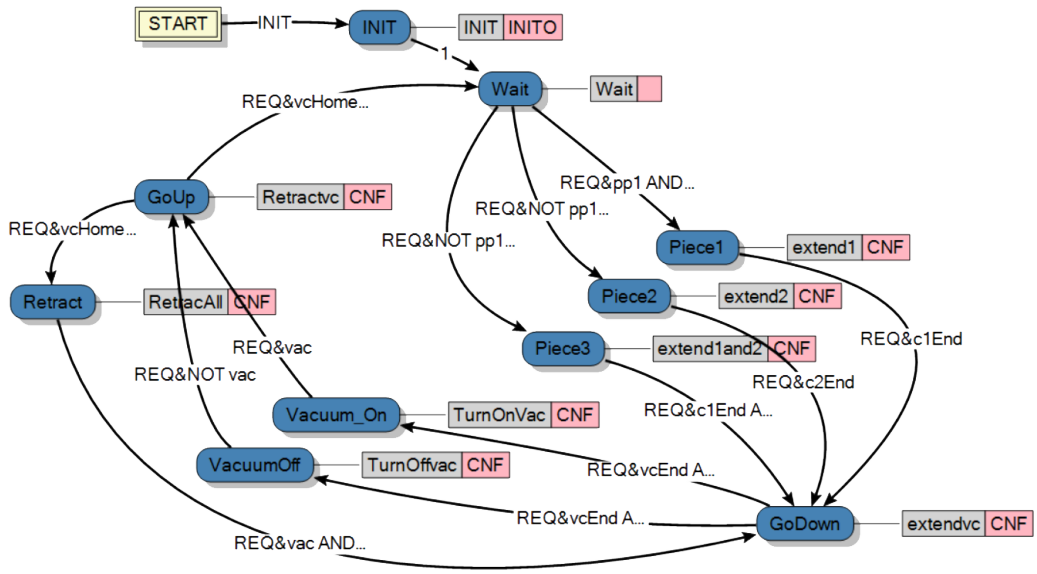

Рисунок 3 – Пример *ECC* контроллера, осуществляющего логическое управление Pick-and-Place манипулятором

1.5 Задачи генерации автоматов

Задача генерации конечных автоматов заключается в их построении по заданным входным данным, например, по примерам поведения (сценариям работы) или по темпоральным свойствам *(Linear Temporal Logic; LTL)* [10,11].

Задача синтеза автомата по примерам поведения заключается в построении автомата, который, как минимум, на каждое входное событие из примеров поведения будет реагировать должным образом. *Обобщением* будем называть процесс синтеза такого автомата, который работает должным образом и на тех примерах поведения, которые не были использованы при его построении. В данной работе обобщение достигается путем минимизации охранных условий.

1.6 Методы генерации автоматов

Альтернативой ручному построению конечно-автоматных моделей является их автоматический синтез по примерам поведения. Эта задача является NP-полной. Распространенным подходом к генерации конечных автоматов является применение метаэвристик, среди которых можно выделить эволюционные [12], генетические [13] и муравьиные [14] алгоритмы. Другой подход – эвристические методы (в английской литературе называемые «state merging»), основанные на объединении состояний [15].

В последнее время все чаще применяется подход *программирования в ограничениях* [11], заключающийся в *сведении* к NP-полным задачам SAT и CSP с последующим применением постоянно совершенствующихся специализированных программных средств – *решателей*.

1.7 Задачи SAT и CSP

Задача о выполнимости (Boolean satisfiability; SAT) формулируется следующим образом – дана булева формула ϕ в конъюктивной нормальной форме (КНФ), необходимо найти такую подстановку X_{SAT} значений переменных, что формула станет истинной $(\phi|_{X_{SAT}} = 1)$, или же доказать ее отсутствие [16]. Задача SAT является первой задачей, для которой была доказана NP-полнота [17].

Задача удовлетворения ограничений (Constraint Satisfiability Problem; CSP) является общим случаем задачи о выполнимости и расширяет задачу SAT произвольными типами переменных взамен только логических [18]. Даны переменные $x_1 \dots x_n$ и их области значений (домены). Задача удовлетворения ограничений заключается в нахождении таких значений переменных $x_1 \dots x_n$, при которых бы выполнялись все заданные ограничения. Формально, задача CSP – тройка $(\mathbb{V}, \mathbb{D}, \mathbb{C})$, где $\mathbb{V}$ – множество переменных, $\mathbb{D}$ – множество их доменов, $\mathbb{C}$ – множество ограничений.

Для решения задачи SAT применяют специализированные программные средства – *SAT-решатели*, например, MiniSat [19], CryptoMiniSat [20], Glucose, lingeling [21], CaDiCaL и множество других. Для описания задач CSP широко используется язык моделирования ограничений Minizinc [22], одновременно являющийся конвертером в низкоуровневый формат Flatzinc, позволяющий применять любой CSP-решатель, поддерживающий этот формат, например, Google OR-Tools [23], Gecode, Choco [24] и многие другие.

1.8 Метод генерации конечно-автоматных моделей контроллеров по примерам поведения на основе сведения к CSP

Отправная точка для исследований, проводимых в данной работе – статья «Chivilikhin et al. *CSP-based inference of function block finite-state models from execution traces*, 2017» [25] в которой предлагается метод построения конечно-автоматной модели контроллера с помощью сведения задачи к CSP. Рассмотрим подробнее этапы этого метода, который здесь и далее будем называть FBCSP.

Входные данные – набор примеров поведения, которые здесь и далее будем называть *сценариями*. Сценарий – последовательность элементов сценария $s_i = \left\langle e^I[\bar{x}], e^O[\bar{z}] \right\rangle$, каждый из которых задается входным событием $e^I \in E^I$, значениями входных переменных $\bar{x} \in \{0,1\}^{|X|}$, выходным событием $e^O \in E^O \cup \{\varepsilon\}$ и значениями выходных переменных $\bar{z} \in \{0,1\}^{|Z|}$. *Размер* (или *длина*) сценария – число элементов сценария. На рисунке 4 приведен пример двух сценариев S_1 и S_2, где $E^I = \{A\}$, $E^O = \{B\}$, $X = \{x_1, x_2\}$, $Z = \{z_1, z_2\}$.

$$S_1 = \Big[A[x_1 = 1, x_2 = 0], B[z_1 = 1, z_2 = 0]\rangle;$$
$$\langle A[x_1 = 1, x_2 = 1], B[z_1 = 1, z_2 = 1]\rangle;$$
$$\langle A[x_1 = 0, x_2 = 0], B[z_1 = 0, z_2 = 1]\rangle\Big]$$
$$S_2 = \Big[\langle A[x_1 = 0, x_2 = 1], B[z_1 = 0, z_2 = 1]\rangle;$$
$$\langle A[x_1 = 0, x_2 = 0], B[z_1 = 1, z_2 = 1]\rangle;$$
$$\langle A[x_1 = 1, x_2 = 1], B[z_1 = 1, z_2 = 0]\rangle\Big].$$

Рисунок 4 – Пример сценариев

Сценарии объединяются в *дерево сценариев*. Целесообразно использовать более эффективную древовидную структуру данных – *бор (префиксное дерево)*. В дереве в качестве вершин выступают элементы сценариев, общим корнем является дополнительная стартовая вершина, отвечающая за момент инициализации; вершина является потомком другой вершины в случае, если соответствующие элементы сценария были встречены последовательно в исходном наборе примеров поведения. На рисунке 5 приведено дерево сценариев, построенное по сценариям из примера (рисунок 4). На рисунке 6 показан автомат *(ECC)*, построенный по дереву сценариев с рисунка 5.

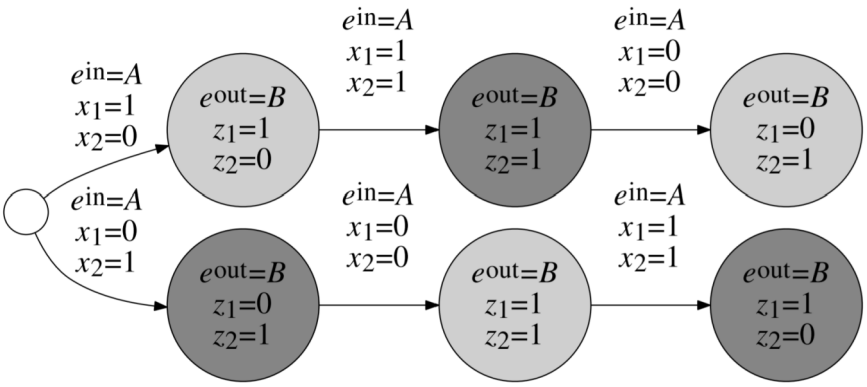

Рисунок 5 – Дерево сценариев, построенное по сценариям из примера (рисунок 4)

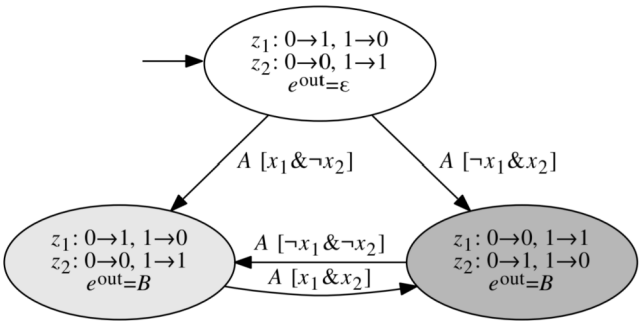

Рисунок 6 – Автомат *(ECC)*, построенный по дереву сценариев с рисунка 5

Как уже было описано в разделе 1.4, логика управления и обработки информации задается с помощью *ECC* – конечного автомата Мура, расширенного охранными условиями на переходах и алгоритмами в состояниях. Формальное определение *ECC* – это кортеж $(S, s_1, E^I, X, E^O, Z, \phi, g, \pi, \lambda)$, где S – множество состояний, s_1 – начальное состояние, E^I – множество входных событий, X – множество входных переменных, E^O – множество выходных событий, Z – множество выходных переменных, $\phi \subseteq S \to S$ – функция переходов, $g : \phi \to (E^I \times \{0,1\}^{|X|} \to \{0,1\})$ – охранные условия для каждого перехода, $\pi : \phi \to \{1, 2, \ldots\}$ – приоритеты переходов, $\lambda : Y \to (\{0,1\}^{|Z|} \to \{0,1\}^{|Z|}) \times (E^O \cup \{\varepsilon\})$ – выходные действия (события и алгоритмы).

Таким образом, задача формулируется следующим образом – синтезировать автомат *(ECC)*, удовлетворяющий заданным сценариям. Говорят, что автомат *удовлетворяет* элементу сценария $s_i = \langle e^I[\bar{x}], e^O[\bar{z}] \rangle$ в состоянии s, если при получении входного события e^I со значениями входных переменных $\bar{x}$ автомат генерирует выходное событие e^O и значения выходных переменных становятся равными $\bar{z}$. Автомат удовлетворяет сценарию, если он удовлетворяет всем элементам сценария в соответствующих состояниях. Описываемый метод FBCSP разбивает задачу на два этапа:

1) Построение минимальной базовой конечно-автоматной модели.
2) Минимизация охранных условий.

Первым шагом решения задачи является построение минимальной *базовой* конечно-автоматной модели. Для этого применяется программирование в ограничениях, а именно, *сведение* к задаче CSP – задача $(\mathcal{T}, N)$ преобразуется к задаче $(\mathbb{V}, \mathbb{D}, \mathbb{C})$, где $\mathcal{T}$ – дерево сценариев, N – число состояний автомата, $\mathbb{V}$ – множество *переменных*, $\mathbb{D}$ – множество областей определений (*доменов*) этих переменных, $\mathbb{C}$ – множество *ограничений* на эти переменные. Формируется задача *раскраски* дерева сценариев $\mathcal{T}$ в N цветов таким образом, что автомат, состояния которого получаются путем слияния всех вершин дерева одного цвета, *удовлетворяет* исходным сценариям и является детерминированным. Сформированная задача в формате

Flatzinc поступает на вход специализированного программного средства – *решателя* – и затем, собственно, решается. Возможны два исхода:

а) *«SAT»:* решатель нашел такую подстановку X_{SAT} значений переменных $\mathbb{V}$, что выполняются все наложенные ограничения $\mathbb{C}$. Искомый базовый автомат строится по переменным, отвечающим за его структуру, а именно, кодирующим состояния и переходы между ними.

б) *«UNSAT»:* решатель так или иначе проверил все пространство возможных значений переменных $\mathbb{V}$ и не нашел такой подстановки, что удовлетворяла бы наложенным ограничениям $\mathbb{C}$. В таком случае необходимо повторить решение с бóльшим числом состояний N.

Рисунок 7 иллюстрирует общий подход к итеративному построению минимальных базовых конечно-автоматных моделей с применением программирования в ограничениях.

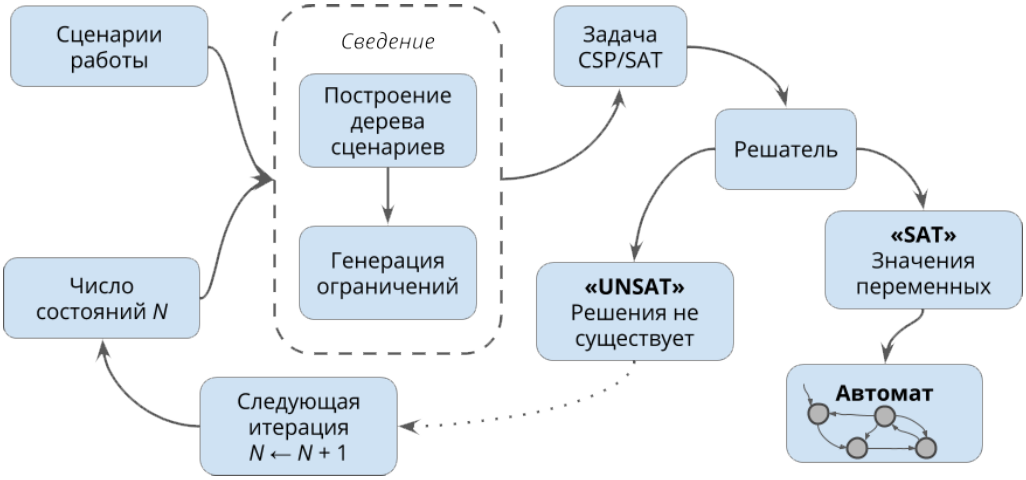

Рисунок 7 – Общий подход к построению минимальных базовых конечно-автоматных моделей с помощью программирования в ограничениях

Вторым шагом решения задачи является *минимизация* охранных условий. Изначально, охранные условия находятся в «полной» форме, являясь конъюнкцией всех возможных входных переменных (или их отрицаний). Для минимизации используется следущий *жадный* алгоритм. Итеративно предпринимается попытка исключения некоторого литерала из булевой формулы, и, если автомат продолжает удовлетворять исходным сценариям, то литерал исключается, иначе

выбирается другой литерал. Процесс завершается при невозможности исключить какой-либо литерал.

Отметим, что жадная эвристика не гарантирует получения минимально возможных охранных условий, поэтому в данной работе предлагается *точный* метод построения минимальных охранных условий.

1.9 Постановка задачи

Целью данной работы является разработка методов генерации минимальных конечно-автоматных моделей контроллеров по примерам поведения. Для достижения цели решаются две задачи.

1) Расширение существующего двухэтапного метода fbCSP [25]:
   а) Улучшение первого этапа – разработка метода генерации базовой конечно-автоматной модели на основе сведения к задаче SAT. Это позволит существенно ускорить решение задачи за счет применения производительных программных средств – *SAT-решателей*.
   б) Улучшение второго этапа – разработка точного метода минимизации охранных условий на основе сведения к задаче CSP. Это позволит получать не только точное, но и более минимальное решение.
2) Разработка совмещенного одноэтапного метода генерации минимальной конечно-автоматной модели на основе сведения к задаче SAT. Это позволит получать наиболее минимальное решение, по сравнению с двухэтапным методом.

2 Предлагаемые методы генерации минимальных конечно-автоматных моделей контроллеров по примерам поведения

В данной главе описываются разработка и реализация точных методов генерации минимальных конечно-автоматных моделей контроллеров по примерам поведения, а именно, усовершенствования двух этапов существующего метода fbCSP и предлагаемый одноэтапный метод fbSAT. В разделе 2.1 приводится описание метода построения *базовых* конечно-автоматных моделей, основанного на сведении к SAT. В разделе 2.2 предлагается *точный* метод минимизации охранных условий. В разделе 2.3 описывается *одноэтапный* метод генерации минимальных конечно-автоматных моделей.

2.1 Метод генерации базовых конечно-автоматных моделей на основе сведения к SAT

В данном разделе предлагается метод генерации *базовых* конечно-автоматных моделей контроллеров по примерам поведения, основанный на сведении к задаче SAT, что отличает его от существующего метода fbCSP, который в свою очередь основан на сведении к CSP. Подобная замена CSP на SAT обусловлена значительно бóльшей производительностью SAT-решателей. Главное отличие SAT от CSP – возможность использования только логических (Булевых) переменных. Также стоит отметить, что CSP-решатели обладают возможностями оптимизации целевой функции – в SAT-решателях похожую роль играет *инкрементальное решение*, рассматриваемое в разделе 2.3.

Напомним, что для построения базовой конечно-автоматной модели контроллера по примерам поведения используется следующих подход:

- на основе сценариев работы строится дерево сценариев;
- формируется задача о раскраске дерева сценариев в C цветов, где C – перебираемое число состояний искомого автомата;
- структура автомата кодируется переменными, на которые накладываются некоторые ограничения, среди которых требование удовлетворения автоматом исходных сценариев работы.

Будем называть рассматриваемый в данном разделе алгоритм построения минимальной базовой конечно-автоматной модели, основанный на сведении к SAT, как «алгоритм SATGEN», псевдокод которого приведен в листинге 1.

Algorithm 1: basic FB model inference

```
Data: scenarios set 𝒮
𝒯 ← buildScenarioTree(𝒮)
foreach C ← 1 to ∞ do
    K′ ← C
    𝒜′ ← findModel(𝒯, C, K′)
    if 𝒜′ ≠ null then
        foreach K ← 1 to C do
            𝒜 ← findModel(𝒯, C, K)
            if 𝒜 ≠ null then return 𝒜
```

Листинг 1 – Псевдокод алгоритма SATGEN

2.1.1 Описание сведения

В данном подразделе описывается сведение задачи построения базовой конечно-автоматной модели, с заданным числом состояний C и максимальным числом переходов из каждого состояния K, к задаче SAT. Здесь и далее будем называть этот алгоритм $\texttt{findModel}(\mathcal{T}, C, K)$, где $\mathcal{T}$ – дерево сценариев.

В дальнейшем будут использованы следующие обозначения:

- V – множество вершин дерева сценариев, пронумерованных от корня в порядке DFS;
- $V^{(\text{active})} = \{v \,|\, \text{toe}_v \neq \varepsilon,\ v \in V$ – множество «активных» вершин дерева сценариев, вырабатывающих непустые выходные события;
- $V^{(\text{passive})} = \{v \,|\, \text{toe}_v = \varepsilon,\ v \in V\}$ – множество «пассивных» вершин дерева сценариев, не вырабатывающих выходных событий;
- E – упорядоченное множество входных событий;
- O – упорядоченное множество выходных событий;
- X – упорядоченное множество входных переменных;
- Z – упорядоченное множество выходных переменных;
- U – упорядоченное множество наборов входных переменных;
- Y – упорядоченное множество наборов выходных переменных.

Далее приведены массивы, описывающие структуру построенного дерева сценариев, а также их синонимы, которые будут использованы для краткости описания:

- $\mathrm{tp} = \mathrm{tree\_parent}_{v \in V} \in V \cup \{\varepsilon\}$ – родитель вершины;
- $\mathrm{tpa} = \mathrm{tree\_previous\_active}_{v \in V} \in [0..V]$ – ближайший «активный» предок;
- $\mathrm{tie} = \mathrm{tree\_input\_event}_{v \in V} \in E$ – входное событие, которым помечено входящее ребро вершины дерева;
- $\mathrm{toe} = \mathrm{tree\_output\_event}_{v \in V} \in O \cup \{\varepsilon\}$ – выходное событие; «пассивные» вершины не вырабатывают выходных событий и помечены ε;
- $\mathrm{input\_number}_{v \in V} \in U$ – набор входных переменных, которыми помечено входящее ребро вершины дерева;
- $\mathrm{output\_number}_{v \in V} \in Y$ – набор выходных переменных вершины дерева;
- $\mathrm{unique\_input}_{u \in U,\ x \in X} \in \{\mathrm{True}, \mathrm{False}\}$ – значение переменной x из набора входных переменных u;
- $\mathrm{unique\_output}_{y \in Y,\ z \in Z} \in \{\mathrm{True}, \mathrm{False}\}$ – значение переменной z из набора выходных переменных y.

Для кодирования структуры синтезируемого автомата, удовлетворяющего всем исходным сценариям выполнения, потребуются следующие переменные:

- $c_{v,c}$ – является ли $c \in [1..C]$ цветом вершины $v \in V$ дерева сценариев;
- t_{c_1,e,u,c_2} – существует ли переход из состояния автомата $c_1 \in [1..C]$ в состояние $c_2 \in [1..C]$ по входному событию $e \in E^I$, помеченный охранным условием $u \in U$;
- $d^0_{c,z}, d^1_{c,z}$ – тернарный алгоритм для обновления значения выходной переменной $z \in Z$ в состоянии $c \in [1..C]$;
- $o_{c,o}$ – является ли $o \in O$ выходным событием состояния $c \in [1..C]$.

Для отражения необходимых требований к структуре синтезируемого автомата, потребуется наложение ограничений на описанные выше переменные. Полный список ограничений приведен в приложении А.

2.1.2 Алгоритм перебора параметров

Имея возможность генерировать конечно-автоматную модель с заданным числом состояний C и максимальным числом переходов из каждого состояния K, остановимся подробнее на нахождении минимальной конечно-автоматной модели. Пусть минимальная модель обладает следующими параметрами: $C_{\min}$ и $K_{\min}$, которые здесь и далее будем называть *«минимальными»*, тогда известно, что вызов $\texttt{findModel}(\mathcal{T}, C_{\min}, K_{\min})$ найдет эту модель, а вызовы со значениями параметров меньше минимальных обнаружат отсутствие решения: $\forall C < C_{\min}\ \forall K < K_{\min}\ \texttt{findModel}(\mathcal{T}, C, K) = \texttt{null}$. Заметим, что решение будет также отсутствовать при произвольном параметре K, в случае, когда $C < C_{\min}$, значит, параметр C «важнее» K.

В качестве алгоритма поиска минимальных параметров предлагается использовать линейный поиск – последовательный перебор значений параметров до тех пор, пока не будет найдено минимальное решение. Рассмотрим подробнее приведенный ранее листинг 1 алгоритма SATGEN. Внешний цикл отвечает за перебор параметра C от 1 и выше, вызывая алгоритм `findModel` с параметром $K = C$. Пока $C < C_{\min}$, решение будет отсутствовать. Как только находится решение с $C = C_{\min}$, начинается выполнение вложенного цикла перебора параметра K от 1 до C, что по аналогичной причине $(K < K_{\min})$ будет приводить к отсутсвию решения, пока перебор не дойдет до искомого минимального параметра $K = K_{\min}$.

2.1.3 Охранные условия

В построенной базовой конечно-автоматной модели охранные условия кодируются индексами $u \in U$ переменной t и являются «полными» – соответствующая булева функция представляет собой конъюнкцию всех входных переменных или их отрицаний. Такой подход к заданию охранных условий полностью исключает возможность *обобщения*, а именно, базовая модель будет исправно работать только на примерах поведения, которые были использованы при ее построении. Поэтому предлагается дополнить метод построения минимальной конечно-автоматной модели *вторым этапом*, который, в отличие от рассмотренного в раз-

деле 1.8 метода FBCSP, будет не эвристическим, а точным, что не только позволит быть уверенными в достоверности ответа, но и получать гораздо более минимальные решения.

2.2 Точный метод минимизации охранных условий

В данном разделе предлагается *точный* метод минимизации охранных условий, основанный на сведения к задаче CSP. Заметим, что описываемый метод оперирует базовым автоматом $\mathcal{A}$, построенным с помощью метода SATGEN (см. предыдущий раздел 2.1). Охранные условия базового автомата являются полными, из-за чего автомат хоть и является полностью работоспособным на тех примерах поведения, которые были использованы для его построения, но в подавляющем большинстве случаев работает неверно в новых ситуациях. Будем называть *обобщением* обратное поведение – когда автомат работает «хорошо» даже в ситуациях, которые не были примерами для его построении. В данной работе обобщение достигается через *минимизацию* охранных условий.

Для обсуждения минимизации охранных условий необходимо сначала определить их представление. Будем задавать охранные условия явным образом – в виде *дерева разбора* соответствующей Булевой функцию от входных переменных. На рисунке 8 показан пример дерева разбора Булевой функции $(x_1 \vee x_4) \wedge \neg x_5$. Символами $\wedge$, $\vee$ и $\neg$ отмечены вершины, соответствующие логическим операторам И, ИЛИ и НЕ, а вершины терминалы отмечены соответствующими входными переменными, например, x_4.

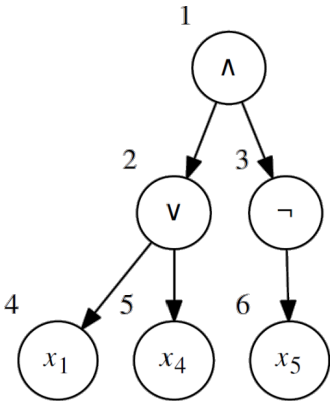

Рисунок 8 – Пример дерева разбора булевой функции $(x_1 \vee x_4) \wedge \neg x_5$ размера $P = 6$

Говоря неформально, задача состоит в том, чтобы генерировать конечный автомат с максимально простыми охранными условиями. Сложность может быть измерена по-разному, и в данной работе в качестве метрики минимизации используется *размер* – суммарное число вершин в деревьях разбора всех охранных выражений в автомате.

Рассмотрим состояние s автомата $\mathcal{A}$ с K' переходами в K других состояний $s'_1 \dots s'_K$ ($K' \geq K$, так как возможно несколько различных переходов из одного состояния в другое, с различными охранными условиями, что сохраняет автомат детерминированным. «Запустим» сценарии работы на автомате, в результате чего получим множество D элементов сценария, обрабатываемых в рассматриваемом состоянии s. Для каждого элемента также получим следующую информацию:

- $\delta_d \in [0..K]$ – индекс $k \in [1..K]$, если при обработке элемента сценария $d \in D$ происходит переход в состояние s_k, или $k = 0$, если перехода не происходит.
- $\omega_{d,x} \in \{\mathrm{True}, \mathrm{False}\}$ – значение входной переменной $x \in X$ перед обработкой элемента сценария $d \in D$. Будем обозначать массив значений $\omega_{d,1} \dots \omega_{d,|X|}$ как ω_d.

Заметим, что для каждого состояния решением задачи, по сути, является K булевых функций $f_1 \dots f_K$ ($f_k : \{0,1\}^{|X|} \to \{0,1\}$), выполняющих роль охранных условий на переходах в К других состояний. На эти функции накладываются следующие ограничения:

- $\forall d \in D\ \delta_d \neq 0 \to f_{\delta_d}(\omega_d) \wedge \bigwedge_{1 \leq k < \delta_d} \neg f_k(\omega_d)$ – если переход выполняется, то выполняется и соответствующее охранное условие, а все остальные охранные условия не выполняются.
- $\forall d \in D\ \delta_d = 0 \to \bigwedge_{1 \leq k \leq K} \neg f_k(\omega_d)$ – если переход не выполняется, то и не выполняется ни одно охранное условие.

Охранные условия задаются в виде деревьев разбора, состоящих не более чем из P вершин. Заметим, что все переходы из одного состояния являются независимыми, а значит, элементарной подзадачей является построение минимального

охранного условия, удовлетворяющего описанным выше ограничениям. Будем называть рассматриваемый в данном разделе точный алгоритм минимизации охранных условий базовой конечно-автоматной модели, основанный на сведении к CSP, как алгоритм TRANSITION-WISE, псевдокод которого приведен в листинге 2.

Algorithm 2: minimal guard conditions inference

Data: automata $\mathcal{A}$ with C states and with at most K transitions from each state
foreach $c \leftarrow 1$ **to** C **do**
  foreach $k \leftarrow 1$ **to** K **do**
    foreach $P \leftarrow 1$ **to** ∞ **do**
      $\mathcal{G} \leftarrow$ `findGuard`$(\mathcal{A}, c, k, P)$
      if $\mathcal{G} \neq$ *null* **then**
        augment $\mathcal{A}$ with $\mathcal{G}$
        break

Листинг 2 – Псевдокод алгоритма TRANSITION-WISE

2.2.1 Описание сведения

В данном подразделе описывается сведение задачи построения охранного условия размера P, которым помечен *k*-тый переход из состояния c базовой конечно-автоматной модели $\mathcal{A}$, к задаче CSP. Здесь и далее будем называть этот алгоритм `findGuard`$(\mathcal{A}, c, k, P)$.

Рассмотрим подробнее переменные, кодирующие структуру дерева разбора охранного условия размера P:

- Тип каждой вершины $p \in [1..P]$ задается переменной $\mathrm{nodetype}_p \in [0..3]$, где 0 – терминал, соответствующий входной переменной, $1..3$ – не-терминалы, соответствующие логическим операциям И ($\wedge$), ИЛИ ($\vee$) и НЕ ($\neg$).
- Каждая вершина $p \in [1..P]$ для каждого элемента сценария $d \in D$ обладает вычислимым булевым значением $\mathrm{value}_{p,d} \in \{\mathrm{True}, \mathrm{False}\}$. Значения терминальных вершины равны значениям соответствующих входных переменных; значения не-терминальных – вычисляются путем применения соответствующих логических операций над значениями дочерних вершин.
- С терминальными вершинами ассоциированы входные переменные, за что отвечает переменная $\mathrm{terminal}_p \in [0..|X|]$. С не-терминалами переменные не ассоциированы, поэтому для них значение переменной равно 0.

– За родительские отношения между вершинами отвечает переменная $\mathrm{parent}_p \in [0..P]$, а за дочерние – переменная $\mathrm{child}_p \in [0..P]$, кодирующая левого ребенка вершины p. Нумерация в дереве происходит в порядке *BFS*-обхода, подробно описываемого в разделе 2.3, поэтому априори известно, что правый потомок, если он вообще присутствует, имеет номер на 1 больше левого.

– Вместо прямых обращений к значению потомка (вида $\mathrm{value}_{\mathrm{child}_p,d}$) вводятся дополнительные переменные $\mathrm{value}^{\mathrm{left}}_{p,d}$ и $\mathrm{value}^{\mathrm{right}}_{p,d}$, кодирующие непосредственно значения левого и правого потомков.

Ограничения на описанные выше переменные записаны на языке моделирования ограничений Minizinc [22] – *mzn*-файл модели приведен в приложении Б.

2.2.2 Алгоритм перебора параметров

Имея возможность построения охранного условия *k*-го перехода из состояния c и заданного размера P с помощью алгоритма $\texttt{findGuard}(\mathcal{A}, c, k, P)$, рассмотрим задачу минимизации охранных условий. По аналогии с разделом 2.1.2, допустим, что минимальные охранные условия имеют размер $P_{\min}$, тогда известно, что вызов $\texttt{findGuard}(\mathcal{A}, c, k, P_{\min})$ найдет их, а вызовы с меньшими значениями P обнаружат отсутствие решения: $\forall P < P_{\min}\ \texttt{findGuard}(\mathcal{A}, c, k, P) = \texttt{null}$. Перебор размера охранных условий P организован в виде линейного поиска – от 1 и выше – пока не будет найдено решение, которое и будет являться минимальным.

2.2.3 Предпосылки к совмещению двух этапов

Заметим, что два метода, рассмотренные в разделах 2.1 и 2.2, являются последовательными этапами двухэтапного метода построения минимальной конечно-автоматной модели контроллера по примерам поведения. Однако, получаемые модели не являются абсолютно минимальными вследствие *потери информации* между этапами. Поэтому предлагается совместить два этапа в один, что подробно рассматривается в разделе 2.3.

2.3 Совмещенный метод генерации минимальных конечно-автоматных моделей

В данном разделе предлагается *одноэтапный* метод генерации минимальных конечно-автоматных моделей, основанный на сведении к задаче SAT. Суть метода заключается в построении совмещенной КНФ-формулы для SAT-решателя, которая кодирует одновременно и структуру автомата (состояния и переходы), и охранные условия (явные деревья разбора соответствующих Булевых функций). Заметим, что метод является самодостаточным и способен синтезировать конечно-автоматную модель «с нуля» – по сценариям исполнения, однако, целесообразно использование двухэтапного метода для оценок параметров синтезируемого автомата – числа состояний C, максимального числа переходов K из каждого состояния и максимального размера P каждого дерева разбора. Это обусловлено тем, что значительная часть времени работы алгоритма – перебор параметров, то есть подтверждение отсутствия решения при параметрах меньше минимальных, а этапы двухэтапного метода решаются SAT-решателем значительно быстрее, чем цельная КНФ-формула. Отметим также, что получаемые таким образом оценки параметров C и K являются минимальными ($\hat{C} = C_{\min}, \hat{K} = K_{\min}$), а оценка $\hat{P}$ является верхней границей параметра P.

Будем называть предлагаемый в данном разделе совмещенный одноэтапный метод генерации минимальных конечно-автоматных моделей, основанный на сведении к SAT, как алгоритм FBSAT, псевдокод которого приведен в листинге 3.

Algorithm 3: minimal FB model inference

```
Data: scenarios set 𝒮
𝒯 ← buildScenarioTree(𝒮)
C, K ← estimate from SATGEN(𝒮)
P ← estimate from TRANSITION-WISE(𝒮)
foreach N ← C · K · P downto 1 do
    𝒜 ← buildModel(𝒯, C, K, P, N)
    if 𝒜 = null then
        return 𝒜_last
    else
        𝒜_last ← 𝒜
```

Листинг 3 – Псевдокод алгоритма FBSAT

2.3.1 Описание сведения

В данном подразделе описывается сведение задачи построения конечно-автоматной модели – с заданным числом состояний C, максимальным числом K переходов из каждого состояния, максимальным размером P каждого дерева разбора и максимальным числом N типизированных вершин всех деревьев разбора – к задаче SAT. Будем называть этот алгоритм `buildModel`$(\mathcal{T}, C, K, P)$, где $\mathcal{T}$ – дерево сценариев. Заметим, что описываемое сведение включает в себя большинство переменных и ограничений с рядом модификаций из двух рассмотренных ранее этапов. Поэтому, для краткости содержания, рассмотрим подробно только некоторые отличия.

Так как все охранные условия строятся одновременно, в отличие от алгоритма TRANSITION-WISE, то параметр P является числом вершин в каждом дереве разбора. Однако, нас интересует минимальное решение, и в большинстве случаев все P вершин требуются только некоторым охранным условиям для кодирования сложных булевых функций. В остальных случаях, часть вершин является «лишней» для дерева разбора, поэтому такие вершины помечаются как нетипизированные, что в сведении кодируется типом 4 – $\mathrm{nodetype} \in [0..4]$.

Ограничение на то, что суммарное число типизированных вершин во всех деревьях разбора не больше N, выглядит следующим образом: $\sum_{c\in[1..C],k\in[1..K],p\in[1..P]}(\neg\mathrm{nodetype}_{c,k,p,4}) \leq N$. Для того, чтобы закодировать такую запись в SAT, применяется способ кодирования *ограничений на мощность (cardinality constraints)* из [26], который заключается в следующем. Дано множество, ограничение на мощность которого необходимо закодировать с помощью КНФ-формулы. Принцип кодирования заключается в формировании унарной записи числа, обозначающего сумму элементов заданного множества, с последующим наложением ограничений на эту запись. Следуя нотации оригинальной статьи, будем называть заданное множество – множество *входных переменных* – и обозначать следующим образом: $E = \{e_1, \ldots, e_n\}$, где n – количество входных переменных. Определим также множество *выходных переменных*, наполненное новосоздан-

ными дополнительными переменными, $S = \{s_1, \ldots, s_n\}$, а также множество *связующих переменных* L, изначально пустое. Необходимо построить бинарное дерево, каждая вершина которого будет отмечена парой из некоторого множества и числа. Отметим, что число в паре является *мощностью* множества из этой пары. Начнем с одной вершины, помеченной парой (S, n). Итеративно будем выполнять следующие действия. К каждому листу дерева, отмеченному числом $m > 1$, необходимо добавить двух потомков, отмеченных числами $\lfloor m/2 \rfloor$ и $(m - \lfloor m/2 \rfloor)$. Если $\lfloor m/2 \rfloor > 1$, то множество, которым отмечается левый потомок состоит из $\lfloor m/2 \rfloor$ *связующих* переменных – $\left\{\nu^l_1, \ldots, \nu^l_{\lfloor m/2 \rfloor}\right\}$, иначе – из одной очередной *входной* переменной – $\{e_i\}$. Итеративный процесс завершается, когда все листья отмечены только входными переменными. На рисунке 9 показан пример дерева для $n = 5$.

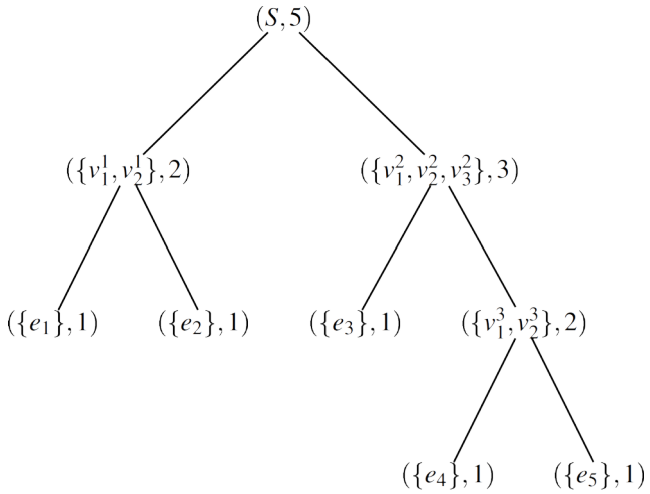

Рисунок 9 – Бинарное дерево для $n = 5$ со связующими переменными $L = \left\{\nu^1_1, \nu^1_2, \nu^2_1, \nu^2_2, \nu^2_3, \nu^3_1, \nu^3_2\right\}$

Следующий шаг – построение так называемого *сумматора* (ориг. – *totalizer*) – КНФ-формулы $\Phi(E)$. Рассмотрим тройку вершин – вершину дерева *r* и ее потомков *a* и *b*. Обозначим число, которым отмечена вершина *r* – m, а множество – $R = \{r_1, \ldots, r_m\}$. Аналогично, для вершины *a* – m_1 и $A = \{a_1, \ldots, a_{m_1}\}$, для вершины *b* – m_2 и $B = \{b_1, \ldots, b_{m_2}\}$. В формулу $\Phi(E)$ добавляются следующие дизъюнкты:

$$\bigwedge_{\substack{0 \le \alpha \le m_1 \\ 0 \le \beta \le m_2 \\ 0 \le \sigma \le m \\ \alpha + \beta = \sigma}} (C_1(\alpha, \beta, \sigma) \wedge C_2(\alpha, \beta, \sigma)) \tag{1}$$

При этом используется следующая нотация:

$$a_0 = b_0 = r_0 = 1, \quad a_{m_1+1} = b_{m_2+1} = r_{m+1} = 0$$
$$C_1(\alpha, \beta, \sigma) = \neg a_\alpha \vee \neg b_\beta \vee r_\sigma \tag{2}$$
$$C_2(\alpha, \beta, \sigma) = a_{\alpha+1} \vee b_{\beta+1} \vee \neg r_{\sigma+1}$$

Отметим, что $C_1(\alpha, \beta, \sigma)$ отвечает за соотношение $\sigma \geq \alpha + \beta$, а $C_2(\alpha, \beta, \sigma)$ – за соотношение $\sigma \leq \alpha + \beta$. Полученная формула упрощается путем удаления дизъюнктов, содержащих константу 1, а также путем сокращения дизъюнктов, содержащих константу 0. Описанный процесс повторяется для всех троек вершин в дереве.

В дополнение к полученному сумматору $\Phi(E)$, необходимо построить *компаратор* – множество единичных дизъюнктов, определяющих интервал, в котором находится унарно закодированное число – мощность заданного множества входных переменных. Ограничение на мощность вида $\mu \leq N(E_1) \leq \rho$ кодируется следущим образом:

$$\bigwedge_{1 \leq i \leq \mu} (s_i) \bigwedge_{\rho+1 \leq j \leq n} \neg s_j \tag{3}$$

Возвращаясь к нашей задаче, множество входных переменных $E = \left\{\neg \mathrm{nodetype}_{c,k,p,4} \mid c \in [1..C], k \in [1..K], p \in [1..P]\right\}$. Применяя описанный выше алгоритм построения *сумматора*, построим КНФ-формулу $\Phi(E)$, попутно создав дополнительные *выходные* и *связующие* переменные. Для ограничения вида $N(E) \leq N$, необходим *компаратор* следующего вида:

$$\bigwedge_{N+1 \leq j \leq n} \neg s_j \tag{4}$$

Заметим, что при итеративном переборе числа N (см. раздел 2.3.2) сверху вниз, достаточно только добавлять необходимые единичные дизъюнкты компаратора (4) в уже построенную КНФ-формулу. Таким образом, мы уже будем не решать задачу, а «дорешивать» – с небольшим количеством новых ограничений. Однако, при таком подходе мы должны выбрать соответствующее программное средство – *решатель*, поддерживающий *инкрементальное решение*. В данной работе был использован SAT-решатель *lingeling* [21], дополненный возможностью непрерывного считывания команд и дизъюнктов из стандартного потока ввода.

Остановимся на подробном рассмотрении техники *нарушения симметрии (symmetry breaking)*, успешно использованной в [27]. Предлагается добавить к уже описанным ограничениям *предикаты нарушения симметрии* [28] которые будут помогать SAT-решателю тем, что большая часть «симметричных» решений будут запрещены (в идеале, останется ровно одно допустимое решение). В контексте рассматриваемой задачи симметричность решений заключается в том, что существует $C!$ способов (число перестановок) раскрасить дерево сценариев в C цветов. Таким образом, задача может быть решена множеством способов, но в случае, когда искомой раскраски вообще не существует, решателю придется так или иначе проверить все варианты решения. Заметим, что добавление предикатов нарушения симметрии никоим образом не влияет на получаемое решение – лишь значительно сокращается дерево поиска решений, что существенно уменьшает время решения «UNSAT» случаев.

Основная идея подхода заключается в задании порядка следования номеров состояний синтезируемого автомата в соответствие с некоторым обходом. Широко распространен *обход в ширину (breadth-first search; BFS)*. Для BFS-предикатов нарушения симметрии будем использовать следующие переменные:

- $\mathrm{bfs\_transition}_{c_1 \in [1..C], c_2 \in [1..C]}$ – существует ли переход между c_1 и c_2;
- $\mathrm{bfs\_parent}_{c_1 \in [1..C], c_2 \in [1..C]}$ – является ли вершина c_2 родителем вершины c_1 в дереве обхода автомата в ширину.

Предикаты нарушения симметрии выглядят следующим образом:

- $\mathcal{F}_t = \bigwedge_{1 \le i < j \le C} (\mathrm{bfs\_transition} \leftrightarrow \bigvee_{k \in [1..K]} \mathrm{transition}_{i,k,j})$
- $\mathcal{F}_p = \bigwedge_{1 \le i < j \le C} (\mathrm{bfs\_parent} \leftrightarrow \mathrm{bfs\_transition}_{i,j} \wedge \bigwedge_{1 \le k < i} \neg \mathrm{bfs\_transition}_{k,j})$
- $\mathcal{F}_{\mathtt{ALO}(p)} = \bigwedge_{1 < j \le C} \mathtt{ALO}_{1 \le i < j}(\mathrm{bfs\_parent}_{j,i})$
- $\mathcal{F}_{\mathtt{BFS}(p)} = \bigwedge_{1 \le k < i < j < C} (\mathrm{bfs\_parent}_{j,i} \rightarrow \neg \mathrm{bfs\_parent}_{j+1,k})$

Полный список переменных и ограничений доступен в [29].

2.3.2 Алгоритм перебора параметров

Имея возможность построения конечно-автоматной модели – с заданным числом состояний C, максимальным числом K переходов из каждого состояния, максимальным размером P каждого дерева разбора и максимальным числом N

типизированных вершин всех деревьев разбора – с помощью алгоритма $\texttt{buildModel}(\mathcal{T}, C, K, P, N)$ – рассмотрим задачу построения минимальной модели. По аналогии с разделами 2.1.2 и 2.2.2, допустим, что *минимальными* параметрами являются $C_{\min}$, $K_{\min}$, $P_{\min}$ и $N_{\min}$, тогда известно, что вызовы с меньшими значениями параметров обнаружат отсутствие решения: $\forall C < C_{\min}\ \forall K < K_{\min}\ \forall P < P_{\min}\ \forall N < N_{\min} : \texttt{buildModel}(\mathcal{T}, C, K, P, N) = \texttt{null}$. Заметим, что решение будет также отсутствовать при произвольных параметрах K, P и N, в случае, когда $C < C_{\min}$; при произвольных параметрах P и N – когда $K < K_{\min}$; при произвольном параметре N – когда $P < P_{\min}$. Перебор параметров C, K и P организован в виде линейного поиска снизу вверху – от 1 и выше. Перебор параметра N огранизован сверху вниз следующим образом. Верхняя граница $N_{\max} = C \cdot K \cdot P$. Очередное значение параметра N на единицу меньше суммарного числа типизированных вершин во всех деревьях разбора последнего найденного решения. Перебор заканчивается при отсутствии решения с $N = N_{\text{last}} - 1$, тогда $N_{\min} = N_{\text{last}}$.

3 Экспериментальное исследование

В данной главе описывается эксперимент по генерации конечно-автоматной модели логического контроллера, управляющего Pick-and-Place манипулятором. В разделе 3.1 описывается симуляционная модель объекта управления – манипулятора. В разделе 3.2 описывается созданный вручную логический контроллер. В разделе 3.3 проводится генерация конечно-автоматно модели контроллера по примерам поведения.

3.1 Модель объекта управления Pick-and-Place манипулятора

Симуляционная модель Pick-and-Place манипулятора (рисунок 10) выполнена в nxtStudio [30] в виде совокупности функциональных блоков (рисунок 11), согласно стандарту IEC 61499, отвечает за моделирование непрерывных процессов, происходящих при движении частей манипулятора, и обладает чисто логическим интерфейсом, что позволяет управлять таким объектом управления с помощью логического контроллера.

Манипулятор закреплен на основе, состоит из двух горизонтальных и одного вертикального цилиндров и оснащен вакуумной присоской, предназначенной для перемещения деталей, схематически изображенных в правой части изображения. Во время симуляции, рабочие предметы помещаются во входные лотки (1–3), и задачей манипулятора, а точнее, управляющего им логического контроллера, является перемещение деталей в выходное отверстие.

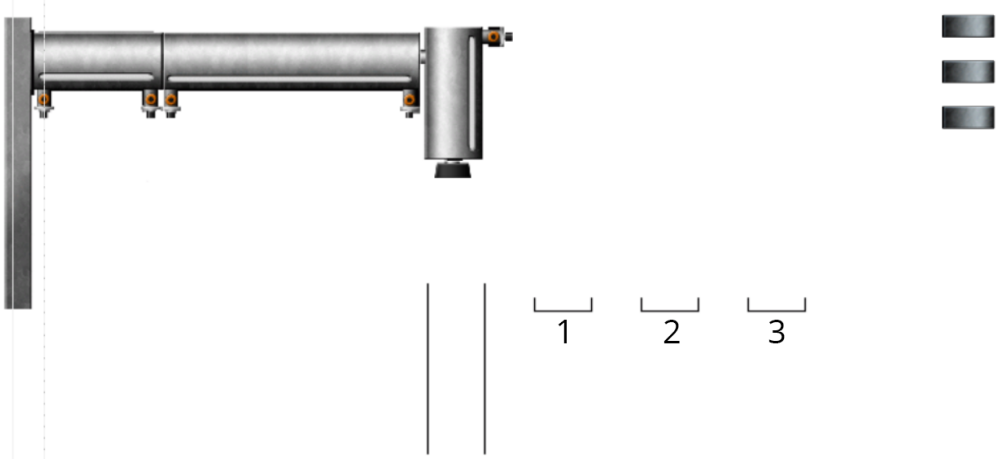

Рисунок 10 – Pick-and-Place манипулятор в nxtStudio

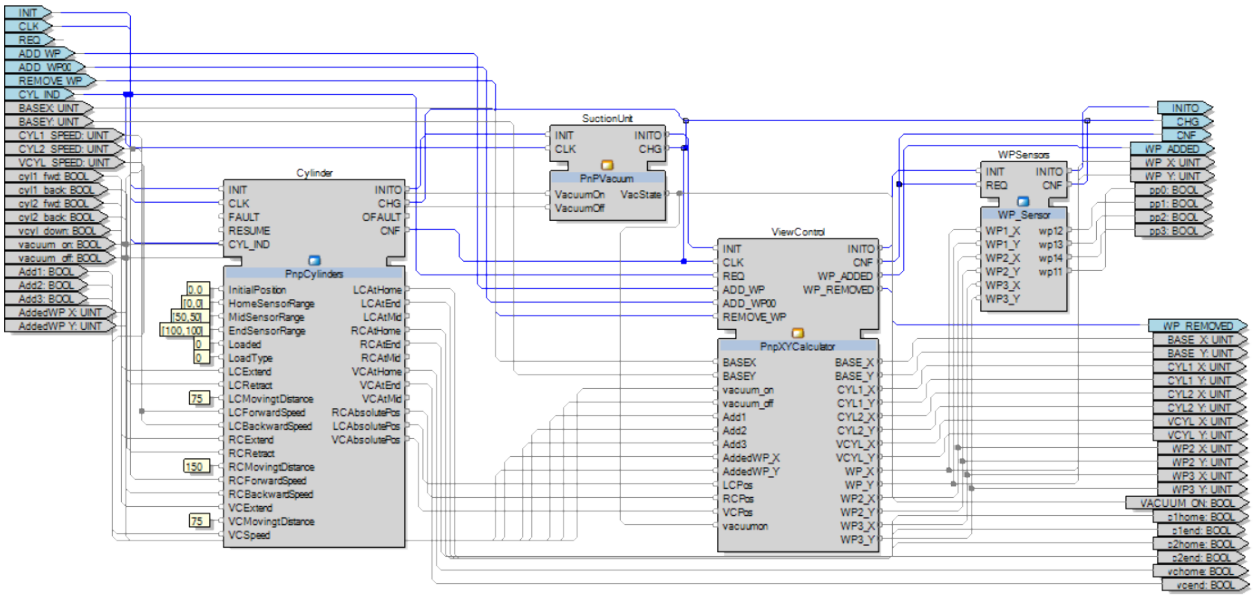

Рисунок 11 – Симуляционная модель Pick-and-Place манипулятора, выполненная в виде совокупности функциональных блоков

3.2 Модель контроллера

Логический контроллер, управляющий Pick-and-Place манипулятором, представлен в виде функционального блока (рисунок 12), согласно стандарту IEC 61499. Диаграмма управления выполнением программы *(ECC)* приведена на рисунке 13. Интерфейс ФБ устроен следующим образом:

а) Входные события E^I:

- `INIT` – событие инициализации;
- `REQ` – событие-запрос.

б) Выходные события E^O:

- `CNF` – событие-ответ.

в) Входные переменные X:

- `c1Home/c1End` – цилиндр 1 находится в крайнем левом/правом положении;
- `c2Home/c2End` – цилиндр 2 находится в крайнем левом/правом положении;
- `c3Home/c3End` – цилиндр 3 находится в крайнем верхнем/нижнем положении;
- `pp1/pp2/pp3` – деталь находится во входном лотке 1/2/3;

— `vac` – включена ли вакуумная присоска.

г) Выходные переменные Z:

— `c1Extend/c1Retract` – удлинить/втянуть цилиндр 1;

— `c2Extend/c2Retract` – удлинить/втянуть цилиндр 2;

— `vcExtend` – удлинить цилиндр 3;

— `vacuum_on` – включить вакуумную присоску;

— `vacuum_off` – выключить вакуумную присоску.

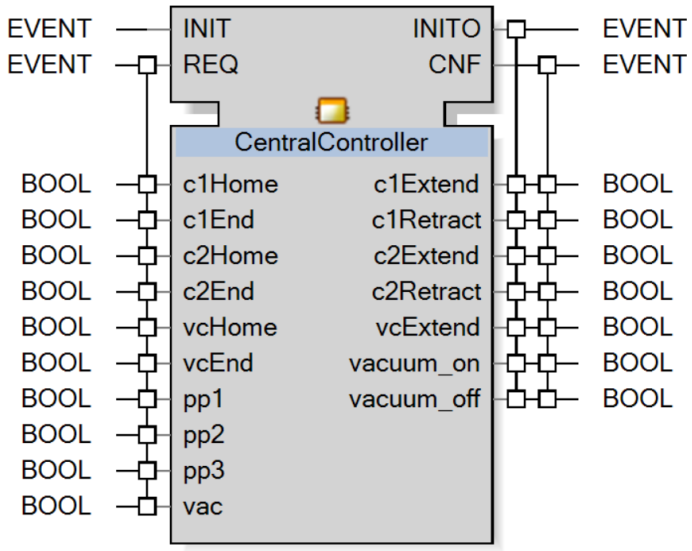

Рисунок 12 – Логический контроллер, управляющий Pick-and-Place манипулятора, выполненный в виде функционального блока

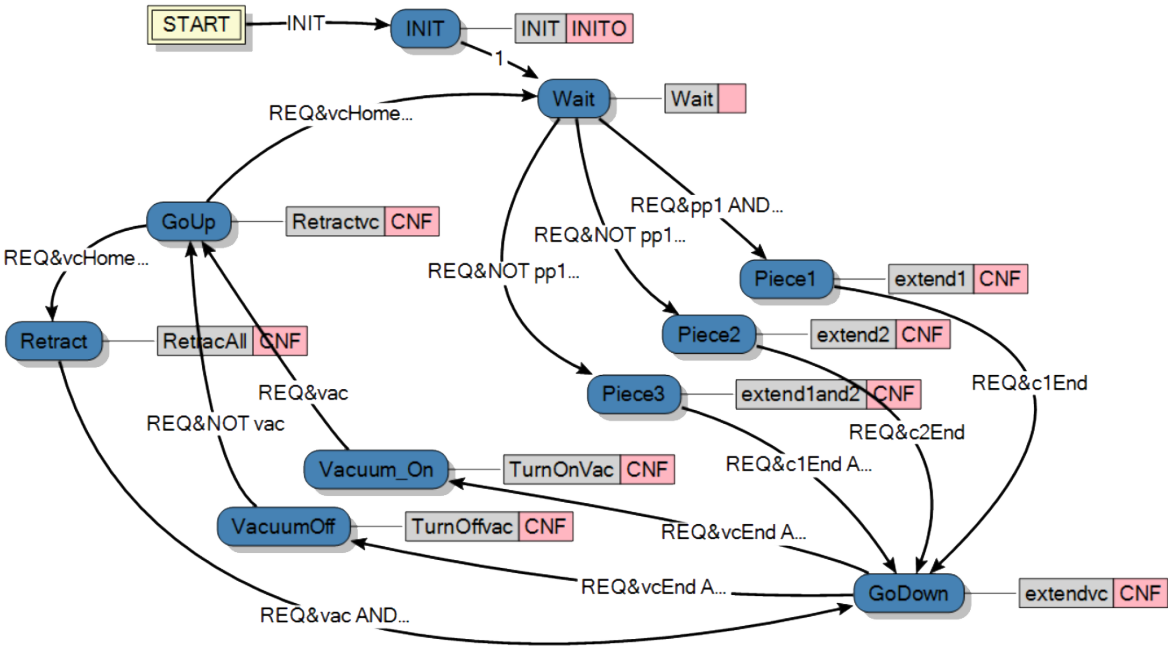

Рисунок 13 – Диаграмма управления выполнением программы логического контроллера, управляющего Pick-and-Place манипулятором

3.3 Генерация модели контроллера по примерам поведения

В данном разделе рассматривается процесс генерации конечно-автоматной модели контроллера, управляющего Pick-and-Place манипулятором, по примерам поведения. Заметим, что подготовительный этап сбора сценариев выполнения, необходимых для построения дерева сценариев, подробно не рассматривается, так как он ничем не отличается от такового в [25].

Сначала была произведена оценка параметров C и K искомого автомата с помощью алгоритма SATGEN. Затем была получена верхняя граница параметра P с помощью алгоритма TRANSITION-WISE. Наконец, был применен алгоритм FBSAT, в результате чего была получена минимальная конечно-автоматная модель. Результаты для различных примеров поведения приведены в таблице 1, где «сценарии» – наименования примеров поведений, $|\mathcal{T}|$ – размер префиксного дерева сценариев, t – время работы, C – число состояний, K – максимальное число переходов из каждого состояния, N – суммарный размер охранных условий.

Таблица 1 – Сравнительная таблица реализованных методов

Сценарии	$\lvert\mathcal{T}\rvert$	SATGEN			TRANSITION-WISE		FBSAT	
		t, с	C	K	t, с	N	t, с	N
tests-1	24	1,9	6	2	8,1	17	5,1	14
tests-10	234	12,4	8	4	22,8	41	170	25
tests-39	960	44,9	8	4	34,9	42	225	25
tests-10-60	2939	83,9	8	4	191,3	64	≈12000	39

Полученная в результате работы алгоритма FBSAT конечно-автоматная модель была проверена на работоспособность в nxtStudio следующим образом. Оригинальная модель логического контроллера была заменена сгенерированной, затем была запущена симуляция, в ходе которой система продолжала выполнять поставленную задачу – перемещение деталей, периодически создаваемых в случайных входных лотках, в выходное отверстие – на протяжении всей симуляции. На основании этого был сделан вывод о том, что синтез модели контроллера по примерам поведения был произведен успешно.

Заключение

В данной работе были предложены и реализованы методы генерации конечно-автоматных моделей контроллеров по примерам поведения. При этом было использовано программирование в ограничениях – задачи решались путем сведения к задачам SAT и CSP. Существующий двухэтапный метод fbCSP [25] был существенно улучшен – для генерации базовой модели было реализовано сведение к SAT, а вместо жадной минимизации был реализован точный метод на основе сведения к CSP. Также, в дополнение к двухэтапного методу был предложен и реализован совмещенный одноэтапный метод генерации минимальных конечно-автоматных моделей fbSAT. Отметим, что одноэтапный метод является самодостаточным и способен синтезировать конечно-автоматную модель «с нуля» – по сценариям исполнения, однако, было показано, что двухэтапный метод может быть успешно использован для быстрой оценки параметров искомого автомата перед применением одноэтапного метода. Реализация совмещенного метода fbSAT в виде программного средства доступна по [29]. Работоспособность методов была подтверждена верификацией, а также экспериментальным исследованием по генерации конечно-автоматной модели логического контроллера, управляющего Pick-and-Place манипулятором.

Список использованных источников

1 Автоматизация процессов [Электронный ресурс]. URL: http://opiobjektid.tptlive.ee/Automatiseerimine (дата обращения: 27.04.2018).

2 Гаврилов М.А., Девятков В.В., Пупырев Е.И. Логическое проектирование дискретных автоматов. М.: Наука, 1977. 363 с.

3 Автоматное управление асинхронными процессами в ЭВМ и дискретных системах. Под ред. В.И. Варшавского. М.: Наука, 1986. 400 с.

4 Шалыто А.А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. Спб.: 1998. 627 с.

5 Шалыто А.А. Логическое управление. Методы аппаратной и программной реализации.Спб.: Наука, 2000. 780 с.

6 Поликарпова Н.И., Шалыто А.А. Автоматное программирование. СПб.: Питер, 2011. 176 с.

7 Vyatkin V. IEC 61499 as Enabler of Distributed and Intelligent Automation: State-of-the-Art Review // IEEE Transactions on Industrial Informatics, vol. 7, no. 4, pp. 768–781, 2011.

8 Chivilikhin D., Shalyto A., Vyatkin V. Inferring Automata Logic From Manual Control Scenarios: Implementation in Function Blocks // In Proceedings of the 13th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA'15), 2015, pp. 307–312

9 Cheng K-T., Krishnakumar A.S. Automatic Functional Test Generation Using The Extended Finite State Machine Model // International Design Automation Conference (DAC). ACM. pp. 86–91, 1993.

10 Царев Ф.Н. Методы построения конечных автоматов на основе эволюционных алгоритмов. – 2012. – Диссертация на соискание ученой степени кандидата технических наук. НИУ ИТМО.

11 Ulyantsev V., Buzhinsky I., Shalyto A. Exact finite-state machine identification from scenarios and temporal properties // Int. Journ. Softw. Tools Techn. Transf., pp. 1–21, 2016.

12 Back T., Fogel D.B., Michalewicz Z. Handbook of Evolutionary Computation // Bristol, UK: IOP Publishing Ltd., 1997. 1130 p.

13 Tsarev F., Egorov K. Finite state machine induction using genetic algorithm based on testing and model checking // Conf. Comp. Genetic Evol. Comput. ACM, 2011, pp. 759–762.

14 Chivilikhin D., Ulyantsev V. MuACOsm: a new mutation-based ant colony optimization algorithm for learning finite-state machines // Conf. Genetic Evol. Comput., 2013, pp. 511–518.

15 Lang K., Pearlmutter B., Price R. Results of the Abbadingo One DFA Learning Competition and a New Evidence-Driven State Merging Algorithm // Grammatical Inference. Lecture Note in Computer Science, vol. 1433, pp. 1–12. Springer, 1998.

16 Biere A., Heule V., van Maaren H., Walsh T. Handbook of Satisfiability. IOS Press, 2009. 980 p.

17 Cook S.A. The Complexity of Theorem-Proving Procedures // Proceedings of the Third Annual ACM Symposium on Theory of Computing. ACM, 1971. pp. 151–158.

18 Bordeaux L., Hamadi Y., L. Zhang. Propositional satisfiability and constraint programming: A comparative survey // ACM Comput. Surv., vol. 38, no. 4, 2006.

19 Eén N., Sörensson N. An Extensible SAT-solver // In Theory and Applications of Satisfiability Testing. SAT 2003. Lecture Notes in Computer Science, vol 2919. Springer, Berlin, 2004.

20 Soos M., Nohl K., Castelluccia C. Extending SAT Solvers to Cryptographic Problems // In Theory and Applications of Satisfiability Testing, SAT 2009. Lecture Notes in Computer Science, vol 5584. Springer, Berlin, 2009.

21 Biere A. Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013 // In Proceedings of SAT Competition 2013, vol. B-2013-1 of Department of Computer Science Series of Publications B, pages 51–52, University of Helsinki, 2013.

22 Nethercote N. et al. MiniZinc: Towards a standard CP modelling language // In Bessiere C., editor, Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, vol. 4741 of LNCS, pp 529–543. Springer, 2007.

23 N. van Omme, Perron L., Furnon V., or-tools user's manual, Google, 2014.

24 Jussien N., Rochart G., Lorca X. Choco: an open source java constraint programming library // CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08), 2008. pp 1–10.

25 Chivilikhin D., Ulyantsev V., Shalyto A. and Vyatkin V. CSP-based inference of function block finite-state models from execution traces // In Proceedings of the 15th IEEE International Conference on Industrial Informatics, 2017, pp. 714–719.

26 Bailleux O., Boufkhad Y. Efficient CNF Encoding of Boolean Cardinality Constraints // In Rossi F., editor, Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, vol 2833. Springer, Berlin, Heidelberg, 2003.

27 Heule M., Verwer S. Exact DFA Identification Using SAT Solvers // Int. Colloquium Conf. on Grammatical Inference, 2010, pp. 66–79.

28 Ulyantsev V., Zakirzyanov I., Shalyto A. BFS-Based Symmetry Breaking Predicates for DFA Identification // Lecture Notes in Computer Science, 2015, vol. 8977, pp. 611–622.

29 ctlab/fbSAT [Электронный ресурс]. URL: https://github.com/ctlab/fbSAT (дата обращения: 27.04.2018).

30 nxtControl - nxtSTUDIO [Электронный ресурс]. URL: http://www.nxtcontrol.com/en/engineering (дата обращения: 27.04.2018).

Приложение А

полный список ограничений для алгоритма SATGEN

	Constraint	Domain
1.1	$c_{1,1}$	
1.2	$\neg d^0_{1,z} \wedge \neg d^1_{1,z}$	$z \in Z$
1.3	$\neg o_{1,o}$	$o \in E^O$
2.1	$c_{v,1} \vee \ldots \vee c_{v,C}$	$v \in V$
2.2	$c_{v,i} \rightarrow \neg c_{i,j}$	$1 \le i < j \le C$
3.1	$c_{u,i} \wedge c_{v,j} \rightarrow t_{i,e^{\text{in}}_{uv},g_{uv},j}$	$\begin{cases} u, v \in V \\ uv \in E \\ 1 \le i, j \le C \end{cases}$
3.2	$t_{i,e,g,j} \rightarrow \neg t_{i,e,g,k}$	$\begin{cases} 1 \le i \le j < k \le C \\ e \in E^I \\ g \in U \end{cases}$
4.1	$c_{v,n} \rightarrow \neg o_{n,e^{\text{out}}_v}$	$v \in V, 1 \le n \le C$
4.2	$o_{n,1} \vee \ldots \vee o_{n,\lvert E^O \rvert}$	$1 \le n \le C$
4.3	$o_{n,j_1} \rightarrow \neg o_{n,j_2}$	$\begin{cases} 1 \le n \le C \\ 1 \le j_1 < j_2 \le \lvert E^O \rvert \end{cases}$
5	$c_{v,n} \rightarrow \begin{cases} \neg d^0_{n,z} & \text{if } \neg z_{u,z} \wedge \neg z_{v,z} \\ d^0_{n,z} & \text{if } \neg z_{u,z} \wedge z_{v,z} \\ \neg d^1_{n,z} & \text{if } z_{u,z} \wedge \neg z_{v,z} \\ d^1_{n,z} & \text{if } z_{u,z} \wedge z_{v,z} \end{cases}$	$\begin{cases} u, v \in V \\ uv \in E \\ z \in Z \\ 1 \le n \le C \end{cases}$
6.1	$\zeta_{i,j} \leftrightarrow \bigwedge_{e,l} t_{i,e,l,j}$	$1 \le i, j \le C$
6.2	$\bigvee_n \gamma_{n,0,0}$	$1 \le n \le C$
6.3	$\gamma_{i,j-1,k} \wedge \zeta_{i,j} \rightarrow \gamma_{i,j,k+1}$	$\begin{cases} 1 \le i, j \le C \\ j > 1 \\ 0 \le k \le C \end{cases}$
6.4	$\gamma_{i,j-1,k} \wedge \neg\zeta_{i,j} \rightarrow \gamma_{i,j,k}$	$\begin{cases} 1 \le i, j \le C \\ j > 1 \\ 0 \le k \le C \end{cases}$
6.5	$\neg\gamma_{n,C,k}$	$\begin{cases} 1 \le n \le C \\ K+1 \le k \le C \end{cases}$

Приложение Б

Minizinc-модель, описывающая CSP-ограничения для алгоритма TRANSITION-WISE

```
% constants
int: X; % number of input variables
int: D; % number of trace elements
int: P; % number of nodes
int: K; % number of transitions
int: k; % index of current transition

constraint assert(1 <= k /\ k <= K, "k should be in [1..K]");

% given data
array [1..D, 1..X] of bool: inputs; % input variables values before the i-th trace element is processed
array [1..D] of bool: fired;        % index of the transition the was fired when the i-th trace element was processed
array [1..D] of 0..K: tran_id;      % index of fired transition (0 - for non-fired)

% variables
array [1..P] of var 0..3: nodetype;
array [1..P] of var 0..X: terminal; % index of the input variable
array [1..P] of var 0..P: child;    % left child
array [1..P] of var 0..P: parent;   % parent index, or 0 if no parent
array [1..P, 1..D] of var bool: value;
array [1..P, 1..D] of var bool: child_value; % Left child value
array [1..P, 1..D] of var bool: child_value_second; % Right child value

var int: E; % number of edges
var int: V; % number of nodes

% constraints
% +++ 1.3. "AND"/"OR" typed nodes can't have number P or P-1
constraint nodetype[P] != 1 /\ nodetype[P] != 2;
constraint if P > 1
           then nodetype[P-1] != 1 /\ nodetype[P-1] != 2
           else true endif;
% +++ 1.4. "NOT" typed nodes can't have number P
constraint nodetype[P] != 3;

% 2.1. Only terminals have associated terminal variables
constraint forall (p in 1..P) (
    nodetype[p] = 0 <-> terminal[p] != 0
);

% 3.1. Root has no parent
constraint parent[1] = 0;
% 3.3. Non-root nodes must have a parent with lesser number
constraint forall (p in 2..P) (
    parent[p] != 0
    /\ parent[p] < p
);
```

```
48
49      % 4.1. Only terminals have no children
50      constraint forall (p in 1..P) (
51          nodetype[p] = 0 <->
52              child[p] = 0
53      );
54      % 4.2 "AND"/"OR" non-terminals have exactly 2 children, while left
        child has number from range [p+1 .. P-1]
55      constraint forall (p in 1..P-2) (
56          nodetype[p] = 1 \/ nodetype[p] = 2 ->
57              child[p] > p
58              /\ child[p] < P
59              /\ sum (c in p+1..P) (bool2int(parent[c] = p)) = 2
60      );
61      % 4.3. "NOT" non-terminals have exactly 1 child with greater number
62      constraint forall (p in 1..P-1) (
63          nodetype[p] = 3 ->
64              child[p] > p
65              /\ sum (c in p+1..P) (bool2int(parent[c] = p)) = 1
66      );
67
68      % 5.1. Parent of left child of node p  *is*  p
69      constraint forall (p in 1..P, c in p+1..P) (
70          child[p] = c -> parent[c] = p
71      );
72      % 5.2. Parent of right child of node p (AND/OR type)  *is*  p
73      constraint forall (p in 1..P, c in p+1..P-1) (
74          (nodetype[p] = 1 \/ nodetype[p] = 2)
75          /\ child[p] = c ->
76              parent[c+1] = p
77      );
78      % 5.3. If vertex p is called a "parent", it is to have a child
79      constraint forall (p in 1..P, c in p+1..P) (
80          parent[c] = p -> child[p] != 0
81      );
82
83      % 6.1. Terminal value
84      constraint forall (p in 1..P) (
85          nodetype[p] = 0 ->
86              forall (d in 1..D)
87                  (value[p, d] <-> inputs[d, terminal[p]])
88      );
89      % 6.2. AND value
90      constraint forall (p in 1..P) (
91          nodetype[p] = 1 ->
92              forall (d in 1..D)
93                  (value[p, d] <-> child_value[p, d] /\
        child_value_second[p, d])
94      );
95      % 6.3. OR value
96      constraint forall (p in 1..P) (
97          nodetype[p] = 2 ->
98              forall (d in 1..D)
99                  (value[p, d] <-> child_value[p, d] \/
        child_value_second[p, d])
100     );
```

```
% 6.4. NOT value
constraint forall (p in 1..P) (
    nodetype[p] = 3 ->
        forall (d in 1..D)
            (value[p, d] <-> not child_value[p, d])
);
% 6.5. not fired
constraint forall (d in 1..D) (
    tran_id[d] = 0 ->
        not value[1, d]
);
% 6.6. fired(k) -> not_fired(<k)
constraint forall (d in 1..D) (
    k < tran_id[d] ->
        not value[1, d]
);
constraint forall (d in 1..D) (
    k = tran_id[d] ->
        value[1, d]
);

% 7.1. Left child value
constraint forall (p in 1..P, c in p+1..P) (
    child[p] = c ->
        forall (d in 1..D)
            (child_value[p, d] <-> value[c, d])
);
% 7.2. Right child value
constraint forall (p in 1..P, c in p+1..P-1) (
    (nodetype[p] = 1 \/ nodetype[p] = 2)
    /\ child[p] = c ->
        forall (d in 1..D)
            (child_value_second[p, d] <-> value[c + 1, d])
);

% 8.
constraint E = sum (p in 1..P) (bool2int(parent[p] != 0));
constraint V = P;
constraint E = V - 1;

solve satisfy;

output ["K=", show(K), ", P=", show(P), ", D=", show(D), ", E=",
show(E), "\n"] ++
       ["# nodetype = ", show(nodetype), "\n",
        "# terminal = ", show(terminal), "\n",
        "# parent = ", show(parent), "\n",
        "# child = ", show(child_orig), "\n",
        "# left-child = ", show(child), "\n"]
```