Министерство образования и науки Российской Федерации
Санкт-Петербургский политехнический университет Петра Великого
Высшая школа программной инженерии

Работа допущена к защите
Директор ВШ ПИ
_____П.Д. Дробинцев
"____"_____ 2018г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА МАГИСТРА

Разработка и реализация алгоритма синтеза систем переходов для $GR(1)$-формул при помощи BDD

По направлению *02.04.02 «Фундаментальная информатика и информационные технологии»*
по образовательной программе
02.04.02_02 «Проектирование сложных информационных систем»

Выполнил
студент гр. 63507 Т. Д. Архипов
Руководитель
к.т.н., доц. И.В. Шошмина

Санкт-Петербург
2018

САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО

Институт компьютерных наук и технологий

Утверждаю
Директор ВШ ПИ
_____П.Д. Дробинцев
"___"_____ 2018г.

ЗАДАНИЕ

по выполнению выпускной квалификационной работы
студенту Т. Д. Архипову гр. 63507

1. Тема: *Разработка и реализация алгоритма синтеза систем переходов для $GR(1)$-формул при помощи BDD*
2. Срок сдачи работы.
3. Исходные данные к проекту (работе).
4. Содержание расчетно-пояснительной записки (перечень подлежащих разработке вопросов).
5. Перечень графического материала с точным указанием обязательных чертежей.
6. Консультанты по проекту (с указанием относящегося к ним разделов проекта, работы).

Дата выдачи задания: _____ г.
Руководитель ВКР: _____к.т.н., доц.И.В. Шошмина
Задание принял к исполнению _____ г.
Студент _____Т. Д. Архипов

Реферат

На 46 с., , табл. 3

В данном исследовании рассматривается задача синтеза контроллера для реагирующей системы по заданным требованиям. Для уменьшения алгоритмической сложности синтеза требования представляются с помощью подкласса LTL-формул - GR(1)-формул.

При заданной реагирующей системе для предъявляемых к ней требований в виде GR(1)-формулы производится проверка, являются ли они реализуемыми, и, если так, производится синтез совместного контроллера, заданную формулу реализующего. Синтез системы переходов реализован символьным методом, при этом символьная запись хранится в памяти в виде BDD.

Для уменьшения количества состояний синтезированной системы к ней применяется алгоритм минимизации. После минимизации синтезированная модель системы транслируется в формат DOT для ее последующей визуализации.

Алгоритм синтеза систем переходов по GR(1)-формулам, на который опирается основная часть реализации, описан в статье [1. PPS06]. (N. Piterman, A. Pnueli, Y. Sa'ar "Synthesis of reactive(1) designs", VMCAI 3855, 364-380, 2006)

Abstract

46 pages , 3 tables
Brief description of the work, main result, keywords

Оглавление

Список обозначений

BDD	Binary Decision Diagrams: бинарные решающие диаграммы
LTL	Linear Temporal Logic: линейная темпоральная логика

Введение

В данном исследовании рассматривается задача синтеза контроллера для реагирующей системы по заданным требованиям. Для уменьшения алгоритмической сложности синтеза требования представляются с помощью подкласса LTL-формул - GR(1)-формул.

При заданной реагирующей системе для предъявляемых к ней требований в виде GR(1)-формулы производится проверка, являются ли они реализуемыми, и, если так, производится синтез совместного контроллера, заданную формулу реализующего. Синтез системы переходов реализован символьным методом, при этом символьная запись хранится в памяти в виде BDD.

Для уменьшения количества состояний синтезированной системы к ней применяется алгоритм минимизации. После минимизации синтезированная модель системы транслируется в формат DOT для ее последующей визуализации.

Алгоритм синтеза систем переходов по GR(1)-формулам, на который опирается основная часть реализации, описан в статье [2]

Глава 1

Постановка задачи синтеза систем переходов из GR–формул

1.1 LTL

Синтаксис при помощи формул. Семантика. Представление в виде систем переходов (здесь имеет смысл писать об недетерминированных автоматах Бюхи только).

1.2 Задача выполнимости LTL–формул. Задача выполнимости LTL–формул на конкретной системе переходов.

Метод проверки модели как способ решения.

1.3 LTL–формулы с переменными разного типа

Среда, система. Задача реализуемости таких формул – задача выполнимости в логике высокого порядка. Сложность решения такой задачи в общем случае.

1.4 GR–формулы синтаксис и семантика

?GR–формулы. Просто дискретные системы (детерминированные автоматы Бюхи)

Задача реализуемости GR–формул. Задача синтеза контроллера. Сложность задачи реализуемости

1.5 Постановка задачи символьного представления систем переходов

Символьное представление систем переходов +BDD

Цель работы —

Задачи работы —

Глава 2

Алгоритм синтеза GR–формул

2.1 Задача синтеза как задача построения отображения

Справедливая симуляция. Симуляционная игра.

2.2 Решение задачи синтеза с помощью мю–уравнения

Мю–исчисление. Мю–уравнение. Теорема о реализуемости. Алгоритм. Синтезируемая система переходов.

2.3 Символьная реализация алгоритма

Прямой, обратный образы, сох

Глава 3

Реализация алгоритма синтеза GR–формул

3.1 Архитектура программного средства

Работа синтеза была реализована на языке $c++$, для сборки проекта использовался компилятор $MSVC++$ версии 11.0 ($_{M}SC_{V}ER == 1700$).

3.1.1 BuDDy

В процессе синтеза предполагается работа с символьным описанием состояний, требований и систем переходов, которое представляется программно в виде BDD. Для работы с BDD используется пакет $BuDDy$ [1]. Ниже описаны ключевые моменты работы с библиотекой $BuDDY$.

Для подключения и использования библиотеки необходимо скачать пакет $BuDDy$: $https://sourceforge.net/projects/buddy/$ и произвести его сборку на своей рабочей машине. Далее необходимо включить библиотеку и заголовочный файл $bdd.lib, bdd.h$ в исходный файл, работающий с BDD, добавлением следующего кода:

```
#pragma comment(lib, "bdd.lib") // BuDDy library
#include "bdd.h"
```

Сперва необходимо инициализировать пакет BDD с помощью команды:

```
bdd_init(nodenum,cachesize);
//Для больших примеров --- nodenum~10000000,cachesize~1000000
% bdd_setcacheratio(10);
```

где *nodenum* — начальное количество выделенных вершин BDD, *cachesize* — размер кэша, используемого при выполнении операций над BDD.

После инициализации необходимо установить количество переменных BDD, с помощью которых будет кодироваться символьная запись.

```
bdd_setvarnum(unsigned N);
```

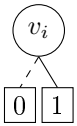
Однако, в нашем случае сразу вычислять количество переменных BDD неудобно, поэтому в программе используется постепенное увеличение количества переменных по мере их прочтения с помощью функции:

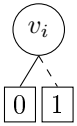
```
bdd_extvarnum(unsigned N);
```

В программе используются переменные двух типов — двоичного и перечислимого. Для кодирования каждой символьной переменной необходимо выделять двукратное количество переменных BDD, так как в каждом требовании к поведению может быть два вида каждой переменной — var и $next(var)$ (то же самое, что var'). Поэтому для каждой переменной верно следующее:

- Для кодирования булевых переменных выделяется 2 переменные BDD
- Для кодирования переменных перечислимого типа выделяется $\lceil \log_2(N) \rceil$, где N — количество возможных значений переменной

Любая булева функция может быть представлена с помощью BDD. Для этого в *BuDDy* используются следующие атомарные функции:

- bdd_ithvarpp(unsigned i); — описывается BDD с одной вершиной, соответствующее булевой функции v_i 

- bdd_nithvarpp(unsigned i); — описывается BDD с одной вершиной, соответствующее булевой функции v_i 

- bddtruepp; — константа *true*

- bddfalsepp; — константа $false$

Получение функции из набора переменных и из другий логических функций происходит с использованием операций `&`, `|`, `\^`, `>>`, `bdd_not`, либо с помощью функции $bdd\_apply(bdd_1, bdd_2, bddop)$, где *bddop* — это любой логический оператор ($bddop\_or, bddop\_and$, и так далее).

3.1.2 язык SMV

На языке SMV описываются модули — модели среды и системы. В описание модулей входит: описание их внутренних переменных, их начальных состояний, правила переходов среды и системы. Также описываются условия справедливости, требующие определенного поведения модулей.

Описание грамматики языка SMV не стандартизировано: в каждом проекте используемый синтаксис языка может отличаться. Однако, одно из наиболее полных описаний языка представлено в статье [?]

Язык, на котором описываются модули для данного программного средства, основан на SMV и включает в себя его часть. Ниже представлено описание грамматики данного языка:

`MODULE <name>[<args>] <body> ::`
| `"MODULE main VAR " atom ":" moduleName ";" Modules`

Modules ::
| `Module "\n" Modules`
| ε

Module :: — модель недетерминированный конечный автомат
| `"MODULE " atom "(" args ")" ModuleData` - Аргументы системы - это внешние переменные. Текущая система не может их

устанавливать - внутри её правил переходов к этим переменным не может применяться оператор $next()$

ModuleData ::
| `"VAR" Variables ModuleData` - описание внутренних переменных данной системы переходов
| `"ASSIGN" Assign ModuleData` - описание начальных состояний у внутренних переменных
| `"TRANS" Trans ";" ModuleData` - описание правил переходов
| `"JUSTICE" JusticeAll ModuleData` -
| ε

Variables ::
| `VarName ":" VarType ";" Variables`
| ε

VarName ::
| `atom`

VarType ::
| `"boolean"`
| `""`$VarValues$

VarValues ::
| `VarValue ", " VarValues`
: `VarValue`

VarValue ::
| `atom`

Assign ::
| `"init(" atom ") := " VarValue ";" Assign`
| ε

VarValue ::
| `"TRUE"` - константное значение $true$ для переменных типа $boolean$
| `"FALSE"` - константное значение $false$ для переменных типа

boolean

Transition :: — символьные правила переходов системы, представленные булевыми формулами. Дорлнительно используется оператор $next()$

| `VarName`

| `"!" VarName` - логический оператор "НЕ"

| `"!" (Expression)` - логический оператор "НЕ"

| `Expression "&" Expression` - оператор "И"

| `Expression "|" Expression` - оператор "ИЛИ"

| `Expression "->" Expression` - оператор "импликация"

| `Expression "<->" Expression` - оператор "эквивалентность"

| `Expression "!=" Expression` - оператор "XOR"(сложение по модулю 2)

| `"next(" VarName ")"` - LTL-оператор X (Next)

| `"(" Expression ")"`

JusticeAll :: - Каждое требование справедливости интерпретируется с префиксом из операторов GF: "в системе переходов бесконечно часто верна булева формула Justice".

| `Justice ";" JusticeAll`

| `Justice`

Justice :: - требования справедливости, представленные булевыми формулами. Не используется оператор $next()$.

| `VarName`

| `"!" VarName` - оператор "НЕ"

| `"!" (Expression)` - оператор "НЕ"

| `Expression "&" Expression` - оператор "И"

| `Expression "|" Expression` - оператор "ИЛИ"

| `Expression "->" Expression` - оператор "импликация"

| `Expression "<->" Expression` - оператор "эквивалентность"

| `Expression "!=" Expression` - оператор "XOR"(сложение по модулю 2)

| `"(" Expression ")"` | `VarName "=" VarValue` - атомарное утверждение для перечислимых переменных, соответствует выражению "значение переменной $VarName$ равно $VarValue$"

`atom` — символьное слово

3.1.3 Описание основных методов

В процессе разработки программного продукта, реализующего синтез, были описаны следующие классы. Помимо названий классов, ниже описаны наиболее важные их методы:

- class Variable — Класс, описывающий внешние и внутренние переменные модулей:
  `int* IDs` — возвращает указатель на массив номеров тех вершин BDD, которыми кодируется данная переменная. Для переменных типа *boolean* длина массива равна 1, для переменных перечислимого типа, где количество возможных значений N, массив будет иметь длину $\log_2(N)$. Используется тип $int*$, так как при многие методы библиотеки $BuDDy$ используют именно такой тип аргументов.
  `int* NextIDs` — возвращает указатель на массив номеров тех вершин BDD, которыми кодируется данная переменная с оператором next(), то есть, штрихованная переменная.
  `unsigned GetVarBDDCnt` — возвращает количество вершин BDD, необходимых для описания данной переменной. Для переменной *boolean* это одна вершина, для переменных перечислимого типа с N возможными значениями это $\log_2(N)$ вершин
  `bdd varValueBDD(std::string value)` — Используется при работе с переменными перечислимого типа. Возвращает BDD, соответствующее утверждению $var = value$.
  `bdd nextVarValueBDD(std::string value)` — Используется при работе с переменными перечислимого типа. Возвращает BDD, соответствующее утверждению $next(var = value)$.
  `std::string GetNameFromBDD_DOT(bdd fun)` — метод, сопоставляющий данному BDD строку - текстовое описание значения переменной. Если данная переменная имеет тип *boolean*, достаточно значения var или $\overline{var}$. Если переменная имеет перечислимый тип, ее строковое представление имеет вид $v = val$.

- SMVModule — класс, описывающий конкретный модуль. В $GR-$игре речь идет о двух экземплярах класса -

`SMVModule env, SMVModule sys`
`std::string GetName()` — метод, возвращающий имя данного модуля
`bdd GetInitial()` — символьное описание начального состояния в виде BDD
`bdd GetTransition()` — символьное описание правил переходов данного модуля в виде BDD. Описывает конечный автомат
`std::vector<bdd> GetJustice()` — набор всех требований справедливости. Каждое требвание является булевой фромулой и представляется в виде BDD. Префикс GF перед каждым требованием опускается.
`Variable GetVariable(std::string varName)` — Получение элемента класса $Variable$ по ее строковому названию. Если переменной с таким именем не существует,
`std::vector<Variable> GetInternalVariables()` — набор всех внутренних переменных модуля.
`std::vector<int> GetVariablesIds()` — набор всех номеров вершин BDD, которые исользуются для описания всех внутренних переменных
`std::vector<int> GetVariablesNextIds()` — набор всех номеров вершин BDD, которые исользуются для описания всех ..... внутренних переменных

- GRGame — класс, описывающий данную GR-игру.
`GRGame(SMVModule environment, SMVModule system)` — основной конструктор класса. Для создания элемента класса достаточно передать два элемента класса $SMVModule$: описание модулей среды и системы.
`bdd WinningRegion()` — Вычисление выигрышного региона и выигрышных для системы стратегий.
`bdd getConroller(bdd winreg)` — Синтез контроллера GR-игры. Готовый контроллер представляется в виде BDD. Аргументом принимается описание выигрышного региона в виде BDD.
`void Minimize(bdd &trans)` — Минимизация контроллера GR-игры
`void printDot(std::string fileName, bdd bddtrans)` — Печать синтезированного контроллера $bddtrans$ в файл $filename$.

Описание контроллера имеет синтаксис и формат файла DOT (???)

- FileText — Класс, описывающий *smv*-файл
  `void removeExtraData()` — Очистка текста файла от комментариев и их разбиение на отдельные токены.
  `void readSMVModules()` — Создание элементов класса $SMVModule$ из файла.
  `std::vector<std::string> getModuleNames()` — Метод, возвращающий набор всех имен модулей, которые представлены в *smv*-файле.
  `SMVModule GetModule(std::string moduleName)` — Создание экземпляра класса $SMVModule$, описание которого было прочитано в *smv*-файле.

Описание основных команд для синтеза

Ниже описаны основные команды, которые должны быть выполнены при синтезе контроллера: от чтения файла до вывода контроллера.

```
FileText file("[filename].smv");
file.readSMVModules();

SMVModule sys(file.GetModule("sys"));
SMVModule env(file.GetModule("env"));
GRGame arbiter2(env, sys);

bdd win_reg = arbiter2.WinningRegion();

bdd jds = arbiter2.getConroller(win_reg);
arbiter2.removeStuttering(jds);

arbiter2.Minimize(jds);
% arbiter2.removeStuttering(jds);
arbiter2.printDot("[dotFile].dot", jds);
```

3.2 Реализация задачи с арбитром

Рассмотрим задачу арбитра для протокола взаимного исключения. Существует несколько (два и более) процессов–клиентов, запрашивающих доступ к разделяемому ресурсу. В произвольный момент времени клиенты могут отправлять запрос на пользование ресурсом. При этом, в каждый момент времени разделяемый ресурс может использоваться не более, чем одним клиентом. Представим формальное описание поведения клиентов (среды) и требуемого поведения арбитра (системы), а также требования справедливости

Формальное описание.

Допустим, в задаче речь идет об N клиентах. Каждый клиент в момент, когда у него появляется необходимость доступа к ресурсу, делает запрос на доступ. Для этого он устанавливает значение переменной $r_i = TRUE$. Арбитр управления доступом к ресурсу даёт разрешения на доступ к ресурсу, устанавливая значение переменной $g_i = TRUE$.

Запрос на доступ к ресурсу должен оставаться поднятым до тех пор, пока доступ не дадут. То есть, если запрос к ресурсу поднят, а доступ к нему в данный момент времени закрыт, то на следующем шаге запрос не будет опущен. Таким образом, требования к поведению среды описываются символьно как: $\rho_e = \wedge_i((r_i \wedge !q_i) \Rightarrow (next(r_i) = r_i))$

Система не может одновременно давать разрешение разным клиентам на доступ к разделяемому ресурсу. Если поднят запрос от некоторого клиента, и разрешение есть (!!!), то запрещать доступ на следующем шаге нельзя. Если запроса на данном шаге нет, то ..... $\wedge_{i \neq j}((g'_i \wedge g'_j) \wedge \wedge_i(r_i = g_i) \Rightarrow (g'_i = g_i))$

Изначально, все флаги запросов на доступ к ресурсу и флаги разрешений доступа опущены: $\wedge_i(\overline{r_i} \wedge \overline{q_i})$

Синтезируемая совместная система должа удовлетворять условию сильной справедливости: $\phi = \wedge_i(GF(r_i \wedge g_i)) \Rightarrow \wedge_j GF(r_j = g_j)$ Условие сильной справедливости представляется в виде набора условий слабой справедливости, и разбивается на предположения: $\phi_e = \wedge_i GF(r_i \wedge g_i)$ и гарантии: $\phi_s = \wedge_j GF(r_j = g_j)$.

Представление на языке SMV

Перечисленные выше требования к арбитру протокола взаимного исключения можно представить на языке SMV. Ниже приведен

пример требований для случая двух клиентов.

В примере представлены описания двух модулей — моделей среды и системы. У каждого модуля есть свои внутренние переменные, описаны их начальные значения. Правила $TRANS$ переходов модулей — это правила поведения ρ_e и ρ_s, представленные на языке SMV. Предположения (как условия слабой справедливости среды) и гарантии (как условия слабой справедливости системы) описываются правилами JUSTICE.

```
MODULE main
  VAR
    e : env(s.g1, s.g2);
    s : sys(e.r1, e.r2);

MODULE env(g1, g2)
  VAR
    r1 : boolean;
    r2 : boolean;
  ASSIGN
    init(r1) := FALSE;
    init(r2) := FALSE;
  TRANS
    ((r1 & !g1) -> (r1 <-> next(r1))) &
    ((r2 & !g2) -> (r2 <-> next(r2)));
  JUSTICE
    !(r1 & g1);
    !(r2 & g2);

MODULE sys(r1, r2)
  VAR
    g1 : boolean;
    g2 : boolean;
  ASSIGN
    init(g1) := FALSE;
    init(g2) := FALSE;
  TRANS
    (!(r1 <-> g1) | (g1 <-> next(g1))) &
    (!(r2 <-> g2) | (g2 <-> next(g2))) &
    !(next(g1) & next(g2));
```

Таблица 3.1. Время синтеза контроллера для задачи арбитра с

lines	Переходы	old		new		jtlv
		WinReg	Synthesis	WinReg	Synthesis	WinReg
2	24	0.001	0.003	0.009	0.007	0.012
3	540	0.01	0.007	0.032	0.013	0.021
4	2348	0.4	0.02	0.058	0.020	0.057
5	9762	1.4	0.074	0.109	0.039	0.235
6	39038	3.6	0.19	0.193	0.078	1.759
7	151072	14.3	0.71	0.346	0.211	16.261
8	568888	51.5	2.74	0.668	0.507	121.15
9	2094290	178.7	10.83	1.32	1.424	—
10	—	—	—	3.22	3.396	—

```
JUSTICE
  r1 <-> g1;
  r2 <-> g2;
```

3.3 Результаты

В следующей таблице приведено сравнение быстродействия, количества узлов ..... !!!!! сказать ранее, что за JTLV!!!! Все вычисления производились на компьютере ..... Характеристики ПК: Процесор Intel Core i5-2410M 2.30GHz Объем оперативной памяти 4Gb.

Можно увидеть преимущества и недостатки минимизации: количество вершин в системе переходов, описывающей синтезируемый контроллер, уменьшается, как правило, более, чем в 2 раза.

При этом растет количество переменных BDD. Это обусловлено тем, что при минимизации возникает необходимость хранения дополнительной информации о номере стратегии jx. И из–за увеличения количества переменных увеличивается размер самих BDD, то есть, количество вершин.

По таблице видно, что рост размера системы переходов и времени выполнения ее синтеза происходит экспоненциальным образом. Однако, главный параметр, определяющий, насколько выполнены поставленные цели — это скорость роста времени синтеза относительно

скорости роста размера синтезируемой системы.

3.3.1 Минимизация BDD

Изменение порядка переменных

Как показано в книге [Model Chacking, Karpov,Y.G.] (9.3, p. 208), размер BDD сильно зависит от предложенного порядка переменных. На начальных этапах разработки программного средства был выбран следующий порядок переменных BDD:

1. Все переменные среды: $env_1, env_2, ..., env_k$
2. Все переменные системы: $sys_1, sys_2, ..., sys_l$
3. Все штрихованные переменные среды: $env'_1, env'_2, ..., env'_k$
4. Все штрихованные переменные системы: $sys'_1, sys'_2, ..., sys'_l$
5. Переменная номера стратегии jx
6. Штрихованная переменная номера стратегии jx'

При этом выполнение многих операций, таких как ...(primeAllVariables) требовало перестроения дерева BDD, что требовало некоторого количества времени. Так как в работе мы имеем дело с символьным представлением систем переходов, то выполнение операции перехода Next к следующим состояниям происходит большое количество раз, чем значительно увеличивает общее время вычислений.

Однако, можно рассмотреть иной вариант порядка расположения переменных BDD.

1. Первая переменная (неважно, среды или системы): var_1
2. Первая штрихованная переменная: var'_1
3. Вторая переменная (неважно, среды или системы): var_1
4. Вторая штрихованная переменная: var'_1 .....(и так далее)
5. Переменная номера стратегии jx
6. Штрихованная переменная номера стратегии jx'

Таблица 3.2. Ускорение синтеза при изменении порядка переменных

lines	Переходы	old		new	
		WinReg	Synthesis	WinReg	Synthesis
2	24	0.001	0.003	0.009	0.007
3	540	0.01	0.007	0.032	0.013
4	2348	0.4	0.02	0.058	0.020
5	9762	1.4	0.074	0.109	0.039
6	39038	3.6	0.19	0.193	0.078
7	151072	14.3	0.71	0.346	0.211
8	568888	51.5	2.74	0.668	0.507
9	2094290	178.7	10.83	1.32	1.424
10	—	—	—	3.22	3.396

Данное расположение имеет существенное преимущество: при применении операции Next к BDD, состоящей из нештрихованных переменных, ... С учетом данных замечаний о порядке переменных, на дальнейших этапах разработки программного средства работа с переменными была изменена так, чтобы

Удаление недостижимых переходов

Правила переходов синтезированного контроллера, как правило, содержут большое количество недостижимых переходов. При минимизации размера контроллера, помимо "сворачивания"состояний, различающихся лишь номером jx-стратегии, можно также удалять неиспользуемые переходы. Переходы системы, не достижимые из начального состояния, после минимизации также не будут достижимы. Поэтому их можно удалить до минимизации.

```
jds = arbiter2.GetAllReachableTrans(jds);
```

Замечание. Однако, после минимизации контроллера могут возникнуть новые недостижимые состояния и переходы. Поэтому после применения метода minimize() необходимо их повторное удаление.

Ниже представлено сравнение времени минимизации полной системы переходов со временем, требуемым для удаления недостижимых состояний системы и последующей минимизации системы, состоящей только из достижимых состояний.

Таблица 3.3. Выигрыш при минимизации

lines	before	later
2	0.003	0.004
3	0.007	0.007
4	0.017	0.014
5	0.030	0.021
6	0.076	0.049
7	0.220	0.075
8	0.661	0.184
9	2.099	0.381
10	6.4	1.042

Глава 4

Реализация синтеза систем переходов из расширенного класса GR–формул

4.1 Расширение GR

Преобразования требований, чтобы привести их к виду GR.

4.2 Реализация задачи AHBA

Описание задачи на естественном языке.
Описание задачи на формальном языке.
Представление на SMV?

4.3 Результаты

Быстродействие, количество узлов
Сравнение с JTLV

Заключение

Глава 5

Основные определения и постановка задачи

5.1 Реагирующие системы

Необходимость параграфа:

Речь о синтезе идет касаемо именно реагирующих систем - нас интересует их поведение при взаимодействии с внешней средой. Требования предъявляются к таким системам, и целью работы является синтез контроллера **реагирующей** системы, удовлетворяющей заданным требованиям.

Должно быть введено определение реагирующей системы, сказано о действиях среды и системы, порядке их взаимодействия.

5.2 Линейная темпоральная логика

Формулы, с помощью которых представляются требования, относятся к классу LTL. Описать, почему нужны именно формулы LTL (важна привязка к времени, возможность описать большой класс требований). Дать формальное определение.

Описание формата $GR(1)$. Описать преимущество использования именно формул из подкласса LTL.

5.3 Представление систем переходов

5.3.1 Символьное представление

Цель - ввести описание символьного представления систем переходов. Правила поведения среды и системы удобно описывать в символьном виде.

Сказать про удвоение количества переменных, описывающих систему: x' или $next(x)$.

Итог - представление конечного автомата в виде булевой функции.

5.3.2 Представление с помощью BDD

Любые булевы функции удобно представлять в виде BDD.

Следствие - системы переходов (конечные автоматы и автоматы Бюхи) можно представлять в виде BDD.

Описать преимущества.

Описать программное представление BDD: BuDDy [1].

Итог - удобное программное представление систем переходов с возможностью простого программно и быстрого по времени изменения правил переходов.

5.4 Just Discrete System

Необходимо ввести определение JDS для того чтобы через них описать понятие реализуемости. Также целью синтеза является именно получение JDS по описанным требованиям.

Можно не вводить понятие Fair Discrete System, так как FDS не используются в работе, а используются лишь fairness-free FDS, которые как раз и являются по определению JDS.

$FDS\ D(V, \theta, \rho)$ удовлетворяет требованию ϕ, или, $D| = \phi$, если .....

5.5 Реализуемость требований

Цель - ввести понятие реализуемости требований. Необходимо для использовании в постановке задачи.

Формальное определение: Требования $\phi \in LTL$ реализуемы, если существует $FDS\ D(V, \theta, \rho)$ такой, что $D| = \phi$.

Просто на словах, неформальное определение: Существует ли поведение системы, удовлетворяющее предъявленным LTL-требованиям, что для любых допустимых действий среды система сможет действовать таким образом, чтобы у их совместного поведения выполнялись условия справедливости.

Объяснить, что таких поведений может быть несколько. Объяснить, что они могут выполняться/не выполняться в разных состояниях.

5.6 Постановка задачи

Так как основные понятия, необходимые для конкретизации постановки задачи, определены, то уже на данном этапе можно дать строгие требования к задаче, которая здесь решается. Кратко: на входе задачи имеем:

- описание поведения среды (с оператором X=next)
- требования, которым должно удовлетворять поведение системы (с оператором X=next)
- условия справедливости ϕ_{GR1}: требования к взаимодействию среды и системы в формате $GR(1)$
- описание начальных состояний среды и системы

Для предъявляемых выше требований необходимо проверить их реализуемость. (т.е. существует ли FDS-контроллер .....)

Если требования реализуемы из начальных состояний среды и системы, необходимо построить контроллер $D(V = X \cup Y, \theta, \rho_{es})$ - совместную модель поведения среды и системы, которая будет удовлетворять предъявленным требованиям поведения и справедливости: $D| = \phi_{GR1}$.

После построения контроллер должен транслироваться в формат, удобный для визуального представления его поведения.

Глава 6

Проверка реализуемости

6.1 Симуляционная игра

Цель параграфа - формализация взаимодействия среды и системы.

Симуляционная игра формально описывает взаимодействие среды и системы.

Реализуемость формулы будет сводиться к поиску победителя в симуляционной игре между двумя игроками - средой и системой.

Симуляционной игрой называется набор $G : (V, X, Y, \theta, \rho_e, \rho_s, \phi)$, где: .....

Описать, кто когда ходит, какие цели, условия выигрыша среды/системы.

6.2 Мю-исчисление

Формулы μ-исчисления необходимы при вычислении выигрышного региона: там идет работа с множествами состояний. А множества состояний можно удобно представлять в виде формул μ-исчисления; также легко определить все необходимые операции над множествами состояний.

Ввести понятие $\mu-$исчисления: множество состояний, в которых верна данная LTL-формула.

Описать, как определяются все операции: $\neg$, $\vee$, $\wedge$. Дать определение и пояснить смысл функции cox(step); пояснить смысл

GreatestFixPoint и LeastFixPoint.

6.3 Вычисление выигрышных регионов

Описать алгоритм вычисления выигрышных регионов.

Пояснить, что также происходит вычисление выигрышной для системы стратегии.

Описать, за что отвечает каждый fixPoint.

6.3.1 Проверка реализуемости

Описать, как из выигрышного региона понимаем, реализуема ли формула ϕ.

Итог - полное описание процесса проверки реализуемости GR(1)-формулы засчет вычисления выигрышного региона и сравнения с начальными состояниями среды и системы.

Глава 7

Синтез

7.1 Синтез

Описать, как определяется и в чем смысл у ρ_1, ρ_2, ρ_3.
Доказать, что система детерминирована.

7.2 Минимизация системы

Проблема при синтезе - в итоговой JDS много лишних состояний.
Описать алгоритм минимизации - удаления похожих состояний.
Замечание. При этом увеличивается количество переменных BDD.

7.2.1 Результаты минимизации

Привести таблицу сравнения размеров JDS-контроллера, количества переменных BDD и количества вершин BDD до и после минимизации.

Результат - представлен основной алгоритм.

Глава 8

Чтение и вывод

Необходимо добавить оболочку алгоритма - чтение из файла удобного формата и вывод в удобный для проверки и изучения контроллера формат.

8.1 Задание симуляционной игры с помощью формул LTL

Цель - перевести представление симулящионной игры в более удобное и понятное. Разделить GR(1) требования на импликацию $Assumptions => Guarantees$.

Получить две системы с условиями справедливости.

Таким образом, можно представлять два SMV-модуля отдельно с условиями справедливости, а при вычислении выигрышных регионов рассматривать $\bigwedge_i GFJ_i^e -> \bigwedge_j GFJ_j^s$

8.1.1 Исользование языка SMV для описания начальных данных

Необходим синтаксис для удобного описания начальных данных: переменных, начальных состояний и правил переходов - для среды и системы. А также для описания GR(1) требований к поведению контроллера. Поэтому описываем все начальные данные в виде модулей smv ?сказать о том, как разбить GR(1) на assumptions and guarantees.

Описать используемый синтаксис языка SMV:

- module main;
- boolean and enumerable types;
- VAR, ASSIGN, TRANS, JUSTICE

8.2 Трансляция синтезированной системы в DOT-файл

Для удобной работой с полученным контроллером (в дальнейшем) нужна его визуализация: необходимо транслировать его в программу, строящую изображение системы переходов.

Результат - вкупе с предыдущими главами описаны все этапы синтеза с обоснованиями корректности: по smv-файлу, описывающему поведение среды и системы, а также требованиям к контроллеру, получаем контроллер, представленный программно и визуально.

Глава 9

Задача арбитра

Описание задачи контроллера взаимного исключения при совместном доступе к разделяемому объекту (арбитр нескольких линий):

- постановка задачи
- действия среды
- действия системы
- требования к контроллеру

9.1 Решение задачи арбитра с использованием BuDDy и c++

Необходимо представить входные данные и результат:
Вставить описание задачи арбитра взаимного исключения на языке smv.
Вставить граф - визуализацию контроллера.

9.2 Проблема порядка переменных

Описание проблемы, с которой столкнулся я сам. Если расположить переменные как:

- все sys
- все env
- все next(sys)
- все next(env)

То время выполнения будет расти экспоненциально из-за роста BDD.

Но можно расположить иначе: Для каждой переменной системы и среды сделать порядок
i - var from env, sys
i+1 - next(var)

9.3 Решение задачи арбитра с использованием JTLV

9.3.1 Про JTLV

Описать, зачем JTLV.

Описать, как пользоваться - пример с вычислением выигрышного региона.

9.4 Сравнение

Таблица, сравнивающая два вида программ на c++, мою программу на jtlv и из статьи Питермана:

- время вычисления выигрышного региона
- время на синтез (только c++)
- время на минимизацию

Глава 10

Решение задачи арбитра для AHB

(возможно, глава будет удалена)

10.1 Постановка задачи и описание шины

10.2 SMV

Описать smv-спецификацию, подаваемую на вход в c++ и jtlv [3].

10.3 Сравнение

Сделать график сравнения.

10.4 Замечания по требованиям к арбитру для AHB

Описать неполноту требований. [5, 4]

Заключение

- Описан парсер с языка SMV
- Реализовано вычисление выигрышных регионов
- Реализован синтез контроллера
- Произведена трансляция контроллера в читаемый и визуально понятный вид
- Рассмотрены примеры задач, для которых может потребоваться синтез контроллера

Сравнение с результатами вычислений в статье Питермана [2]: выяснилось, что описанная в статье синтезированная система переходов содержит ошибки.

Сравнение времени синтеза для задачи арбитра:

- в статье Питермана
- с использованием JTLV
- на c++

Литература

[1] *Lind-Nielsen J.* Buddy: Binary decision diagram package: Tech. rep.: IT-University of Copenhagen, 2002. — jun2008.

[2] *Piterman N., Pnueli A., Sa'ar Y.* Synthesis of reactive(1) designs // Verification, Model Checking, and Abstract Interpretation / Ed. by E. A. Emerson, K. S. Namjoshi. — Vol. 3855 of *Lecture Notes in Computer Science*. — 2006. — Pp. 364–380.

[3] *Sa'ar Y.* Jtlv home page. — http://jtlv.ysaar.net/. — 2018. — май.

[4] *Unknown.* articles about amba/ahb, e.g. arm966e-s, technical reference manual. — Unknown. — 2018. — май.

[5] *Unknown.* по amba ahb. — http://books.ifmo.ru/file/pdf/728.pdf, page 119-. — 2018. — май.

Приложение А

Визуализация контроллера - арбитра для 2 линий

Приложение B

Приложение 2: Программа, реализующая синтез, на языке C++

(?!) - надо ли прикладывать? Или необходимо лишь выложить список описанных классов и методов?

Приложение C

Приложение 3: Программа SMV, описывающая модель задачи арбитра для двух линий

Приложение D

Приложение 4: Программа SMV, описывающая модель AHB

$FDS\ D : (V, \theta, \rho)$ Требования слабой справедливости $J_i \in J$ описывают те условия, которые должны выполняться в FDS бесконечно часто. Формально их можно представить на языке формул линейной темпоральной логики как $\bigwedge_{i=1..m} GJ_i$.

Требования сильной справедливости $(p_i, q_i) \in C$ представляют собой пару утверждений, таких что должно выполняться условие: если на данной FDS бесконечно часто истинно утверждение p_i, то должно бесконечно часто выполняться утверждение q_i. Формально такие требования представляются на языке LTL-формул как

$$\bigwedge_{1 \le i \le m} (GFp_i \Rightarrow GFq_i)$$

Обязательным элементом документа является оглавление. После него рекомендуется привести список обозначений и сокращений. Для оформления заголовков разделов, не подлежащих нумерации (введение, заключение, список обозначений и т.п.), следует использовать ко-

манду `\Chapter`. Для оформления рисунков и таблиц следует использовать окружения `Table` и `Figure`, которые принимают следующие аргументы:

1. Необязательный параметр, указывающий предпочтительное место (h,p,t,b) расположения таблицы или рисунка. Не рекомендуется указывать этот параметр без необходимости.

2. Название рисунка/таблицы.

3. Метка, на которую должна присутствовать ссылка в тексте работы.

Замечания и предложения по использованию пакета следует направлять доц. Трифонову П.В. на адрес petert@dcn.ftk.spbstu.ru.