Д.Е. Иванов

ГЕНЕТИЧЕСКИЕ АЛГОРИТМЫ ПОСТРОЕНИЯ ВХОДНЫХ ИДЕНТИФИЦИРУЮЩИХ ПОСЛЕДОВАТЕЛЬНОСТЕЙ ЦИФРОВЫХ УСТРОЙСТВ

НАЦИОНАЛЬНАЯ АКАДЕМИЯ НАУК УКРАИНЫ
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ И МЕХАНИКИ

Д.Е. Иванов

ГЕНЕТИЧЕСКИЕ АЛГОРИТМЫ ПОСТРОЕНИЯ ВХОДНЫХ ИДЕНТИФИЦИРУЮЩИХ ПОСЛЕДОВАТЕЛЬНОСТЕЙ ЦИФРОВЫХ УСТРОЙСТВ

Донецк 2012

УДК 004.3:681.518
И 20

Утверждено к печати учёным советом Института прикладной математики и механики НАН Украины (протокол №8 от 30.10.2012г.)

Рецензенты:
Скобцов Ю.А. - доктор технических наук, профессор
Каргин А.А. - доктор технических наук, профессор

Иванов Д.Е.
И 20 **Генетические алгоритмы построения входных идентифицирующих последовательностей цифровых устройств** / Д.Е. Иванов. – Донецк, 2012. – 240с.

ISBN 978-966-02-6608-7

В монографии рассматриваются вопросы построения генетических алгоритмов идентификации цифровых устройств. Предложены модели применения таких алгоритмов, на основании которых показывается практическое «конструирование» различных прикладных методов построения идентифицирующих последовательностей широкого класса. Разрабатываются методы построения параллельных версии генетических алгоритмов на основании схемы «хозяин-рабочий» и схемы «островов» для параллельных вычислительных систем широкого класса. Описываются структура и построение системы моделирования и идентификации, основой которой служат разработанные эволюционные методы.

Целевой аудиторией данной монографии являются научные сотрудники, работающие в области проектирования и производства цифровых систем, а также в области разработки средств проектирования таких систем. Книга должна вызвать интерес у аспирантов и студентов, обучающихся на соответствующих специальностях.

Ил. 45. Табл.18. Библиогр.: 223-240с. (179 наим.)

ISBN 978-966-02-6608-7

УДК 004.3:681.518

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Цифровая техника нашла широкое применение как в производстве, так и в быту. В обеих сферах применения потребитель вправе требовать высокой надёжности эксплуатируемых устройств. Стремительный технологический прогресс позволяет проектировать всё более сложные цифровые устройства (ЦУ), включая «системы на кристалле» (SOC). Для обеспечения их надёжности требуется соответствующее развитие методов автоматизации проектирования ЦУ, их контроля и диагностики. Одним из мощных направлений развития таких методов являются эволюционные вычисления.

В настоящей монографии рассматриваются вопросы применения эволюционных методов, в частности генетических алгоритмов (ГА), к задачам диагностики и контроля цифровых устройств. Данная цель достигается путём практической разработки моделей применения ГА и их компонент, что позволяет конструировать методы, включая параллельные, построения широкого класса идентифицирующих последовательностей (ИдП). Основной материал монографии составляют результаты, полученные автором за последнее десятилетие. Они являются развитием результатов донецкой школы технической диагностики [1-3] в область эволюционных вычислений.

Первую главу монографии следует рассматривать как вступительную, где даются необходимые предварительные сведения. В начале кратко рассмотрены модели цифровых устройств и неисправностей, принципы логического моделирования. Используется материал из [1-40]. Далее рассматриваются основные задачи построения диагностических последовательностей ЦУ, а также приводится обзор методов решения таких задач. Обзор основан на работах [41-124] Глава заканчивается описанием простого генетического алгоритм, где показывается техника построения новых популяций решений и применения эволюционных операций. Используются результаты из [125-129].

Во второй главе вводятся две модели применения ГА в задачах построения ИдП ЦУ: одно- и двухуровневые. Вначале строится шаб-

лон одноуровневых ГА-методов[1] построения ИдП, что выполняется путём описания компонент: кодирование особей и популяций, эволюционные операции и т.д. Подробно рассмотрен вопрос построения оценочных функций особей-последовательностей на основании статических и динамических параметров моделирования. Далее на основании данного шаблона разрабатывается ряд одноуровневых ГА-методов решения практических задач: логическая инициализация ЦУ, достижение (подтверждение) состояния ЦУ, верификация эквивалентности поведения двух заданных ЦУ. Используются результаты работ [102-123, 130-137].

Третья глава посвящена разработке ГА-методов построения тестов на основе двухуровневой модели. Сначала рассматриваются два подхода к решению задачи построения проверяющих тестов. Первый подход основан на стратегии, в которой вначале производится активизация неисправности, а затем с помощью ГА активизирующая последовательность расширяется до проверяющей. Во второй стратегии сначала с помощью метода существенного пути выполняется активизация неисправности и распространение её влияния на внешние выходы ЦУ в начальный момент модельного времени. Затем с помощью ГА строится последовательность достижения состояния, которое необходимо для такого распространения. Далее в главе описывается метод построения диагностических тестов ЦУ, в котором на каждой итерации строится диагностическая последовательность для некоторого множества неисправностей, которые не удалось различить к текущему моменту времени. При написании раздела использованы результаты из [45-62, 138-152].

В четвёртой главе рассматриваются вопросы построения параллельных версий ГА рассматриваемых методов. В частности разрабатываются методы параллелизации по схеме «хозяин-мастер» для слабо- и сильно параллельных вычислительных систем (ВС) с общей памятью. Далее в главе разрабатывается схема «островов» параллелизации, целью которой является улучшение качества поиска решений. В

[1] Поскольку термины «эволюционные алгоритмы», «генетические алгоритмы» и т.д. используют термин «алгоритм», показывающий алгоритмическую реализацию некоторого метода, то возникает терминологическая путаница. В указанных случаях под терминами «эволюционный алгоритм», «генетический алгоритм» и т.д. будем понимать классы методов, основанных на эволюции решений. Также во избежание терминологической путаницы будем применять термины ЭА-метод, ГА-метод, ГА-ориентированный метод и т.п.

главе использованы результаты работ [153-164].

Пятая глава посвящена вопросам практической реализации рассмотренных ГА-методов. Также в главе показано построение системы моделирования и идентификации, в которой упор сделан на применении эволюционных методов. Приведены основные эксплуатационные характеристики системы. Материал раздела использует работы [165-176].

ПЕРЕЧЕНЬ УСЛОВНЫХ СОКРАЩЕНИЙ

ВС	-	вычислительная система;
ГА	-	генетический алгоритм;
ИдП	-	идентифицирующая последовательность;
ИнП	-	инициализирующая последовательность;
КА	-	конечный автомат;
КЭ	-	комбинационный эквивалент;
НЗ	-	неизменяющая замена;
ПАЭ	-	последовательностная аппаратурная эквивалентность;
ПДС	-	последовательность достижения состояния;
ПГА	-	параллельный генетический алгоритм;
РГА	-	распределённый генетический алгоритм;
САПР	-	система автоматизации проектирования;
СБИС	-	сверхбольшая интегральная схема;
СО	-	симуляция отжига;
ТНЗ	-	трёхзначная неизменяющая замена;
ТЭ	-	трёхзначная эквивалентность, трёхзначный эквивалент;
УП	-	установочная последовательность;
ХП	-	характеристическая последовательность;
ЦС	-	цифровая схема;
ЦУ	-	цифровое устройство;
ЭА	-	эволюционный алгоритм.

Глава 1

ВВЕДЕНИЕ

1.1. Модели цифровых устройств

Логическое моделирование является основным средством при автоматизации проектирования и диагностике цифровых устройств. Оно основано на построении математических моделей ЦУ. Традиционно выделяются пять уровней моделирования.

- *Системный уровень* представляет устройство с наименьшей детализацией и служит для моделирования потоков данных в системе, тогда как значения сигналов на линиях устройства не вычисляются.
- *Уровень регистровых передач* или функциональный уровень, на котором устройство детализируется до функциональных блоков. Основное назначение - верификация дизайна на высоком уровне перед его детализацией.
- *Логический или вентильный уровень* используется для моделирования логического поведения как исправных устройств, так и ЦУ с неисправностями.
- *Переключательный уровень*, на котором моделирование выполняется обработкой транзисторов, обладающих набором заданных характеристик.
- *Схемный уровень* для определения поведения решает специальным образом составленные схемные уравнения.

Два последних уровня используются для моделирования схем малой размерности ввиду значительных вычислительных ресурсов.

При исследовании поведения ЦУ необходимо построение их моделей, которые являются абстракцией схемы с уровнем детализации, достаточным для проводимых исследований.

В качестве объекта исследования в данной работе выбираются цифровые устройства на логическом уровне представления, которые предназначены для обработки информации, представленной в цифровом виде. Выделяют два класса таких устройств [1-3, 4]:

- комбинационные ЦУ, в которых в произвольный момент времени входные значения сигналов однозначно определяют значения на выходе;
- последовательностные ЦУ, в которых в произвольный момент времени для определения выходных значений необходимо знать не только входы устройства, но и внутреннее состояние в предыдущий момент времени.

При построении моделей используются два основных подхода: функциональный и структурный.

Функциональный подход рассматривает только логику функционирования устройства, при этом внутренняя структура не учитывается. Пусть $X = (x_1, x_2, ..., x_n)$ - вектор входных переменных ЦС, $Y = (y_1, y_2, ..., y_m)$ - вектор выходных переменных и $Z = (z_1, z_2, ..., z_k)$ - вектор состояний. Тогда комбинационную математическую модель можно задать системой булевых функций:

$$\begin{cases} y_1 = f_1(x_1, x_2, ..., x_n); \\ ... \\ y_m = f_m(x_1, x_2, ..., x_n). \end{cases} \qquad (1.1)$$

Для последовательностных устройств математической моделью является абстрактный конечный автомат:

$A = \{Z, X, Y, \delta, \lambda\}$, где

Z - конечное множество состояний автомата;

X - конечное множество входных символов автомата;

Y - конечное множество выходных символов автомата;

$\delta : Z \times X \to Z$ - функция перехода;

$\lambda : Z \times X \to Y$ - функция выхода.

При задании поведения последовательностного ЦУ с помощью булевых функций система уравнений выше усложняется за счёт определения переменных состояний и принимает вид:

$$\begin{cases} y_1 = f_1(x_1,...,x_n,z_1,...,z_k); \\ ... \\ y_m = f_m(x_1,...,x_n,z_1,...,z_k); \\ z_1 = g_1(x_1,...,x_n,z_1,...,z_k); \\ ... \\ z_k = g_k(x_1,...,x_n,z_1,...,z_k). \end{cases} \quad (1.2)$$

Такое задание ЦУ позволяет представить его в виде блока комбинационной логики и блока памяти, которые реализуются D-триггерами (FF) (рис.1.1).

Степень детализации на функциональном уровне является тем ограничителем, который не позволяет решать актуальные задачи, связанные с идентификацией. С другой стороны, моделирование поведения ЦУ на структурном уровне предоставляет существенно больше информации, что позволяет эффективно использовать её в рассматриваемых задачах. Поэтому в качестве основной модели в работе будет использоваться модель синхронных последовательностных ЦУ (цифровых схем) на структурном уровне детализации.

Функционирование синхронного последовательностного ЦУ заключается в следующем. Одновременно с подачей синхроимпульса

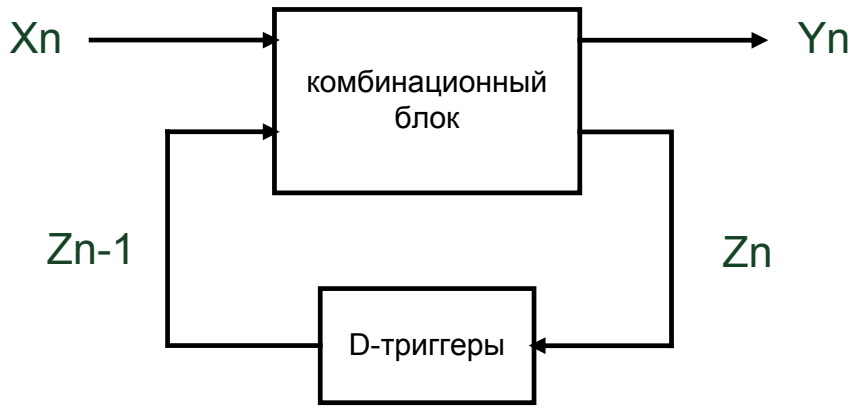


Рис.1.1. Представление последовательностного ЦУ в виде комбинационного блока и блока памяти.

на внешние входы комбинационного блока подаётся очередной входной набор X_n, а из блока памяти на его же вход подаётся вектор состояния в предыдущий момент времени Z_{n-1}. После этого отрабатывает комбинационный блок, формируя выходные сигналы Y_n и вектор состояний Z_n. Индекс n здесь показывает номер такта модельного времени. После этого ожидается подача очередного синхроимпульса. Логика подачи синхроимпульсов практически во всех задачах не рассматривается, поскольку носит вспомогательный характер.

Для описания ЦУ используется логическая сеть, в которой вершинами выступают логические элементы, входы и выходы ЦУ, а также узлы разветвлений. Контролепригодное проектирование требует построения правильной сети, в которой выходы никаких двух элементов не соединены между собой, а функция выхода ЦУ может быть представлена как функция выхода конечного автомата. При этом в работе будут проводиться некоторые аналогии между выбранной и функциональной моделями.

При описании ЦУ на структурном уровне используются языки описания. Одним из наиболее широко распространённых является текстовый формат описания ISCAS-89 [5]. В данном формате описание синхронного последовательностного ЦУ содержит несколько зон. Первая зона содержит строки комментариев, начинающихся с символа «#». Здесь приводятся основные данные об устройстве: число внешних входов и выходов, число элементов состояния (D-триггеров), число логических элементов. Вторая зона в каждой строке описывает внешний вход или выход устройства с использованием ключевых слов INPUT и OUTPUT соответственно. В третьей зоне каждая строка содержит описание одного элемента состояния (ключевое слов DFF). В следующей зоне перечисляются все логические элементы устройства в следующем формате:

```
Имя_Элемента=Тип_Элемента(Список_Предшественников)
```

Используются следующие типы вентилей:

`AND`	- логическое И	`XOR`	- исключающее ИЛИ
`NAND`	- логическое И-НЕ	`NXOR`	- исключающее ИЛИ-НЕ
`OR`	- логическое ИЛИ	`DFF`	- D-триггер

NOR - логическое ИЛИ-НЕ BUFF - повторитель
NOT - логическое НЕ

Для наглядности на рис.1.2 представлено графическое описание самого простого устройства (схемы) s27 из каталога ISCAS-89. Ниже приведено её текстовое описание в формате ISCAS-89.

```
# 4 inputs
# 1 outputs
# 3 D-type flipflops
# 2 inverters
# 8 gates (1 ANDs + 1 NANDs + 2 ORs + 4 NORs)
INPUT(G0)
INPUT(G1)
INPUT(G2)
INPUT(G3)
OUTPUT(G17)
G5 = DFF(G10)
G6 = DFF(G11)
G7 = DFF(G13)
```

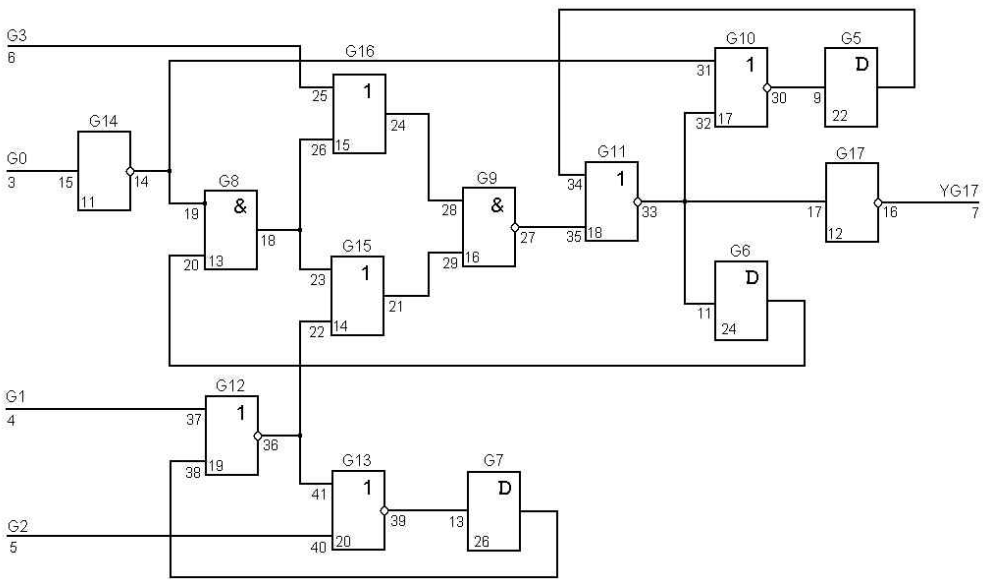


Рис.1.2. Пример графического описания схемы s27.

```
G14 = NOT(G0)
G17 = NOT(G11)
G8 = AND(G14, G6)
G15 = OR(G12, G8)
G16 = OR(G3, G8)
G9 = NAND(G16, G15)
G10 = NOR(G14, G11)
G11 = NOR(G5, G9)
G12 = NOR(G1, G7)
G13 = NOR(G2, G12)
```

Отметим также, что такой простой формат представления позволяет легко реализовывать ввод описания ЦУ для дальнейшей обработки.

При логическом моделировании ЦУ необходимо обозначать физические сигналы, которые распространяются по схеме. Для этого вводится понятие алфавита моделирования: во время моделирования значения на всех линиях в схеме могут быть элементами только заданного множества. Минимальным алфавитом, который используется при моделировании работы ЦУ, является двоичный алфавит $B_2 = \{0,1\}$. В данном алфавите символ «0» соответствует низкому уровню сигнала, символ «1» - высокому.

Неопределённость начального состояния схемы, а также явления состязаний сигналов заставляют вводить символ неизвестного состояния «u», При этом строится трёхзначный алфавит $E_3 = \{0,1,u\}$ [6]. Задание поведения основных вентилей в трёхзначном алфавите приведено в табл.1.1-1.3.

На практике с целью повышения адекватности процессов моделирования и построения тестов используют алфавиты и большей значности [7]. Однако для целей данной работы достаточно логического моделирования в алфавите E_3 и мы не будем останавливаться на алфавитах большей значности. Соответствующие замечания будут сделаны там, где исследование будет предполагать эффективное расширение на логики большей значности.

Логическое моделирование ЦУ используется в двух основных приложениях: моделирование исправных схем и моделирование с неисправностями.

Таблица 1.1

Задание вентиля И в 3-значном алфавите.

И	0	1	u
0	0	0	0
1	0	1	u
u	0	u	u

Таблица 1.2

Задание вентиля ИЛИ в 3-значном алфавите.

ИЛИ	0	1	u
0	0	1	u
1	1	1	1
u	u	1	u

Таблица 1.3

Задание вентиля НЕ в 3-значном алфавите.

НЕ	0	1	u
	1	0	u

Главной целью моделирования поведения исправного ЦУ является верификация его дизайна [7]. При этом получаются временные диаграммы для внешних входов и выходов ЦУ, а также для других необходимых внутренних линий, например, для множества контрольных точек.

Алгоритм моделирования поведения синхронного ЦУ на одном входном наборе представляется итеративным процессом. Здесь одна итерация состоит в подаче очередного набора на вход ЦУ. После этого вычисляются значения для всех элементов ЦУ. Если в результате такого вычисления выходные значения некоторых элементов изменились, то происходит новая итерация. Итерации завершаются в момент

стабилизации значений сигналов, либо при превышении заданного порога числа итераций.

Существует два принципиально различных способа обработки последовательности вентилей внутри одной итерации. При сквозном моделировании происходит вычисление значений сигналов для всех вентилей устройства. Событийный способ предполагает обработку только тех элементов, для которых произошло изменение значения сигнала хотя бы на одном из входов. Очевидно, что второй способ должен обеспечивать существенно большее быстродействие. Именно событийное моделирование используется в реализации всех рассматриваемых в работе методов.

Для моделирования поведения синхронных последовательностных ЦУ применяется техника их разворачивания в комбинационный эквивалент (блок). При этом итеративно повторяется КБ устройства, а само моделирование выполняется по тактам времени. Пример построения комбинационного эквивалента для схемы на рис.1.3 приведён на рис.1.4. В такой интерпретации каждая копия КБ соответствует такту модельного времени.

Очень часто при проектировании ЦУ возникает необходимость выполнять моделирование с неисправностями. Основное его назначение заключается в следующем:

- определение качества тестовой последовательности на основании заданной метрики; наиболее часто в качестве метрики выступает полнота теста;
- формирование диагностических словарей неисправностей;
- анализ поведения устройства в присутствии неисправности/неисправностей;
- использование как вспомогательного инструмента в программах генерации тестов для определения оценки потенциальных решений.

Перечисленные задачи являются центральными в жизненном цикле проектирования ЦУ, поэтому средства моделирования с неисправностями являются неотъемлемой частью современных САПР проектирования ЦУ.

Для решения задачи моделирования с неисправностями необходимо определить понятие «неисправность». Неисправность называется логической, если в её присутствии изменяется логика функционирования ЦУ. При этом моделирование ЦУ с неисправностями обычно

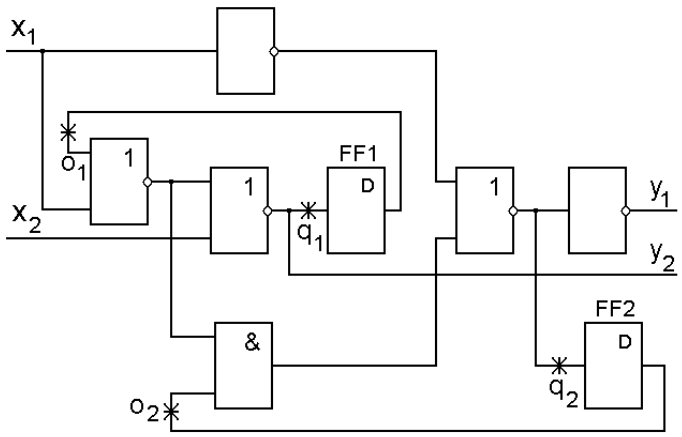


Рис.1.3. Пример последовательностного ЦУ.

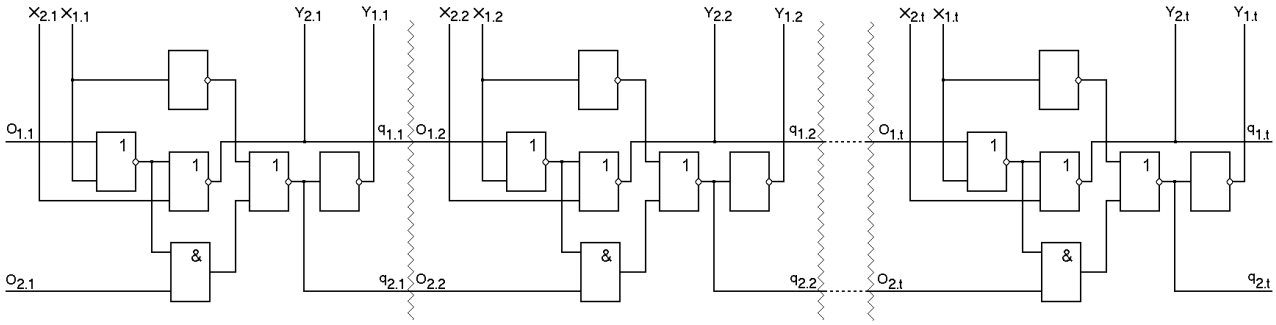


Рис.1.4. Пример разворачивания последовательностного ЦУ

в комбинационный эквивалент.

реализуется на основе исправного моделирования путём внесения влияния неисправности.

Наиболее распространенным типом рассматриваемых неисправностей являются одиночные константные неисправности. При этом любая линия связи может принять два неисправных значения:

- константа 0 (const0, в англоязычной литературе s-a-0), что соответствует физическому дефекту замыкания линии на землю;
- константа 1 (const1, s-a-1) – замыкание линии на питание.

Обычно рассматривают класс одиночных константных неисправностей, когда в ЦУ возможно наличие только одной неисправности и её присутствие не изменяется во времени.

Также широко распространенными являются следующие типы неисправностей:

- кратные константные неисправности, когда несколько линий устройства имеют постоянные значения сигналов;

- задержка распространения сигнала, что обуславливается задержкой прохождения сигнала в элементах устройства;
- неустойчивые неисправности, вызываемые наводками от соседних линий устройства;
- мостиковые неисправности, когда электрически связанными становятся две или более линий, являющихся независимыми в исправном устройстве;

Более полно различные типы неисправностей описаны в [2].

Многие из разрабатываемых в работе методов явно не ссылаются на тип рассматриваемых неисправностей и, вследствие этого, могут быть применены для всех таких типов неисправностей, для которых разработаны методы их моделирования. Однако при апробации методов мы всегда используем одиночные константные неисправности и метод их моделирования, основанный на методе [8].

Моделируемая неисправность называется проверяемой [1], если хотя бы на одном из внешних выходов ЦУ в результате моделирования устанавливаются различные значения в исправной и неисправной схемах. Часто также говорят об условной проверяемости неисправностей, когда на внешнем выходе в исправной схеме установилось определённое значение сигнала (0 или 1), а в неисправной – неопределённое (u) из алфавита моделирования E_3.

Мерой качества входной последовательности наиболее часто выступает полнота теста, которая определяется как отношение числа обнаруженных (проверенных) неисправностей $N_{пр}$ к числу моделируемых неисправностей N, выраженное в процентах:

$$P = \frac{N_{пр}}{N} \cdot 100\%. \qquad (1.3)$$

Данная мера качества тестовой последовательности будет основной в работе.

Для задач проектирования также предлагались другие метрики качества входной последовательности, например в [9].

Даже при рассмотрении класса одиночных константных неисправностей размер списка неисправностей всё равно оказывается очень большим. Если в ЦУ содержится N линий связи, что число рассматриваемых неисправностей будет $2 * N$. Именно поэтому од-

ним из важнейших качеств алгоритмов моделирования с неисправностями является скорость их работы.

С целью ускорения работы алгоритмов моделирования используют различные техники. Наиболее известными являются конкурентный, разностный, дедуктивный и параллельный по битам алгоритмы моделирования.

В алгоритме дедуктивного моделирования [10-11] проводится анализ зависимости значения сигнала на текущем входном наборе от наличия неисправности. Таким образом, для каждой линии в схеме хранится исправное значение сигнала, а также список всех тех неисправностей, которые изменяют его на противоположное. Следовательно, за один шаг моделирования определяются все проверяемые текущим входным набором неисправности. Конкурентный алгоритм моделирования [12-13] также является «одношаговым», поскольку за один проход определяются все проверяемые данных набором неисправности. Основное его отличие от дедуктивного алгоритма в том, что списки неисправностей ассоциируются не с линиями схемы, а с элементами.

Дедуктивный и конкурентный алгоритмы также реализовывались в многозначных алфавитах [10, 14]. В разностном алгоритме моделирования [15] узлы неисправных схем вычисляются и сохраняются только в том случае, когда они отличны от исправной схемы. Такой подход позволяет существенно сократить требования к ресурсам памяти.

Наиболее распространённым в настоящее время является параллельный по разрядам машинного слова алгоритм моделирования [16-17]. Данный метод использует то свойство, что логические операции над разрядами выполняются одновременно для всего машинного слова. При этом неисправные ЦУ для моделирования объединяются в группы, моделируемые в одном машинном слове.

Задача событийного моделирования больших ЦУ для одиночных константных неисправностей считается решённой на логическом уровне.

Наиболее быстрыми методами моделирования являются метод PROOFS [17] и его многочисленные последователи [18-20]. Методы ориентированы на быстрое определение полноты тестовой последовательности и используются в тех случаях, когда нет необходимости восстанавливать поведение неисправного ЦУ для всех входных набо-

ров. Для повышения быстродействия рассматриваемая неисправность удаляется из списка в том момент модельного времени, когда она обнаружена. При этом данная неисправность не вносится на моделирование для следующих входных наборов.

В данной работе при конструировании методов моделирования ЦУ с неисправностями в качестве основы используется разработанный авторами метод, описанный в [8]. Его особенностью в сравнении с [17] является параллельное по разрядам машинного слова моделирование неисправных ЦУ. В зависимости от конкретной задачи возможны два варианта реализации:

- исправное ЦУ моделируется в первом разряде машинного слова, остальные $n-1$ разрядов моделируют $n-1$ неисправных ЦУ, где n - число разрядов инструментальной ЭВМ;
- исправное ЦУ моделируется в одном машинном слове, в другом слове все n разрядов используются для моделирования неисправных ЦУ.

В обоих случаях внесение влияния неисправностей, а также сравнение поведения неисправных ЦУ с исправным устройством выполняется с помощью битовых масок. Дополнительно для повышения быстродействия при внесении влияния неисправностей используется техника фиктивных вентилей [17].

1.2. Основные задачи построения идентифицирующих последовательностей ЦУ

Задачи построения входных идентифицирующих последовательностей различного типа являются центральными в технической диагностике и связаны с задачами обеспечения надёжности и контроля ЦУ. Наиболее часто выделяют следующие их типы:

- тестовые последовательности;
- характеристические последовательности; данный класс последовательностей очень широк [21], поэтому выделяют его подклассы: синхронизирующие, инициализирующие, достижения заданного состояния и т.п.;
- верифицирующие эквивалентность последовательности.

Два последних перечисленных типа последовательностей используются как вспомогательные при построении тестов, однако имеют и самостоятельную ценность.

Одной из центральных задач технической диагностики является задача построения тестов. Различают два типа тестирования. При функциональном тестировании на ЦУ подают только воздействия, которые предусмотрены алгоритмом его работы. Диагностическое тестирование допускает подачу наборов неосуществимых в процессе реальной работы. В данной работе мы будем ориентироваться на более широкий класс диагностического тестирования.

Математическая формулировка задачи тестирования заключается в следующем. Пусть задано исправное цифровое устройств A_0. С помощью заданного класса неисправностей порождается множество его неисправных модификаций $A = \{A_1, A_2, ..., A_n\}$, причём $A_0 \neq A_i$ для $i = \overline{1,n}$. Тогда тестом, проверяющим заданную неисправность, называют входную последовательность, для которой выходные реакции устройств A_0 и A_i различны. Тестом для множества A называется такая входная последовательность, которая является проверяющей для всех $A_i \in A$.

Если тестовая последовательность позволяет просто определить исправность устройства, то говорят о проверяющих теста. Если же последовательность позволяет локализовать неисправность, т.е. указать какое именно из устройств A_i тестировалось, то говорят о диагностических тестах.

Различают задачи построения тестов для комбинационных и последовательностных ЦУ. Во втором случае задача существенно усложняется [22-23], что обусловлено невозможностью однозначного корректного восстановления поведения ЦУ вследствие неопределённости начального состояния, а также состязания сигналов.

Часто методы построения тестов последовательностных ЦУ являются развитием методов для комбинационных ЦУ.

В общем методы построения тестов делятся на вероятностные и детерминированные. Первый класс получил широко распространение ввиду простоты [24]. Данные методы не ориентированы на конкретную неисправность. В настоящее время вероятностные методы получили развитие в виде ГА.

По использованию информации о внутренней структуре схемы методы построения тестов для последовательностных схем делят на абстрактные и структурные. В первой группе в качестве математической модели используется конечный автомат, задаваемый таблицей или графом переходов и выходов. При этом информация о внутренней структуре устройства не известна и не используется. Основы теории экспериментов с автоматами были заложены в работе [25] и развиты в [26]. Данные методы основаны на построении дерева преемников или предшественников состояний и требуют явного задания каждой неисправной модификации исправного ДУ таблицей (графом) переходов и выходов. Наиболее полно абстрактные методы разработаны в работах [27-28]. Преимуществом данных методов является высокая полнота проверки неисправностей полученными тестами и гарантия нахождения решения (если таковое существует). К их недостаткам относятся большая длина тестов, необходимость генерации таблицы переходов и выходов исходного ДУ и игнорирование информации о структуре ДУ.

К структурным методам, которые не ориентированы на конкретную неисправность, относятся методы активизации критических путей [29]. Они основаны на том предположении, что половина константных неисправностей, которые расположены на активизированном пути, проверяются заданным входным набором. Поэтому требуется строить входные последовательности, которые порождают как можно больше критических путей. Они строятся от внешних входов к внешним выходам и используют технику критических кубов

К алгебраическим относят методы, основанные на применении скобочных и эквивалентных нормальных форм [30]. К основополагающим в теоретическом плане относится метод различающей функции [31]. На данном методе основаны все алгоритмы, использующие булево дифференциальное исчисление. Данные методы путём получения булевых производных позволяют аналитически получить условие распространения неисправности. StarAlgorithm [32] также сводится к построению различающей функции для получения аналитических условий распространения неисправностей.

Активизация путей распространения неисправностей в многозначных алфавитах является общей чертой всех структурных методов. При этом ранние алгоритмы реализовывали одномерную активизации путей в 3-х значном алфавите, что не гарантировало построе-

ния тестов. Поэтому позднее обратились к многомерной активизации путей в многозначных алфавитах.

На следующем этапе применяются методы, ориентированные на заданную неисправность. Первым был разработан D-алгоритм [33], использующий многомерную активизацию критических путей в 6-значном алфавит. В отличие от методов одномерной активизации, D-алгоритм гарантирует построение теста. Однако для схем, содержащих большое число сходящихся разветвлений, D-алгоритм теряет свою эффективность, вследствие большого числа возникающих конфликтных ситуаций. Из-за этого время генерации становится очень большим, а в ряде случаев процесс завершить не удаётся. Для таких схем был разработан метод PODEM [34]. К отличиям следует отнести другие процедуры D-распространения и возврата, а также иная стратегия обработки конфликтных ситуаций и выполнение процедуры импликации в прямом направлении (от входов устройства к выходам). Дальнейшие улучшения [35] направлены на уменьшение числа возвратов по дереву решений, а следовательно, и на уменьшение времени работы. Позже были разработаны методы, в которых для сокращения пространства поиска и более эффективной активизации применяются алфавиты большей значности. В работе [36] используется 10-значный алфавит, а в [37] 12-значный алфавит. В работе [38] предложена 16-значная логика.

Большинство структурных методов генерации тестов для последовательностных схем основаны на преобразовании схемы в итеративный комбинационный эквивалент путём обрыва обратных связей [1]. При этом наибольшее распространение получил подход, при котором к полученной итеративной схеме применяется один из методов генерации тестов для комбинационных ЦУ.

Структурные методы, которые по своей идеологии аналогичны абстрактным, впервые предложены и разработаны в [39]. Поскольку в данном подходе учитывается информация о выходных реакциях всех копий комбинаторного эквивалента схемы, то он получил название кратной стратегии наблюдения. Математически условия активизации неисправностей строятся на основании аналитических выражений, представляющих структуру устройства. В зарубежной литературе данный подход появился позже и был развит в [40].

Для больших ЦУ с количеством вентилей в реализации от 5 тысяч, проектирование которых осуществляется в настоящее время, за-

дача построения тестов остаётся открытой. Это связано с тем, что структурные методы не могут обработать такие устройства: либо происходит переполнение стеков возвратов в процедурах выбора непротиворечивых альтернатив, либо не удаётся полностью построить дерево решений. Поэтому в последние два десятилетия развилось направление, в котором алгоритмы дают не точное решение задачи, а лишь приемлемое в некотором практическом смысле. Для задачи построения тестов таким критерием может служить достижение заданной полноты. При этом для произвольной наперёд заданной неисправности такие методы не дают однозначного ответа о том, существует ли для неё тест и, следовательно, относятся к группе вероятностных.

Простейшими представителями данного направления являются случайные и псевдослучайные методы построения тестов, которые были названы выше. В работе [41] генерация теста основана на процедурах моделирования с неисправностями. Идеи алгоритма SOCRATES, который основан на технике обучения, развиты в работе [42].

Однако наиболее сильно в настоящее время развивается направление, связанное с генетическими алгоритмами. Как и в других подходах адаптация ГА к проблеме построения тестов началась с комбинационных ЦУ.

Первыми работами, в которых ГА применяется к задаче построения тестов, были [43-44]. Реализация генератора тестов CRIS в [43] использует, вообще говоря, больше специфических свойств, чем требует реализация ГА. В этой связи данный метод является, скорее, гибридным. При этом очень тяжело определить, что оказывает большее влияние на его конечную эффективность: ГА или используемые специфические эвристики. В работе же [44] ГА применяется к только к решению задачи построения тестов комбинационных ЦУ.

В работах [45-46] впервые описан простой ГА генерации тестов последовательностных ЦУ. Однако и здесь особь представляла собой входной двоичный вектор, а не последовательность. При этом применяемые эволюционные операции скрещивания и мутации также соответствуют простому ГА. Оценочная функция в методе прямо показывает число вновь обнаруженных данным вектором неисправностей. Также для ускорения вычисления фитнесс функции рассматривается только часть возможных неисправностей в ЦУ.

В работе [47], по видимому, впервые предложено кодирование особей ГА в виде двоичной матрицы, соответствующей входной последовательности. Данное кодирование является в настоящее время стандартом «де-факто» на логическом уровне представления ЦУ.

Именно две данные школы, Иллинойский университет и Политехнический университет в Турине, дали мощный импульс развитию в этом направлении.

Наиболее полно задача тестирования ЦУ на вентильном уровне с помощью ГА рассмотрена в [48]. Задача построения теста одной неисправности решается разделением на две фазы: активизация и последующее тестирование с помощью ГА. Здесь также проведено сравнение разработанного подхода с наилучшими известными структурными методами. Многие последующие работы являются развитием данной в том числе и авторов [49-52]. Главным отличием данных работ от других является параллельное вычисление оценок 32 особей одновременно путём параллельного моделирования соответствующего числа неисправных ЦУ в разрядах машинного слова, что существенно повышает скорость работы метода. Часто различные предложенные методы различаются только методами моделирования, используемыми для вычисления оценки особей-последовательностей, а также применёнными эвристиками.

Особенностью ГА оказалось то, что они способны за короткое время построить тест для достаточно большой части всех неисправностей, тогда как для при построении теста остальных неисправностей время превышало установленный лимит, и они оставались непротестированными. Это привело к появлению гибридных генераторов, в которых ГА использовался либо на одном из этапов генерации теста для множества неисправностей, либо в одной из фаз генерации для отдельно взятой неисправности. Остальную работу выполняли структурные методы. Например в работе [53] на первом этапе генерация выполняется с помощью ГА. Когда же прогресс останавливается, то происходит переход ко второму этапу, реализованному на структурном методе. Похожий подход реализован в [54-56]. В [57] фаза генерации теста каждой отдельной неисправности реализована на основе двух методов: ГА и структурного. Вначале распространение неисправности на внешние выходы выполняется методом активизации одномерного пути. Далее полученные условия распространения влияния неисправности реализуются с помощью ГА.

В работе [58] ГА применяется для решения задачи построения диагностических тестов. Здесь с его помощью строится различающая последовательность для каждой пары неисправностей. При этом, как и в работах [45-46], адаптирована версия простого ГА, а особь представляет двоичный вектор. В работе [59] разработан генератор диагностических тестов на основе ГА, в котором техника обработки особей-последовательностей аналогична используемой в [48, 60-62].

Поскольку списки рассматриваемых неисправностей для больших ЦУ являются очень большими, это ведёт к повышенным временным затратам при работе генераторов тестов. В этом случае часто рассматривают не полное множество неисправностей, а некоторую случайную выборку. Пример такого подхода приведён в [45,63]. В [64] изучается вопрос построения такой выборки из полного множества неисправностей, которая бы наилучшим образом отражала диагностические свойства входных последовательностей при работе с полным множеством. В [65] ГА применяется для удаления избыточной информации из диагностических словарей, что позволяет существенно уменьшить время обработки диагностической информации.

Ещё один интересный подход предложен в [66-69]. Здесь ЦУ рассматривается с функциональной точки зрения: сумматор, декодер и т.п. Тогда особями в ГА являются операнды (числа) для соответствующих блоков. Для их обработки разрабатываются эволюционные операции, которые задаются над вещественными числами.

Дальнейшее развитие применения ГА в задаче тестирования пошло в направлении создания параллельных версий [70-80]. Особенностью реализаций подходов является то, что они, большей частью, ориентированы на доступные авторам аппаратные платформы. При этом несмотря на возможную кроссплатформенность реализации их эффективность всё равно изучается только для одной платформы. Более того, авторы работ практически не описывают механизм взаимодействия независимых компонент в таких методах.

Например в работе [81] схема островов описывается как эволюция параллельных популяций, в которых «каждый процессор выполняет один и тот же ГА эксперимент», при этом процессоры периодически обмениваются лучшими полученными в группах результатами, тогда как худшие особи удаляются из популяций. Аналогичные замечания применимы к [63].

Это же замечание касается и работ, описывающих параллельные ГА с использованием схемы «хозяин-рабочий». Например, в [82] описание параллелизации сводится к тому, что в фазах 1 и 2 метода используется параллельное по особям моделирование, а в фазе 3 – параллельное моделирование с разбиением списка неисправностей.

Поскольку совершенно отсутствует описание механизма взаимодействия компонент, то это не позволит оценивать эффективность параллельной реализации ГА, а также возможность переноса на системы другого класса.

Некоторые вопросы масштабируемости ПГА изучались в [72], однако безотносительно к задаче построения тестов.

Также полностью вне внимания исследователей остался вопрос эффективности параллельных версий ГА при работе на рабочих станциях с многоядерными процессорами. Очевидно, что работы должны проводиться как для слабопараллельных ВС, содержащих 2-4 ядра, так и сильнопараллельных ВС, в которых присутствует несколько процессоров с общим числом ядер 8 и более [83]. Особое внимание при таких исследованиях следует уделять степени «связности» методов, работающих на разных ядрах, поскольку это определяет объём информации для их взаимодействия, что в свою очередь ограничивается подсистемой общей памяти. Отсутствие системных исследований в этой области объясняется тем фактом, что такие ВС получили распространение только в последние 5-10 лет. Следует отметить, что число вычислительных ядер в рабочих станциях постоянно растёт, как и их распространённость, что показывает необходимость проводить такие исследования.

Ещё одним аспектом построения ПГА, который не освещается в известных работах, является применение в них параллельных версий методов моделирования [84-93]. Использование таких методов в обычном ГА при этом позволяет фактически строить его параллельную версию для доступной аппаратной платформы не изменяя сути метода. Единственной известной работой, в которой даются некоторые замечания по данному вопросу является [63].

С другой стороны, в последнее время параллельные версии ГА начали создаваться для многоядерных графических ускорителей, построенных по технологии CUDA [94-95], что объясняется относительной дешевизной данного типа сильнопараллельных систем при

достаточно большом числе простых вычислительных ядер. Одна из таких реализаций ПГА генерации тестов описана, например в [96].

Известны также публикации, где к проблеме генерации тестов последовательностных устройств применены метод роя частиц [97] и оптимизационная стратегия симуляции отжига [98].

Отметим также новую эволюционную стратегию Selfish [99], которая была предложена итальянскими учёными. Если в генетических алгоритмах работа строится вокруг особей и популяций, то в данной стратегии ключевым элементом являются хромосомы, которые представляют перспективные свойства с точки зрения решения задачи. К задаче построения тестов стратегия была применена на функциональном уровне при описании устройств на языке VHDL.

Ещё одним возможным подходом к решению задачи построения тестов является адаптация экспериментов с автоматами [26] на структурный уровень задания ЦУ. До настоящего времени это ограничивалось существенной сложностью таких методов. Например для метода, который основан на различении всех пар состояний, временная сложность оценивается как $O(n^4)$, где n - число состояний рассматриваемого конечного автомата A. Оценка строится на основании того, что алгоритму необходимо различить n^2 пар состояний при том, что алгоритм такого различения имеет сложность $O(n^2)$. Однако в последнее время именно в донецкой школе диагностики начали разрабатывать методы, в которых на структурный уровень переносятся автоматные методы идентификации [100-101]. Такая адаптация стала возможна с ростом вычислительной мощности рабочих станций.

Характеристические последовательности являются вспомогательными для решения задач построения тестов. Сам класс таких последовательностей зародился как вспомогательный при проведении экспериментов с ЦУ, заданными в виде конечных автоматов. Однако многие подклассы характеристических последовательностей на структурном уровне представления ЦУ приобретают дополнительный смысл и представляют самостоятельную ценность в процессах проектирования и диагностирования ЦУ. Примерами таких последовательностей являются инициализирующие и верификации эквивалентности. Решение задач построения последовательностей таких классов на структурном уровне прошло, в общем, ту же эволюцию,

что и задачи построения тестов: вероятностные, структурные детерминированные, ГА.

Одним из подклассов характеристических последовательностей являются инициализирующие последовательности. Неформальное определение инициализирующей последовательности говорит, что она должна перевести ЦУ из неопределённого состояния в некоторое определённое. Отличие от соответствующего подкласса характеристических последовательностей заключается в том, что на логическом уровне они используются в момент включения ЦУ в работу, тогда как в автоматных методах они являются вспомогательными при построении обходов деревьев решений.

Первоначально задача построения данных последовательностей решалась структурными детерминированными методами. В частности, к таким относятся методы, предложенные в [102]. Они основываются на техниках построения дерева решений и символьных операций, что позволяет их применение для схем малой и средней размерности. Естественным ограничением данных методов являются ресурсы памяти для хранения строящегося дерева, что не позволяет применять их к ЦУ большой размерности.

Развитием данных методов является работа [103], в которой предлагается выполнять декомпозицию схемы, что упрощает построение заданных последовательностей. В работе [104] проблема построения инициирующих последовательностей исследуется в связке с задачей автоматизированного построения тестов. Однако методы оказались не достаточно общими, что выражалось в невозможности построить инициализирующую последовательность для ряда ЦУ.

Наиболее полно задача исследована в работе итальянских авторов [105], где предложен соответствующий ГА построения логически инициализирующих последовательностей, а техника обработки потенциальных решений аналогична рассмотренной в [48]. При этом замечания о возможной параллелизации метода отсутствуют. Развитием данной работы является [106] тех же авторов.

Известны также попытки адаптировать к решению задачи другие эвристические методы, в частности алгоритм муравьиных колоний [107]. В [108] задача решается с помощью стратегии симуляции отжига. В [109] рассматривается задача сохранения инициализирующих последовательностей во время процесса оптимизации.

В последнее время задача рассматривается и на функциональном уровне при задании ЦУ на языках описания типа VHDL [110]. В [111] рассматривается задача верификации инициализирующих последовательностей при переходе от функционального к логическому уровню задания ЦУ, поскольку изменяется алфавит моделирования с двоичного B_2 на троичный E_3.

Ещё одним очень широким подклассом характеристических последовательностей являются последовательности верификации эквивалентности поведения двух заданных ЦУ. На автоматном уровне задания такие последовательности являются частью экспериментов с автоматами [26, 28]. На структурном уровне задания задача получает два оттенка:

- проверка эквивалентности поведения исправного и неисправного ЦУ, в частности в генераторах тестов, основанных на моделировании;
- проверка эквивалентности оригинального и оптимизированного ЦУ.

Вторая задача возникает в связи с тем, что разрабатываемое ЦУ на структурном уровне подвергается серии оптимизаций по различным параметрам или набору параметров. Такие средства оптимизации, обычно, изменяют логическую структуру схемы, минимизируя некоторые параметры, например, число вентилей в логическом блоке. Однако они должны оставлять неизменной логику функционирования, проверка чего и выполняется с помощью данных последовательностей.

Математически задача выглядит достаточно просто. Задано два цифровых устройства A_0 и A_1. Необходимо построить такую входную последовательность $S \in \Sigma$, выходные реакции на которую устройств A_0 и A_1 различны, где Σ - множество всех возможных входных последовательностей. При этом считается, что устройство A_1 получено в результате некоторой оптимизации внутренней структуры устройства A_0, т.е., вообще говоря, структура их элементов состояний одинакова, тогда как логические сети, описывающие комбинационные блоки, различны. Эквивалентной будет постановка задачи, когда говорят об исправном A_0 и неисправном A_1 устройствах.

Такая постановка является достаточно общей и для удобства разные исследователи вводят свои понятия эквивалентности. Для устройств, в которых отсутствует сигнал внешнего сброса, предложено несколько различных определений эквивалентности [112-113]. В основном они различаются в определении режима функционирования устройства. В некоторых подходах предполагается, что существует синхронизирующая последовательность, переводящая устройство в необходимое начальное состояние перед тем как начнётся процедура верификации. В [112] предлагается определение последовательностной аппаратурной эквивалентности (ПАЭ). Оно заключается в том, что если два устройства имеют одинаковое постсинхронизирующее поведение, то они являются эквивалентными. Подобное определение даётся в [113] применительно к понятию избыточных (нетестируемых) неисправностей.

В связи с проблемой оптимизации дизайна в [114] дано определение, называемое 3-значная неизменяющая замена, которое также применяется для устройств без цепей сброса и ориентировано на 3-значное моделирование. В данном определении считается, что устройства находятся в отношении 3-значной неизменяющей замены, если выходные реакции Y_1 и Y_2 соответственно являются парами следующих значений из алфавита E_3: (0,0), (1,1), (u,u), (u,0) и (u,1), при этом устройства начинают работу из полностью неопределённых состояний. Другими словами, в данном определении предполагается, что оптимизированное устройство имеет более детерминированное поведение.

Решение задачи верификации эквивалентности по аналогии с задачами построения тестов возможно точными структурными методами, которые основаны на преобразовании булевых функций [115-116]. При этом сохраняются замечания о практической применимости методов. Поэтому также были предложены ГА построения таких последовательностей [117-118], что частично позволило обойти недостатки структурных методов. Такие методы оказались эффективнее структурных, например [119]. Техника применяемая в ГА для верификации эквивалентности на логическом уровне оказалась успешной и была применена также к верификации протоколов [120] и обобщена в работе [121]. В [122-123] задача решается с помощью другой эволюционной стратегии симуляции отжига.

Поскольку в данных ГА верификация эквивалентности основана на моделировании поведения ЦУ, то методы являются относительно медленными. Однако в литературе задача построения параллельных методов данного типа не рассматривалась.

Задача построения последовательностей достижения состояний на логическом уровне представления рассматривалась лишь как вторичная в работе [57] для решения задачи построения теста и в [101] при адаптации абстрактных методов на структурный уровень. В первом случае достижение необходимого состояния в ЦУ осуществлялось с помощью ГА, тогда как во второй работе вводился специальный класс $R-$ и $S-$последовательностей. Очевидно, что во втором направлении данный тип последовательностей должен применяться очень широко и, следовательно, задача требует более системного изучения.

Делая выводы по проведённому обзору, можно отметить следующее. Задачи построения входных идентифицирующих последовательностей являются одними из центральных в жизненном цикле проектирования и контроля ЦУ. В настоящее время разработан целый ряд алгоритмов решения задач данного типа: построения функциональных и диагностических тестов, синхронизирующих последовательностей разного рода и т.д. Современные методы построения таких последовательностей базируются, в основном, на структурных и автоматных подходах, которые были разработаны для комбинационных ЦУ, а потом адаптированы к последовательностным ЦУ. Это привело к тому, что такие методы имеют неприемлемые с практической точки зрения эксплуатационные характеристики: алгоритм либо не в состоянии обработать схему, либо такая обработка длится неприемлемо долго.

Невозможность решить задачу синтеза идентифицирующих последовательностей ЦУ за приемлемое время заставляет разработчиков искать альтернативные подходы к решению таких задач. Одно из возможных направлений основано на замене задачи синтеза на итеративное применение задачи анализа, поскольку она является более простой. К этому направлению принадлежат, в частности, генетические алгоритмы. Подход с применением ГА характеризуется тем, что для проверки качества потенциальных решений используется моделирование поведения ЦУ на заданной последовательности. В зависимости от целей задачи и необходимого качества решения использует-

ся как исправное моделирование, так и моделирование ЦУ с неисправностями. При этом задачи моделирования больших ЦУ в настоящее время считаются решёнными.

В целом, направление, связанное с эволюционными подходами в задачах идентификации ЦС, считается в настоящее время наиболее перспективным [124].

1.3. Простой генетический алгоритм

В [125] Голдбергом был предложен генетический поход, который хорошо зарекомендовал себя при решении комбинаторных NP-полных задач, породив волну исследовательского интереса к области эволюционных вычислений [126-128].

Первоначально ГА рассматривался как эффективное средство поиска оптимума некоторой функции. Впоследствии он нашёл применение для широкого класса задач. В широком смысле термин «генетический алгоритм» применяют для обозначения любой математической модели, которая основана на популяциях и использует эволюционные операции (в частности селекции и рекомбинации) для построения новой выборки точек в пространстве поиска. Его суть заключается в попытке повторить природный путь улучшения характеристик живых организмов.

Для понимания эволюционной техники опишем кратко простой генетический алгоритм. Данное описание в основном базируется на [128].

Пусть задана произвольная задача и её пространство решений, в котором будет производиться поиск. Решение задачи с помощью ГА предполагает, что любая точка в заданном пространстве решений может быть закодирована как строка битов $(x_1, x_2, ..., x_n)$. Такая строка называется особью, а входящие в неё битовые переменные $x_i, i = \overline{1, n}$ называются генами или аллелями. Некоторый произвольный набор особей называется популяцией. Каждой особи ставится в соответствие значение некоторой оценочной функции $f(x_1, x_2, ..., x_n)$, которое показывает: насколько близко данная особь приблизилась к решению поставленной задачи. Таким образом, решение задачи

предполагает поиск минимума (максимума) некоторой многомерной функции f.

Проблемы кодирования решений и построения оценочной функции являются центральными при решении задачи с помощью генетического алгоритма. Рассматриваемый пример простого ГА использует простейшее битовое кодирование. Используемое в реальных задачах кодирование существенно шире. В частности используется вещественное кодирование решений, кодирование в виде матриц, графов и т.д. Используемый тип кодирования определяется предметной областью приложения ГА [127,129].

Выбор вида оценочной функции и эффективность её вычисления также в значительной степени определяют эффективность конечной реализации генетического алгоритма. Чем проще вид оценочной функции и меньше сложность её вычисления, тем меньшие временные ресурсы необходимы генетическому алгоритму для поиска оптимального решения. Часто вычисление оценочной функции сводится к моделированию решения задачи в заданной точке пространства. Для уменьшения временных затрат на вычисление оценочной функции применяют также приближённое моделирование. В частности, при решении задач построения входных ИдП для оценки потенциальных решений используется моделирование поведения ЦУ на входной последовательности, которая представлена заданной особью. При этом возможно моделирование поведения как исправного ЦУ, так и ЦУ с неисправностями.

На основе оценочных функций вычисляется фитнесс функция. Смысловая нагрузка в данном случае приобретает несколько другой оттенок. Если оценочная функция показывает качество особи в смысле решения задачи, то фитнесс функция особи показывает её качество относительно других особей популяции. Именно на основании фитнесс-функции принимается решение о выживании данной особи, т.е. о её переходе в следующее поколение. В простом генетическом алгоритме фитнесс функция особи полагается равной $f_i / \overline{f}$, где f_i - оценочная функция $i-$й особи популяции, $\overline{f}$ - средняя оценочная функция по популяции. В реализациях ГА часто не выделяют отдельно фазу вычисления фитнесс-функции. При этом принцип соревновательности особей закладывается в процесс построения новой либо промежуточной популяции.

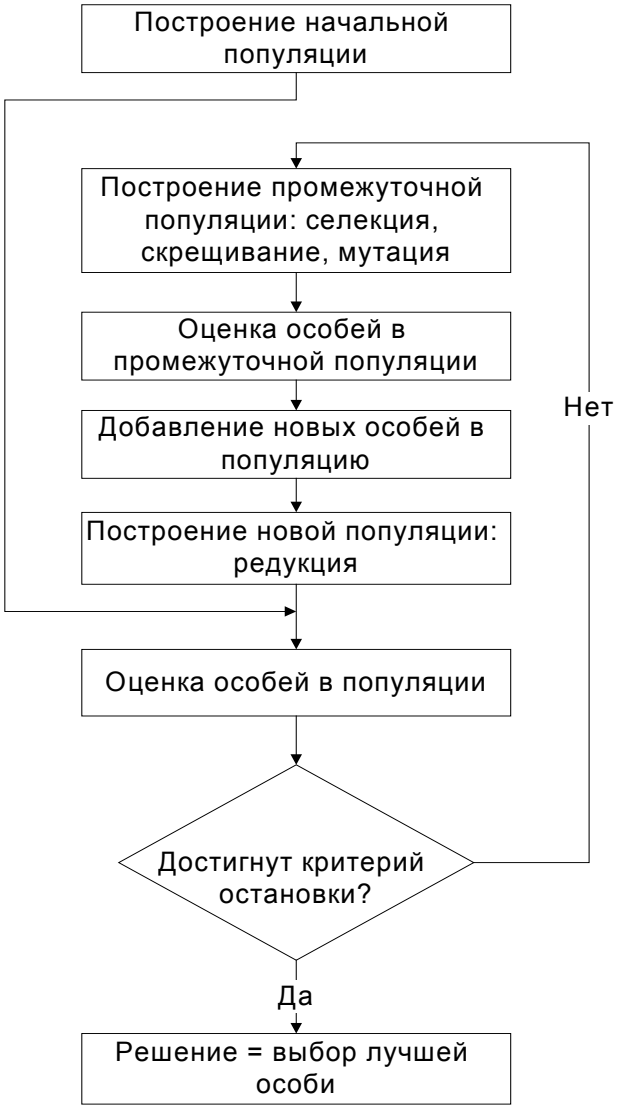


Рис.1.5. Общая схема генетического алгоритма.

В целом поиск решения задачи с помощью ГА состоит из итеративного процесса порождения новой популяции на основании текущей с помощью набора эволюционных операций (рис.1.5). Набор применяемых эволюционных операций, вообще говоря, также может варьироваться в зависимости от конкретной реализации ГА.

Процесс поиска решения обычно начинается со случайно сгенерированной популяции или предполагаемого решения. Возможно также построение начальной популяции на основе некоторых эвристик или исходя из накопленного опыта решения задач данного типа, что может улучшить сходимость метода. Выполнение одной итерации генетического алгоритма часто разбивают на два шага

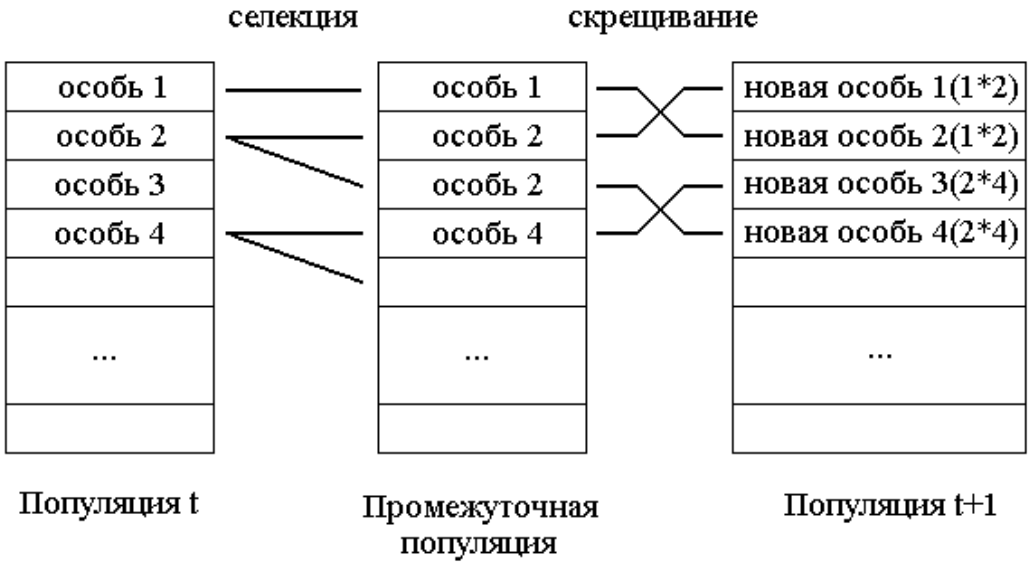
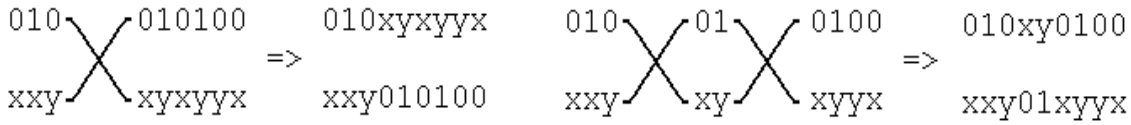


Рис.1.6. Построение нового поколения особей.

(рис.1.6). Первый шаг: из текущей популяции строится промежуточная. Выбор особей для промежуточной популяции называется селекцией. Она выполняется сразу после вычисления фитнесс функции $f_i / \overline{f}$ для каждой особи в популяции. Чаще всего реализуется метод «рулетки», когда вероятность особи с номером i попасть в промежуточную популяцию пропорциональна её фитнесс функции $f_i / \overline{f}$. При этом создаётся копия выбранной особи, которая помещается в промежуточную популяцию.

Также встречаются реализации, когда перед методом «рулетки» производится линеаризация особей. Она заключается в том, что особи сортируются в порядке возрастания оценочных функций, а в качестве фитнесс функции им присваивается порядковый номер в полученном списке.

К классическим относится и схема «остаточного стохастического отбора». В этом методе для каждой особи с номером i, для которой $f_i / \overline{f} > 1.0$, целая часть показывает число копий данной особи, помещаемых в промежуточную популяцию. После этого в промежуточную популяцию добавляются копии всех особей с вероятностью, равной дробной части $f_i / \overline{f}$. Например, для особи с $f_i / \overline{f} = 1.65$ в про-



а) одноместное скрещивание б) двуместное скрещивание

Рис.1.7 Скрещивание.

межуточную популяцию попадёт одна копия точно, а вторая копия с вероятностью 0.65.

На втором шаге из промежуточной популяции строится популяция следующего поколения. Для этого над особями промежуточной популяции выполняются некоторые трансформирующие эволюционные операции, под воздействием которых особи могут получить новые свойства и тем самым улучшить свою оценочную функцию. Обычно сюда входят операции скрещивания и мутации.

Скрещивание это операция объединения хромосом двух особей-родителей (иногда более), в результате которой получаются особи-потомоки с новыми свойствами. Вероятность применения операции скрещивания перед помещением выбранных особей в популяцию следующего поколения считается заданной и равна P_c. Реализуют как одноместную, так и многоместную операцию скрещивания. При одноместном скрещивании точка пересечения выбирается случайно. После этой точки биты в особях меняются местами. Рассмотрим пример на рис.1.7. Пусть имеются особи 010010100 и ххухухуух (во второй особи 0 и 1 заменены соответственно на х и у только для удобства восприятия). Тогда операция одноместного скрещивания будет выглядеть как на рис.1.7а, где в качестве точки пересечения случайно выбрана третья позиция.

При многоместном скрещивании случайно выбираются две (три или более) точек скрещивания (рис.1.7б). Существует также универсальное скрещивание по всем битовым позициям. В этом случае для каждой позиции случайным образом решается: следует ли обменять биты в особях-родителях.

Иногда операция скрещивания реализуется с порождением только одной особи-потомка. Выбор особей в качестве родителей в промежуточной популяции происходит случайно. При этом предполагается, что поскольку в промежуточную популяцию более вероятно попали особи с высокой оценочной функцией, то и особи-последователи будут иметь более высокую среднюю оценочную функцию, чем в предыдущей популяции. Т.е. процесс поиска будет направляться в области более близкие к оптимальному решению. Цель операции скрещивания заключается в том, чтобы направить процесс поиска в те регионы пространства, где предположительно находится решение задачи.

После скрещивания применяется операция мутации, для которой также задаётся вероятность применения P_m. Она реализуется как случайное изменение генов в особях всей популяции. Цель данной операции - направить поиск в неисследованные регионы пространства поиска и предотвратить его сходимость к локальным экстремумам. Обычно вероятность мутации битов выбирается постоянной и достаточно малой: менее 1%. Иногда вероятность мутации может изменяться в процессе поиска и зависеть от некоторых параметров. Также часто мутацию интерпретируют как установку бита в случайном гене. Видно, что при этом реальная мутация происходит примерно в 50% случаев. Очевидно, что выбор вероятностей для операции мутации очень важен и сильно зависит от предметной области. При высокой вероятности мутации будут теряться положительные свойства особей, что будет оказывать разрушительное воздействие. Напротив, при низкой вероятности мутации шанс покинуть область локального минимума становится малым, и необследованными останутся значительные области пространства поиска.

Таким образом, процесс эволюции реализуется как замена (полная или частичная) особей популяции на новые, после чего снова происходит вычисление их оценочных и фитнесс функций. Механизм выживания «сильнейших» особей является попыткой формализации использования накопленной в процессе эволюции информации. Порождение новых популяций прекращается, когда найдено решение проблемы или выполняются установленные критерии, например, превышено число итераций.

Следовательно, чтобы задать или «построить» некоторый ГА, необходимо определить понятия особи, популяции, операции скрещивания и мутации, задать оценочную функцию. Очевидно также, что эффективность генетического алгоритма зависит от целого ряда параметров: размера популяции, метода выбора особей из предыдущей популяции, скрещивания и мутации, а также вида оценочной функции и т.д. Выбор таких параметров необходимо производить на основе экспериментов с реализацией ГА.

Именно такой конструктивный подход построения ГА и будет выбран в качестве основного в данной работе.

Глава 2

ОДНОУРОВНЕВЫЕ ГЕНЕТИЧЕСКИЕ АЛГОРИТМЫ ПОСТРОЕНИЯ ВХОДНЫХ ИДЕНТИФИЦИРУЮЩИХ ПОСЛЕДОВАТЕЛЬНОСТЕЙ ЦУ

2.1. Одно- и двухуровневые генетические алгоритмы построения идентифицирующих последовательностей ЦУ

В зависимости от сложности задачи построения ИдП и структуры генетических алгоритмов для их решения выделим две модели применения:

- одноуровневые ГА-методы;
- двухуровневые ГА-методы.

В том случае, если алгоритм поиска решения (построения последовательности) может его найти за один вызов ГА, будем говорить об *одноуровневой модели генетических алгоритмов* (рис.2.1) или *одноуровневой схеме применения ГА*. В алгоритмах данного класса цель поиска ГА фактически задаётся извне и известна до начала работы всего алгоритма. При этом такая цель часто не строится в явном виде, а для ГА-метода поиска она задаётся в виде оценочной функции потенциального решения (особи).

К задачам данного класса можно отнести частные случаи задач построения синхронизирующих последовательностей различного типа. В данной главе на основании одноуровневой модели применения ГА будет показано построение методов решения следующих задач:

- построение инициализирующих последовательностей ЦУ; для заданного последовательностного ЦУ A_0, всегда стартующего из полностью неопределённого состояния $Z_u = (uu...u)$ и при полностью неопределённых значениях на всех внутренних линиях, необходимо построить такую входную последовательность S, после

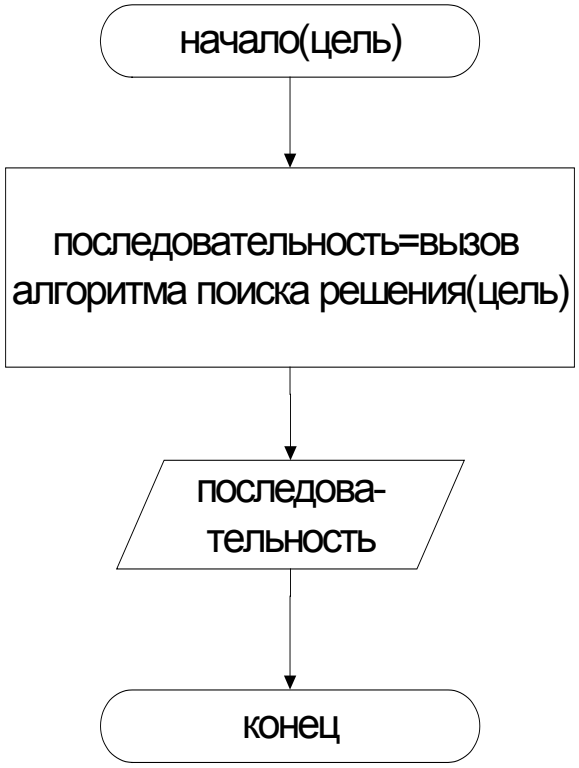


Рис.2.1. Одноуровневый ГА построения ИдП.

моделирования на которой элементы состояния A_0 примут определённые значения из алфавита B_2;

- построение последовательностей для достижения заданного состояния в ЦУ; определяющим здесь является то финальное состояние $Z_{фин}$, которое должно бать достигнуто после приложения входной последовательности S к устройству A_0;
- построение последовательностей, которые верифицируют эквивалентность двух заданных ЦУ; поскольку формальное доказательство эквивалентности является сложной задачей, то задача переформулируется в обратную: построить последовательность S (контрпример), моделирование на которой двух заданных ЦУ A_1 и A_2 порождает разные выходные реакции: $A_1(S) \neq A_2(S)$.

Видно, что к одноуровневым относятся алгоритмы решения относительно простых задач.

В том случае, если сложность задачи не позволяет методу найти решение за один вызов ГА поиска, то говорят об *двухуровневой модели* (рис.2.2) или *двухуровневой схеме применения ГА*. Методы данного

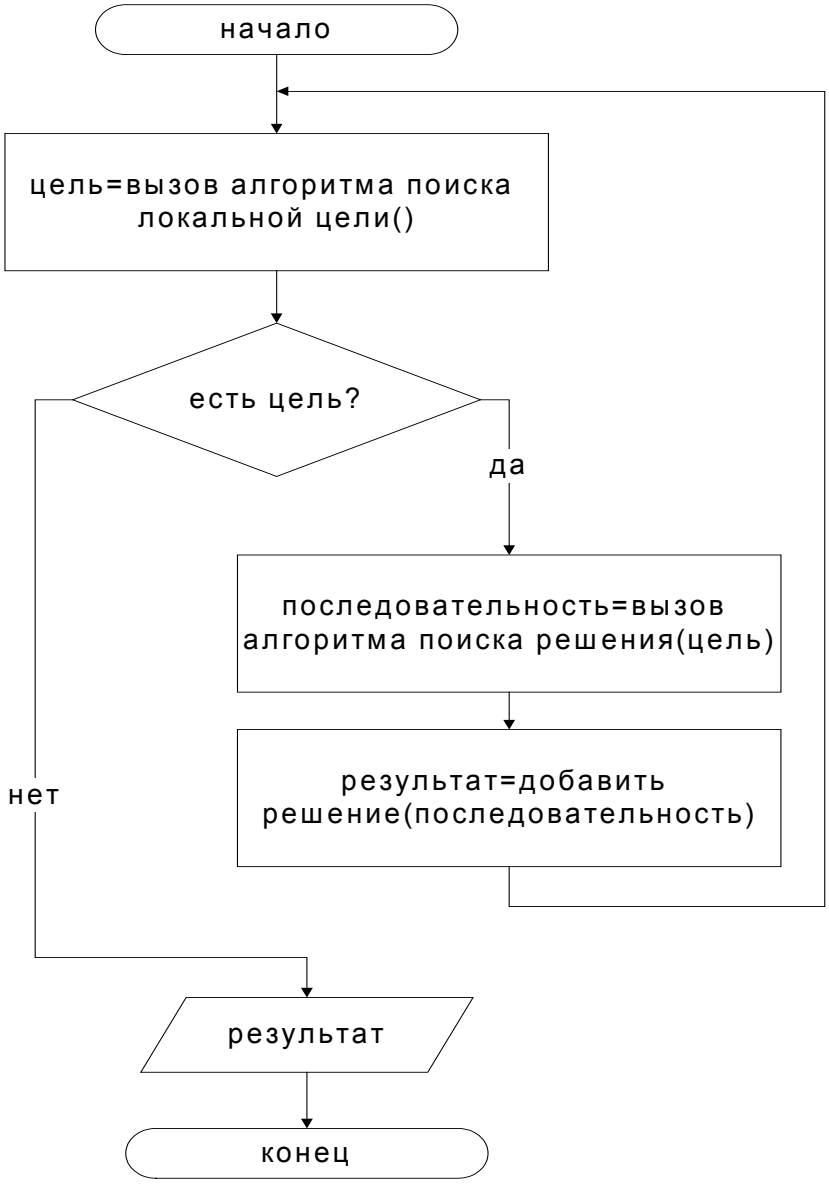


Рис.2.2. Двухуровневый ГА построения ИдП.

класса предполагают итеративную схему построения. Каждая итерация состоит из двух фаз. В первой фазе происходит поиск промежуточной (локальной) цели. Если такая цель найдена, то вызывается ГА поиска решения для данной локальной цели, который и формирует вторую фазу итерации. Итеративный поиск промежуточных целей и их достижения ведёт к решению общей задачи.

В такой постановке будем называть фазу 1 поиска локальной цели *верхним уровнем ГА*, а фазу 2 достижения локальной цели – *нижним уровнем ГА*. При этом структура фазы 2 алгоритма соответствует

одноуровневому ГА построения ИдП.

В задачах построения входных ИдП конечное решение (последовательность) часто строится по аддитивному принципу (является совокупностью промежуточных решений). Т.е. данные задачи естественным образом проецируются на двухуровневую модель применения ГА.

К классу двухуровневых алгоритмов построения ИдП, которые рассматриваются в данной работе, относятся:

- метод построения проверяющих тестов; стратегия построения тестов для последовательностных схем предусматривает сначала активизацию некоторой неисправности f_i (фаза 1) – распространение её влияния на элементы состояний, а затем транспортировку различий поведения исправного и неисправного ЦУ на внешние выходы схемы (фаза 2). Таким образом, обработка происходит для каждой неисправности из заданного списка отдельно, что порождает итеративный процесс со структурой на рис.2.2;
- метод построения диагностических тестов; данная задача является ещё более сложно в сравнении с предыдущей, поскольку входная последовательность S должна не только обнаружить влияние неисправности, но и локализовать её, т.е. указать какая неисправность из списка F породила наблюдаемое неисправное поведение. Здесь применяется иная стратегия: на верхнем уровне из текущего множества классов неразличимости выбирается один такой класс (фаза 1), а на нижнем уровне для выбранного класса неразличимости строится последовательность, которая разбивает его на более мелкие классы неразличимости (фаза 2). Процесс итеративно повторяется до того момента, пока все классы неразличимости будут состоять из одной неисправности;
- метод построения проверяющих тестов с подтверждением состояний; стратегия нижнего уровня ГА-метода построения теста одной целевой неисправности здесь разбивается на два этапа: на первом этапе для выбранной на верхнем уровне неисправности f_i выполняются активизация и распространение влияния на внешние выходы (возможно потребуется рассмотрение нескольких тактов модельного времени вперёд), а также процедура обратного распространения в такте времени, соответствующем активизации f_i. Результатом работы этого этапа является некоторое состояние Z' в

ЦУ, возможно, определённое лишь частично. На втором этапе выполняется достижение заданного состояния Z' с помощью соответствующего ГА-метода.

В данной работе мы рассматриваем методы, которые в качестве стратегии поиска используют ГА, т.е. основаны на популяционной эволюции. Однако применение одно- и двухуровневой моделей возможно и для ЭА с эволюцией одного потенциального решения. Например, в [130] подобные методы разрабатываются на основании стратегии симуляции отжига.

Будем также далее для разрабатываемых методов, которые используют ГА в качестве метода поиска, использовать название *ГА-ориентированный метод*, либо *ГА-метод*.

В данной главе описывается построение одноуровневых ГА-методов решения задач построения некоторых типов ИдП.

Двухуровневые эволюционные методы построения ИдП, в которых в качестве стратегии поиска выступает генетический алгоритм, разрабатываются в главе 3.

Задачи построения параллельных версий ГА применительно к задачам генерации ИдП рассматриваются в главе 4.

2.2. Шаблон одноуровневых ГА-методов построения идентифицирующих последовательностей ЦУ

Особенностью одноуровневых эволюционных методов является то, что их цель задаётся только один раз и она известна до начала выполнения алгоритма. Для такого ЭА формализация цели выражается в виде оценочной функции потенциальных решений. Достижение этой цели показывает завершение работы алгоритма в целом. Фактически, весь алгоритм построения последовательности и является эволюционным алгоритмом. Таким образом, структура методов данного рода такова, что основной цикл эволюции построения новых решений является в нём также самым внешним.

В широком смысле понятие генетических алгоритмов включает все такие ЭА поиска, которые основаны на эволюции популяции решений.

Формально некоторый ГА можно задать в виде множества сле-

дующих объектов:

$$\text{ГА} = \{ Pop_{нач},\ N_{особ},\ l,\ Sel,\ Cross,\ Mut,\ O,\ fit,\ p_{скр},\ p_{мут} \}, \tag{2.1}$$

где:

$Pop_{нач}$ - начальная популяция особей; часто при реализации строится случайным образом;

$N_{особ}(Pop)$ - размер популяции, который задаёт число особей, входящих в популяцию Pop; в зависимости от реализации размер популяции может оставаться неизменным от поколения к поколению, либо изменяться заданным образом;

l - длина особи в битах при двоичном кодировании; также в зависимости от реализации может изменяться или быть постоянной от поколения к поколению;

$Sel : Pop_j \to ind_i$ - оператор селекции (репродукции): выбирает из заданной популяции одну (или несколько - в зависимости от конкретной реализации) особь для выполнения генетических операций; обычно быть выбранными в качестве родителей более высокую вероятность имеют особи с большей фитнесс-функцией;

$Cross : ind_i \times ind_j \to ind'$ - оператор скрещивания: по двум заданным особям строит новую особь в соответствии с выбранным правилом скрещивания;

$Mut : ind_i \to ind'$ - оператор мутации: строит новую особь, применяя к заданной особи правило мутации;

$O = O(ind_i)$ - оценочная функция: показывает качество данной особи в смысле решения поставленной задачи;

$fit = fit(ind_i, Pop_j)$ - фитнесс-функция особи ind_i в популяции Pop_j; оценочную функцию следует отличать от фитнесс-функции, поскольку последняя показывает качество особи относительно других в популяции, то и для её вычисления необходимо знать не только оценку особи, но и оценку всех других особей в популяции; часто фитнесс-функция задаётся неявно соответствующим построением оператора репродукции;

$p_{скр}$, $p_{мут}$ - вероятности применения операторов скрещивания и

мутации соответственно.

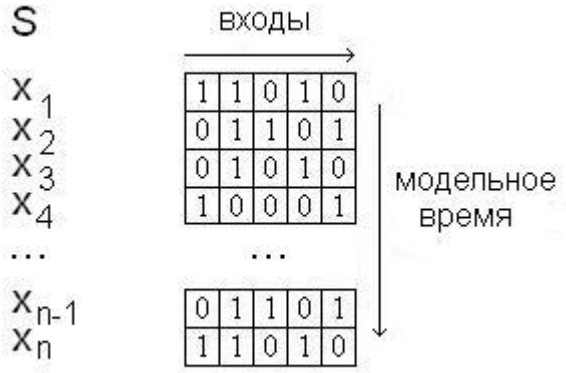
Видно, что для реализации конкретного ГА-ориентированного метода построения ИдП целый ряд его компонент необходимо задавать конструктивно: оператор репродукции, оценочная функция, способ построения фитнесс-функции и т.д. Часто при построении соответствующего метода ГА конкретная реализация таких компонент алгоритма предполагает их экспериментальное обоснование. В конечную реализацию метода входит тот вид оператора, функции и т.п., при котором получаются наилучшие числовые результаты. Будем называть такие компоненты *зависящими от реализации*.

Абстрагируясь от реализации таких компонент, можно разработать обобщённый шаблон ГА-ориентированных методов построения ИдП, которые используют одноуровневую модель применения. Такой ГА-шаблон будет включать как кодирование особей, набор генетических операций (селекция, скрещивание, мутация), так и базовую структуру метода ГА. При этом зависящие от реализации компоненты должны быть только названы, а их конкретное наполнение вынесено за данный шаблон и отнесено к построению конкретного метода, его алгоритмической реализации и настройке.

Таким образом, на основании такого шаблона далее частные ГА-методы построения ИдП будут строиться путём задания оценочной функции и наполнения зависящих от реализации компонент.

Поскольку ГА нашли широкое применение для решения различных задач, то для них разработано достаточно много компонент, а также подходов к исследованию эффективности их применения. В данном разделе при построении шаблона внимание будет акцентироваться на компонентах ГА, которые используются при построении ИдП: кодирование последовательностей и популяций, эволюционные операции, построение оценочных функций с использованием параметров, получаемых при моделировании ЦУ.

В качестве особи в ГА-методе построения ИдП выступает входная двоичная последовательность S. Для её кодирования используется двоичная матрица (рис.2.3б), в которой число столбцов соответствует числу внешних входов $N_{вх}$ обрабатываемого ЦУ A_0. При обработке синхронных ЦУ число строк в матрице соответствует числу тактов модельного времени и равно длине последовательности S.



а) б)

Рис.2.3. Представление особи в ГА построения ИдП в виде последовательности символов (а) и соответствующей ей двоичной матрицы (б).

Если устройство A_0 рассматривается в виде абстракции КА, то каждая строка матрицы кодирует некоторый символ входного алфавита. Число строк матрицы в данном случае соответствует числу n входных символов в последовательности $S = (X_1,...,X_n)$ при проведении эксперимента с автоматом (рис.2.3а).

Популяция ГА задаётся как набор таких матриц-последовательностей (рис.2.4).

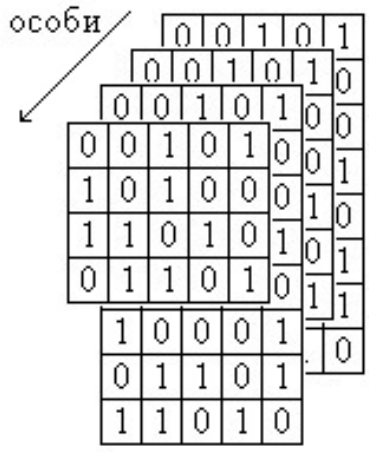


Рис.2.4. Представление популяции в ГА построения ИдП.

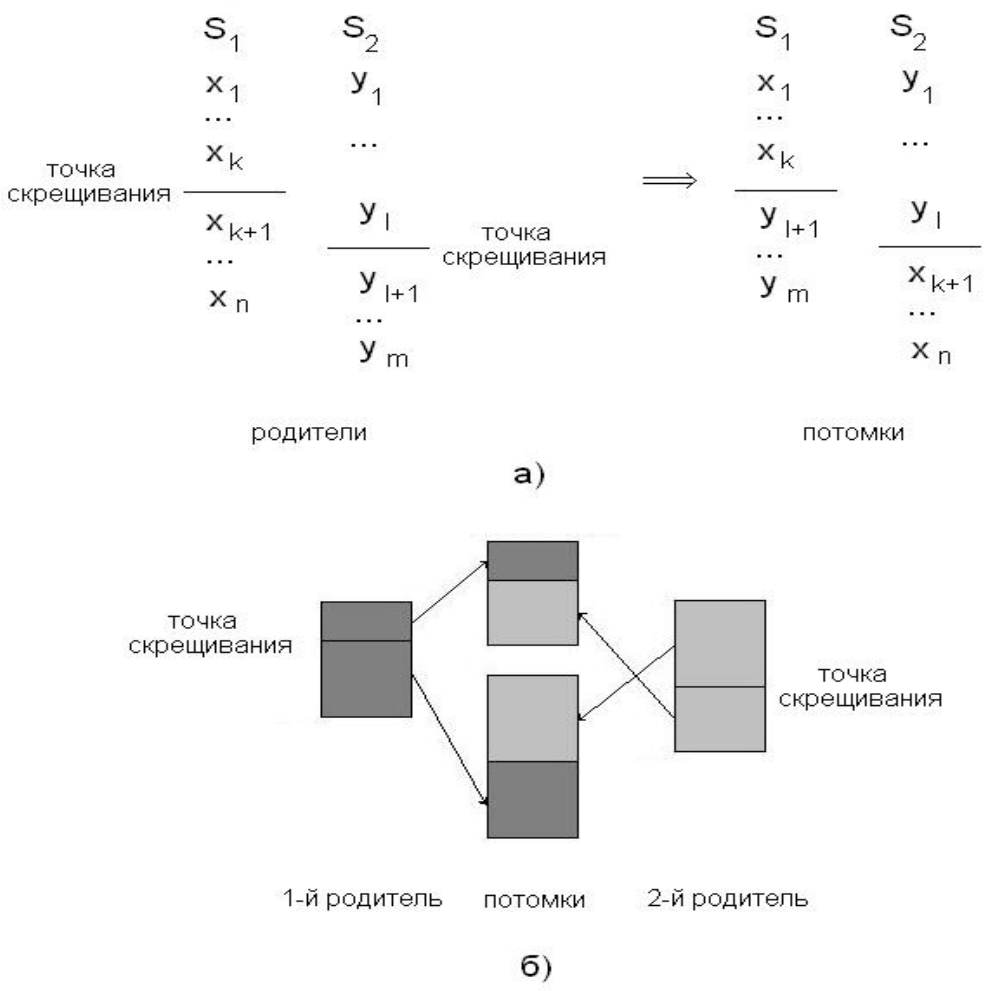


Рис.2.5. Операция горизонтального скрещивания последовательностей:

а) для автоматной модели;

б) для структурного уровня.

Для популяции Pop её размерность $N_{особ}(Pop)$ показывает число особей в ней. Оно может быть постоянным, либо изменяться от итерации к итерации. В последнем случае для популяций итераций с номерами i и $i+1$ имеем $N_{особ}(Pop_i) \neq N_{особ}(Pop_{i+1})$. Свойство постоянства числа особей в популяции определяется реализацией ГА.

Операции $Cross : ind_i \times ind_j \to ind'$ скрещивания разделим на два класса. К первому классу отнесём те операции, которые имеют аналоги для структурного и функционального уровней задания ЦУ, когда особи представлены двоичными матрицами и строками символов соответственно. Для примера на рис.2.5а представлена операция гори-

зонтального скрещивания последовательностей $S_1 = (x_1,...,x_n)$ и $S_2 = (y_1,...,y_m)$ для автоматной модели (символы x входного алфавита изменены на y во второй последовательности для удобства восприятия примера). Здесь выбираются точки скрещивания k и l в первом и втором родителях соответственно. Формирование потомков происходит следующим образом. Первый потомок состоит из k символов первого родителя и $(m-l)$ символов второго родителя. Второй потомок состоит из l символов второго родителя и $(n-k)$ символов первого родителя. Видно, что при такой операции длины особей-потомков могут отличаться от длин родителей. Для примера на рисунке длины потомков составят $(k+m-l)$ и $(l+n-k)$ символов соответственно.

Та же самая операция, но для структурного уровня представлена на рис.2.5б. Возможно применение данной операции как с одной так и с несколькими точками скрещивания.

Второй класс операций скрещивания определяется только для структурного уровня задания ЦУ, когда особи представлены двоичными матрицами. Такие операции не имеют явного аналога для функционального уровня. Для примера на рис.2.6 приведена одна из

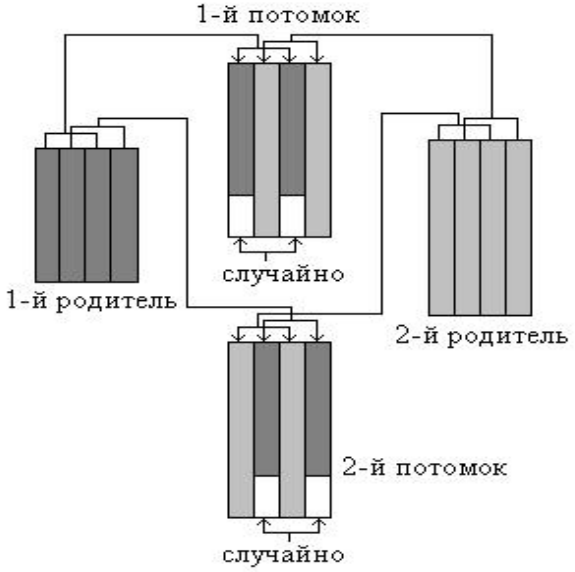


Рис.2.6. Операция вертикального скрещивания.

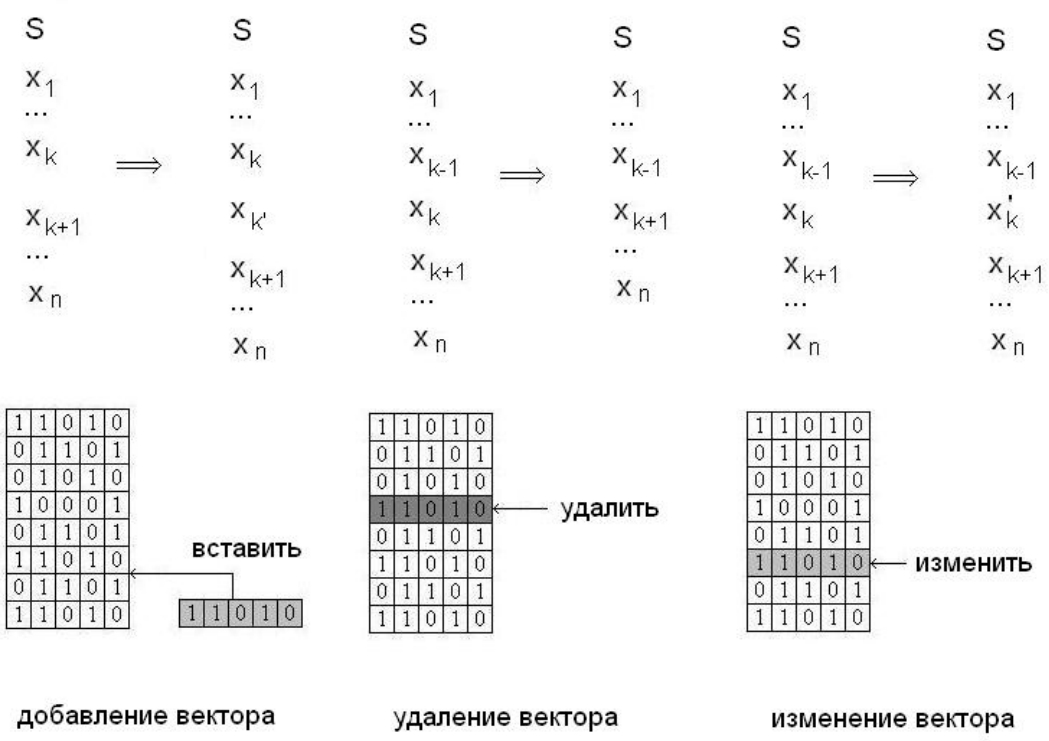


Рис.2.7. Операции мутации для различного представления особей-последовательностей.

таких операций – вертикальное скрещивание. Очевидно, что применение таких операций повышает гибкость ГА при обработке ЦУ именно на структурном уровне.

По аналогии операции мутации $Mut : ind_i \rightarrow ind'$ также можно разделить на два класса. На рис.2.7 вверху показаны операции мутации для особей, заданных в виде строк символов. Соответствующие им операции для матричного представления особей показаны на том же рисунке внизу.

На рис.2.8 показан пример операции мутации, аналог которой для особей в виде строк символов отсутствует.

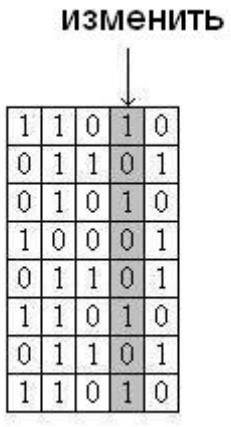


Рис.2.8. Мутация изменения столбца.

Поскольку программная реализация рассматриваемых в работе методов ГА построения ИдП будет производиться для ЦУ, заданных на структурном уровне, то будем использовать операции скрещивания и мутации обоих классов (рис.2.5-2.8).

Использование различных видов операций скрещивания и мутации не ограничивается рассмотренными. Во многих реализациях ГА-методов построения ИдП вводятся и другие операторы, например, в [131] описывается вертикальный свободный кроссинговер для особей, представленных двоичными матрицами.

Очевидно, что применение различных операций скрещивания и мутации при реализации конкретного алгоритма необходимо обосновывать экспериментально. Поскольку экспериментальное получение зависимости эффективности алгоритма от используемых генетических операций часто требует проведения очень большого числа экспериментов, во многих реализациях используют равномерный выбор генетических операций из множества используемых.

Центральным элементом ГА-методов построения ИдП является оценочная функция, которая формализует цель всего алгоритма. Для построения оценочных функций последовательностей в ГА-методах построения ИдП используются два типа параметров: статические и динамические.

Статические параметры (табл.2.1) являются характеристиками обрабатываемого ЦУ A_0 и не изменяются в процессе работы ГА. Предполагается, что если для ЦУ представлена информация об управляемости H_i и наблюдаемости I_i элементов, то она доступна не только для всех логических элементов, но и для входов, выходов и элементов состояний ЦУ, а также контрольных точек. Все статические параметры могут быть вычислены до начала работы метода.

Динамические характеристики (табл.2.2) являются свойством оцениваемой особи-последовательности S и вычисляются на основании моделирования поведения исследуемого ЦУ A_0.

В самом общем случае оценочная функция строится с учётом параметров обоих типов, т.е.:

$$O = O(A_0, S). \qquad (2.2)$$

Если в ГА-методе построения ИдП происходит обработка двух

Таблица 2.1.

Статические параметры оценочных функций

параметр	содержание	применение	замечание
$N_{вых}$	число внешних выходов	задачи построения ИдП	
$N_{вх}$	число внешних входов	задачи построения ИдП	
$N_{тр}$	число элементов состояний	задачи построения ИдП	полный список достижимых состояний не доступен
$N_{к.т.}$	число контрольных точек	задачи построения ИдП	могут отсутствовать, $N_{к.т.} = 0$
$N_{эл}$	число логических элементов, либо комбинационных блоков	задачи построения ИдП	
$\|F\|$	число неисправностей	в задачах тестирования	велико для больших ЦУ
H_i	управляемость элемента с номером i	задачи построения ИдП	для больших ЦУ часто не доступно
I_i	наблюдаемость элемента с номером i	задачи построения ИдП	для больших ЦУ часто не доступно

устройств A_1 и A_2, то (2.2) можно записать как:

$$O = O(A_1, A_2, S). \qquad (2.3)$$

В задачах построения тестов обычно рассматривается исправное ЦУ A_0 и множество неисправных ЦУ $A = \{A_1,..., A_n\}$, порождаемое соответственно некоторым множеством неисправностей $F = \{f_1,..., f_n\}$. В этом случае (2.2) принимает следующий вид:

$$O = O(A_0, A, S) = O(A_0, F, S). \qquad (2.4)$$

Очевидно, что включение всех перечисленных в табл.2.1-2.2 параметров в оценочную функцию при решении конкретной задачи может оказаться неэффективным. Например, в задачах достижения

Таблица 2.2.

Динамические параметры оценочных функций.

параметр	содержание	применение	замечание
$U\_N_{вых}$, $U\_N_{вх}$ $U\_N_{тр}$, $U\_N_{к.т.}$, $U\_N_{эл}$	число соответствующих компонент, для которых установлено заданное состояние в алфавите моделирования	в задачах достижения для исправных и неисправных ЦУ	будут введены как функции достижимости компонент $U()$ - раздел 2
$R\_N_{вых}$, $R\_N_{вх}$, $R\_N_{тр}$, $R\_N_{к.т.}$, $R\_N_{эл}$	число соответствующих компонент, для которых получены различные значения	в задачах верификации двух и более устройств	будут введены как функции различия компонент $r()$ - раздел 2, $R()$ - раздел 3
$N_{соб}$	число событий	в задачах построения ИдП, оценки рассеивания тепла	
L	длина последовательности	в задачах построения ИдП	может изменяться в соответствии с определением эффективной длины
$\|F_{тест}\|$	число обнаруженных неисправностей	в задачах тестирования	мера полноты теста
$\|F_{акт}\|$	число активизированных неисправностей	в задачах тестирования, основанных на активизации	
$N^i_{нер}$	число множеств неразличимых неисправностей мощности i	в задачах построения диагностических тестов	последовательность $N^i_{нер}$, $i = 1,...$ является одной из диагностических мер

состояний нет необходимости учитывать параметры, связанные с моделированием неисправностей.

Чтобы понять, какие из перечисленных в табл.2.1-2.2 параметров

необходимо учитывать при построении оценочной функции, необходим тщательный анализ семантики строящихся последовательностей. После того как такие параметры отобраны, необходимо определить конкретный вид функционала $O(A_0, S)$.

Также в ряде случаев полезной бывает не численная, а структурная информация. Сюда можно отнести, например, списки состояний ЦУ, достигнутых в процессе работы того или иного метода. Причём такая информация может быть полезна не только для вычисления оценочной функции особей, но и для ускорения работы метода в целом.

Большое количество параметров порождает многокритериальность оценочной функции. При этом построение эффективной оценочной функции может быть затруднено, а учёт сразу всех необходимых параметров в ней может ухудшить сходимость ГА-метода [132]. Это также заставляет искать другие пути достижения цели в ГА.

В случае многокритериальности функция оценки часто строится по аддитивному принципу как взвешенная сумма некоторых из перечисленных параметров или некоторых функций (критериев) от них (т.н. наивные методы, [132], раздел 7.1):

$$O = c_1 \cdot f_1(параметр1) + c_2 \cdot f_2(параметр2) + \ldots, \qquad (2.5)$$

что по сути является скаляризацией оценочной функции.

В этом случае требуется обоснование численных значений констант c_i, что осуществляется, обычно, на основании машинных экспериментов.

Второй из возможных подходов заключается в том, что сначала достигается один из критериев качества последовательности, затем второй и т.д. Таким образом, в ГА выделяется несколько этапов работы, для каждого из которых строится своя оценочная функция $O_i()$, где i - номер этапа. Переход от этапа $i-1$ к этапу i осуществляется при выполнении критерия этапа $i-1$. При этом эволюционные операции на этапе с номером i, которые нарушают критерии предшествующих этапов $1,\ldots,i-1$ считаются запрещёнными. Отметим, что такой подход хоть и требует введения дополнительных проверок, но используется достаточно часто.

Для дальнейшего построения шаблона одноуровневого ГА-

метода будем считать, что для каждой особи-последовательности S известен метод вычисления её оценочной функции $O(A_0, S)$. Поскольку основной формой представления методов в данной работе является псевдокод, то будем считать, что вычисление оценки заданной последовательности реализовано процедурой *ОценитьОсобь(A_0, S)*.

ГА в целом представляет собой итеративное построение новых популяций потенциальных решений, которые выполняются либо до момента, когда найдено приемлемое решение, либо пока не выполняются некоторые условия остановки.

Укрупненный псевдокод шаблона одноуровневого ГА-метода построения ИдП может быть представлен в следующее виде.

Алгоритм А2.1

```
ГА_построения_ИдП(A0, Параметры)
{
  ПредварительнаяОбработка(A0);
  Pop_нач=ПостроениеНачальнойПопуляции(N_особ, L);
  ОценитьПопуляцию(Pop_нач, A0, N_особ);
  ВычислитьФитнессФункцию(Pop_нач, N_особ);
  Pop_тек=Pop_нач;
  НомерПопуляции=0;
  // основной цикл по поколениям
  while( НеДостигнутКритерийОстановки() )
  {
    ВычислитьФитнессФункцию(Pop_тек, N_особ);
    // цикл построения промежуточной популяции
    while( СтроитсяНоваяПопуляция () )
    {
      Родители=ОперацияСелекции();
      Потомки=ОперацияСкрещивания(Родители);
      Потомки=ОперацияМутации(Потомки);
      ДобавлениеВНовуюПопуляцию(Потомки);
    } // конец while – построение новой популяции
    ОценитьПопуляцию(Pop_тек, A0, N_особ);
    НомерПопуляции++;
```

```
    АдаптацияПараметров();
  } // конец while – достигнут критерий остановки
  СортироватьПопуляциюПоОценке(Pop_тек);
  // решение=лучшая особь в посл. популяции
  Решение=ТекущаяПопуляция[0];
} // конец одноуровневого ГА построения ИдП
```

Дадим пояснения к псевдокоду метода. Здесь использованы следующие переменные:

$Pop_{нач}$, $Pop_{тек}$ - начальная популяция и текущая популяция итерации соответственно;

L - начальная длина строящихся особей-последовательностей, показывающая число входных наборов;

Родители – особи, выбранные для генетических операций скрещивания и мутации;

Потомки – особи, являющаяся результатом генетических операций;

НомерПопуляции – счётчик итераций, показывает номер текущей популяции.

В коде функции имеют следующую нагрузку:

ПостроениеНачальнойПопуляции() – реализует стратегию построения начальной популяции особей;

ОценитьПопуляцию() – для всех особей в заданной популяции вычисляет оценочную функцию;

ОперацияСелекции(), ОперацияСкрещивания(), ОперацияМутации() – реализуют генетические операции селекции, скрещивания и мутации соответственно;

ДобавитьОсобьВПопуляцию() – добавляет особь в заданную популяцию;

ПредварительнаяОбработка() – путём обрыва обратных связей выполняет преобразование описания ЦУ в псевдокомбинационный эквивалент, который далее используется при моделировании, а также вычисляются статические параметры оценочной функции.

Работа метода в соответствии с псевдокодом начинается с построения начальной популяции $Pop_{нач}$, которая состоит из заданного числа особей $N_{особ}$. После этого вычисляется оценка каждой особи в данной популяции.

Далее открывается основной цикл ГА построения новых популя-

ций.

Вначале для всех особей в текущей популяции вычисляется их фитнесс-функция. Наиболее часто используются несколько следующих видов фитнесс-функций [127]:

- линейное ранжирование; после сортировки популяции по оценочной функции в убывающем порядке особь с номером i получает следующее численное значение фитнесс-функции:

$$fit(ind_i) = \frac{1}{N_{особ}}(a - (a-b)\frac{i-1}{N_{особ}-1}), \qquad (2.6)$$

где $1 \le a \le 2$ - выбирается случайно, $b = 2 - a$;

- линейное преобразование:

$$fit(ind_i) = a \cdot O(ind_i) + b, \qquad (2.7)$$

где a и b - константы;

- сигма отсечение:

$$fit(ind_i) = O(ind_i) + (\overline{O} - c \cdot \sigma), \qquad (2.8)$$

где $\overline{O}$ - средняя оценка поп популяции, c - малое натуральное число, σ - среднеквадратичное отклонение по популяции; если в результате вычисления (2.8) будет получено отрицательное значение, то фитнесс-функция полагается равной нулю.

После этого реализуется построение промежуточной популяции. Оно основано на применении эволюционных операций селекции, скрещивания и мутации. Для выполнения двух последних необходимо выбрать особей, называемых родителями. Это реализуется оператором оператор репродукции (селекции) $Sel : Pop_j \to ind_i$.

Наиболее часто используемые схемы репродукций [127]:

- колесо рулетки (пропорциональный отбор); особи отображаются в виде секторов рулетки, особь с большей фитнесс-функцией имеет больший сектор на колесе выбора и, следовательно, большую вероятность быть выбранной в качестве родителя;
- турнирный отбор; все особи популяции разбиваются на группы

(чаще 2-3 особи), среди которых выбирается лучшая.

Часто фитнесс-функция не вычисляется явно, а заложенный в ней принцип сравнения оценки данной особи с оценками других особей в популяции (принцип соревновательности) реализуется в операторе репродукции, т.е. происходит совмещение вычисления фитнесс-функции с оператором селекции. В этом случае функция $fit = fit(ind_i, Pop_j)$ из (2.1) не задаётся явно, а в реализации ГА-метода функции *ВычислитьФитнессФункцию()* и *ОперацияСелекции()* будут совмещены.

Построение новой популяции из текущей возможно различными способами: с использованием промежуточной популяции, либо без неё. Приведённый шаблон допускает применение обеих схем. При этом общим в данных схемах остаётся использование генетических операций селекции, скрещивания и мутации, которые определены ранее.

Параметры $p_{скр}$ и $p_{мут}$, определяющие вероятности применения эволюционных операций скрещивания и мутации соответственно, в процессе работы ГА также могут изменять своё значений с целью увеличения эффективности их применения. Такой процесс называется адаптацией параметров и в коде представлен одноимённой процедурой. Основная цель такого процесса заключается в подборе значений вероятностей $p_{скр}$ и $p_{мут}$, чтобы, с одной стороны, предотвратить преждевременную сходимость, а с другой стороны – избежать сходимости к локальным экстремумам.

Один из возможных методов адаптации заключается в фиксировании нескольких возможных значений параметров $p_{скр}$, $p_{мут}$ и выборе одного из них в зависимости от текущего этапа работы метода, либо от размеров популяции/особи. Однако такой подход не позволяет гибкое изменение значений вероятностей $p_{скр}$ и $p_{мут}$, хотя и применяется на практике, например, в [46].

Более гибким является подход, в котором значения $p_{скр}$ и $p_{мут}$ вычисляются на каждой итерации ГА в зависимости от текущего достигнутого состояния.

Эксперименты показывают, что разность между максимальным значением оценки особи и средним значением по популяции $O_{\max} - \overline{O}$

тем меньше, чем ближе особи находятся к некоторому оптимуму. При этом не известно: является ли данный оптимум локальным или глобальным? Поскольку задачей адаптации является возможность избегать локальных оптимумом, то при приближении к нему вероятности операций скрещивания и мутации $p_{скр}$ и $p_{мут}$ должны увеличиваться. Таким образом, вероятности $p_{скр}$ и $p_{мут}$ должны изменяться обратно пропорционально показателю $O_{\max}$-$\overline{O}$:

$$p_{скр} = k_1 \cdot \frac{1}{O_{\max} - \overline{O}},$$
$$p_{мут} = k_2 \cdot \frac{1}{O_{\max} - \overline{O}}. \qquad (2.9)$$

Такое задание вероятностей $p_{скр}$ и $p_{мут}$ позволяет определить их значения для текущей итерации ГА. Причём эти значения будут применяться сразу ко всем особям в популяции на текущем шаге.

Очевидно, что для особей с лучшей оценкой необходимо применять эволюционные операции с меньшей вероятностью, тогда как для особей с меньшей оценкой такая вероятность должна быть выше.

Для учёта данного положения введём в (2.9) параметр, который будет показывать отклонение оценки текущеё особи O_i от максимальной оценки по популяции $O_{\max}$. Тогда:

$$p_{скр} = k_1 \cdot \frac{O_{\max} - O_i}{O_{\max} - \overline{O}},$$
$$p_{мут} = k_2 \cdot \frac{O_{\max} - O_i}{O_{\max} - \overline{O}}. \qquad (2.10)$$

Из (2.10) легко видеть, что для лучших особей в популяции при $O_{\max} = O_i$ вероятности скрещивания и мутации будут равны $p_{скр} = p_{мут} = 0$, что позволяет ГА сохранять лучшие особи. Однако для особей с оценкой существенно ниже максимальной по популяции значения в (2.10) могут оказаться больше единицы, что противоречит смыслу ГА. В этом случае будем использовать фиксированные значе-

ния $p_{скр} = p'_{скр}$ и $p_{мут} = p'_{мут}$ для особей с низкой оценкой $O_i < \overline{O}$. Данный эвристический подход является достаточно общим [177-179] и будет использоваться в работе как основной при построении различного типа ИдП.

Итерации построения новых популяций прекращаются при выполнении одного из следующих условий:

- найдено точное решение задачи;
- достигнуто предельное число итераций;
- заданное число итераций не происходит улучшения оценки лучшей особи.

В псевдокоде А2.1 условие остановки метода проверяется в функции *НеДостигнутКритерийОстановки()*.

После остановки итераций решением задачи является последовательность-особь с лучшей оценкой в последней достигнутой популяции.

При таком построении шаблона ГА-методов все зависящие от реализации механизмы являются скрытыми в соответствующих процедурах и должны быть конкретизированы при реализации.

Для реализации конкретного ГА-метода построения ИдП на основе данного шаблона необходимо:

- исходя из поставленной цели задать оценочную функцию $O(A_0, S)$ для строящихся последовательностей-особей;
- для повышения эффективности метода экспериментально обосновать применение механизмов ГА, зависящих от реализации.

Далее на основании предложенного структурного шаблона (Алгоритм А2.1) будет разработан ряд методов построения ИдП, которые попадают в данный класс.

2.3. ГА-метод построения инициализирующих последовательностей синхронных последовательностных ЦУ

Инициализируемость (инициируемость) синхронных последовательностных ЦУ является одним из свойств, которое разработчик должен обеспечить при их проектировании. Обеспечение возможности перехода ЦУ в необходимое начальное состояние является обяза-

тельным как для корректной дальнейшей работы, так и для его тестирования [103]. В этом случае ЦУ должно обладать способностью при подаче некоторой последовательности перейти из начального полностью неопределённого состояния в некоторое определённое. Такое определение показывает, что инициализирующие последовательности (ИнП) являются частным случаем синхронизирующих. Задача данного рода возникает всякий раз, например, при инициализации работы ЦУ во время его включения.

Одним из вариантов решения является введение в аппаратурную часть ЦУ сигнала сброса, который соединён со всеми элементам состояния ЦУ. Подача сигнала выбранного логического уровня по линии сброса обеспечивает перевод триггеров ЦУ в нужное начальное состояние.

С другой стороны, если условия не позволяют модификацию аппаратной части, либо на размер аппаратуры введены явные ограничения, возможным подходом является построение входных последовательностей, решающих данную задачу.

Такая неформальная постановка показывает необходимость построения автоматизированного инструментария, который позволяет:

1) определить, существует ли для выбранного ЦУ инициализирующая последовательность?
2) построить инициализирующую последовательность в случае положительного ответа на предыдущий вопрос.

Для решения поставленных задач необходимо сначала произвести анализ существующих определений инициализируемости ЦУ.

Наиболее часто рассматриваются два типа инициализируемости [104-105]: функциональная инициализация и логическая инициализация.

Неформально понятие функциональной инициализируемости требует, чтобы при начале работы ЦУ из произвольного состояния и подаче на его вход заданной последовательности устройство должно перейти в некоторое заранее специфицированное состояние. При этом данное финальное состояние называется состоянием сброса.

Логическая инициализация требует, чтобы ЦУ, которое находится в полностью неопределённом состоянии, после подачи на его вход последовательности и моделировании поведения в трёхзначном алфавите E_3, перешло в некоторое полностью определённое состояние.

Дадим теперь формальные определения данных понятий. В качестве модели используем синхронные последовательностные ЦУ (рис.1.1).

Пусть используется моделирование работы синхронного последовательностного ЦУ A_0 в трёхзначном алфавите $E_3 = \{0,1,u\}$. В заданном ЦУ A_0: $N_{вх}$ - число внешних входов, $N_{вых}$ - число внешних выходов, $N_{эл}$ - число логических элементов, $N_{тр}$ - число элементов состояний (псевдовыходов/псевдовходов). Пусть Q - множество всевозможных состояний последовательностного ЦУ, включая неопределённые. Тогда $Q = \{0,1,u\}^{N_{тр}}$. Пусть Z - множество всех определённых состояний ЦУ: $Z = \{0,1\}^{N_{тр}}$. Легко видеть, что $Z \subset Q$. Множество всех возможных входных последовательностей S_i образуют множество Σ, причём элементы S_i кодируются элементами из E_3. Для нашей задачи $Z_u = (uu...u)$ - начальное полностью неопределённое состояние ЦУ, $Z_u \in Q$. Отображение $F : Q \times \Sigma \to Q$ показывает все достижимые состояния ЦУ, когда на его вход поступают последовательности $S_i \in \Sigma$ при работе в алфавите E_3. В автоматной терминологии: F есть функция перехода при работе в алфавите E_3.

Определение 2.1 Последовательность $S \in \Sigma$ называется функционально инициализирующей для заданного ЦУ A_0 с функцией перехода F, если $\exists Z_i \in Z$ такое, что для $\forall \alpha \in Q$ будет выполняться: $Z_i = F(\alpha, S)$. Тогда Z_i называется состоянием сброса.

Определение 2.2. Последовательность $S \in \Sigma$ называется логически инициализирующей для заданного ЦУ A_0 с функцией перехода F, если финальное состояние $Z_i = F(Z_u, S) \in Z$, т.е. в нём определены все элементы состояний.

Определение логической инициализируемости имеет следующие преимущества с точки зрения разработчика ЦУ:

1) используется моделирование ЦУ в трёхзначном алфавите E_3, которое является наиболее широко распространённым на практике; процедуры его реализации присутствуют во всех известных САПР ЦУ;
2) поскольку трёхзначное моделирование использует в дополнение к двухзначному символ неопределённости $u \in E_3$, то оно, вообще говоря, является приблизительным, поэтому понятие логической инициализации является более строгим по отношению к функциональной; это заключается в том, что может быть показано следующее утверждение: функционально инициализируемое ЦУ может не быть логически инициализируемым [105].

Основываясь на этом, большинство исследователей на практике используют именно понятие логической инициализируемости. Мы также выберем данное определение в качестве основного для реализации метода.

Поскольку логически инициализирующие последовательности являются частным случаем характеристических последовательностей, то, вообще говоря, к решению задачи построения ИнП могут быть применены любые подходы построения характеристических последовательностей, например на основе преобразования булевых выражений. Однако весь класс характеристических последовательностей гораздо шире [21] и применение общих методов к решению данной конкретной задачи может оказаться неэффективным. Более того, если использовать методы построения характеристических последовательностей, то новый метод будет обладать всеми присущим им недостатками (раздел 1), главным из которых является невозможность обработки ЦУ большой размерности. Это связано с необходимостью обработки очень больших выражений для булевых функций, что влечёт за собой требование чрезмерных ресурсов памяти.

ГА позволяет обходить данное ограничение, поскольку не производит символьных преобразований, а для проверки качества ИнП использует моделирование поведения ЦУ.

Исходя из анализа выше, можно сделать вывод, что ГА-метод построения логических ИнП может быть сведён к построению таких входных последовательностей, которые, по возможности, устраняют наибольшее число неопределённостей в начальном состоянии ЦУ $Z_u = (uu...u)$. Чем большее число неопределённостей снято для задан-

ной последовательности, тем её качество считается более высоким. Т.е. оценку качества последовательности необходимо построить на основании определения 2.2.

Следовательно, качество последовательности можно формально выразить заранее до запуска алгоритма и для конструирования ГА-метода построения ИнП будем использовать одноуровневую схему (рис.2.1).

Зададим компоненты такого ГА-ориентированного метода.

Поскольку пространство поиска Σ состоит из всех возможных входных последовательностей, то в качестве особи выбираем одиночную последовательность $ind_i = S_i \in \Sigma$. При этом заранее длина S_i не известна. Кодирование особи (рис.2.3) заключается в представлении её в виде таблицы.

В качестве популяции выступает набор особей (входных последовательностей): $Pop = \{S_1, S_2, ..., S_n\}$, где $n = N_{особ}$ - размерность популяции. В описываемом подходе число особей в популяции n (её размер) является величиной неизменной.

Применяемые генетические операции Sel, $Cross$, Mut соответствуют описанным для шаблона ГА-метода – раздел 2.2.

Зададим оценочную функцию $O(A_0, S)$ особи-последовательности S. Именно построение оценочной функции является ключевым при разработке метода.

Вид оценочной функции для некоторой особи-последовательности S будем строить исходя из семантики определения инициализирующих последовательностей - определение 2.2. Для этого введём следующие эвристические параметры:

1) n_1 - отношение числа элементов состояний с определённым значением сигнала из $E_2 = \{0,1\}$ к их общему числу элементов состояний N_{mp}; очевидно, что чем больше элементов состояния устройства A_0 получило определённые значения на линиях, тем выше качество заданной последовательности S;
2) n_2 - число событий при моделировании поведения или активность ЦУ; чем выше число событий при моделировании ЦУ A_0 на заданной последовательности S, тем выше вероятность, что определённые значения сигналов из $E_2 = \{0,1\}$ достигнут элементов состояний ЦУ;

3) n_3 - длина входной последовательности-особи S; поскольку длина результирующего решения также важна, то для двух заданных особей с одинаковой оценкой параметров n_1 и n_2 следует выбирать ту последовательность, длина которой меньше.

Видно, что параметр n_1 прямо соответствует определению 2.2, тогда как параметры n_2 и n_3 являются эвристическими.

Для формального задания параметров n_1 и n_2 введём понятие функции установки значений компонент ЦУ в ноль или единицу.

Пусть функция $g(X_j, A_0)$ показывает значение на выходе компонента ЦУ g при завершении моделирования ЦУ A_0 на входном наборе $X_j = S[j]$ последовательности S.

Определение 2.3. Функция установки в ноль $U^0(g, X_j, A_0)$ равна 1, если после моделирования устройства A_0 на входном наборе X_j выход элемента g равен нулю:

$$U^0(g, X_j, A_0) = 1 \Leftrightarrow g(X_j, A_0) = 0. \qquad (2.11)$$

Определение 2.4. Функция установки в единицу $U^1(g, X_j, A_0)$ равна 1, если после моделирования устройства A_0 на входном наборе X_j выход элемента g равен единице:

$$U^1(g, X_j, A_0) = 1 \Leftrightarrow g(X_j, A_0) = 1. \qquad (2.12)$$

Тогда установка фиксированного значения 0 или 1 вместо значения $u \in E_3$ на выходе некоторого элемента g определяется выражением:

$$U^1(g, X_j, A_0) \vee U^0(g, X_j, A_0). \qquad (2.13)$$

Однако ценность того факта, что данный элемент g получил оп-

ределённое значение на выходе зависит от того насколько легко данный элемент управляется, что численно задаётся параметром управляемости H_g. Он должен быть учтён в виде веса для (2.13). Тогда важность того, что элемент g получил определённое значение на выходе определится выражением:

$$H_g \cdot (U^1(g, X_j, A_0) \vee U^0(g, X_j, A_0)). \qquad (2.14)$$

Далее без потери общности будем использовать выражение (2.13), считая, что если доступны сведения об управляемости элементов ЦУ, то они могут быть учтены в виде (2.14).

Тогда для эвристического параметра n_1 можно записать:

$$n_1 = \frac{\sum_{g \in Z} (U^1(g, X_j, A_0) \vee U^0(g, X_j, A_0))}{N_{mp}}, \qquad (2.15)$$

где Z - множество линий элементов состояний ЦУ.

Параметр числа событий $N_{соб}$ (активности) ЦУ A_0 при моделировании на входном наборе X_j также может быть выражен через функции установки значений:

$$N_{соб}(A_0, X_j) = \sum_{g \in G} ((U^1(g, X_{j-1}, A_0) = U^0(g, X_j, A_0)) \vee \\ \vee (U^0(g, X_{j-1}, A_0) = U^1(g, X_j, A_0))), \qquad (2.16)$$

где G - множество логических элементов ЦУ.

Тогда число событий $N_{соб}$ для ЦУ A_0 при моделировании на входной последовательности S определяется:

$$N_{соб}(A_0, S) = \sum_{i=1}^{длина(S)} N_{соб}(A_0, X_j). \qquad (2.17)$$

Именно значение (2.17) определяет параметр n_2:

$$n_2 = n_2(A_0, S) = N_{соб}(A_0, S). \qquad (2.18)$$

Вычисление параметров n_1 и n_2 производится по результатам моделирования поведения ЦУ A_0 на заданной последовательности-особи S.

Параметр n_3 зависит только от последовательности: $n_3 = n_3(S)$.

Таким образом, формально оценочная функция в целом будет зависеть от параметров A_0 и S: $O = O(A_0, S)$.

С учётом весовых коэффициентов можно записать:

$$\begin{aligned} O(A_0, S) &= O(n_1(A_0, S), n_2(A_0, S), n_3(S)) \\ &= (c_1 * n_1(A_0, S) + c_2 * n_2(A_0, S)) * {c_3}^{n_3(S)}, \end{aligned} \qquad (2.19)$$

где n_1, n_2 и n_3 – описанные выше параметры; c_1, c_2, c_3 – нормализующие константы.

Видно, что оценочная функция последовательности S является многокритериальной. Для параметров n_1 и n_2 она строится по аддитивному принципу. Параметр n_3 входит в функцию $O()$ в виде показателя степени.

Важность каждого из критериев в общей оценке определяется нормирующими константами c_1 и c_2, которые показывают вес параметров инициализации триггеров и активности элементов в устройстве. Очевидно, что первый из них $n_1(A_0, S)$ является более ценным в смысле решения задачи, тогда как на параметр $n_2(A_0, S)$ следует ориентироваться в том случае, когда инициализация триггеров приостановилась.

Коэффициент ${c_3}^{n_3(S)}$ необходим для того, чтобы более высокую оценку получали более короткие последовательности S. Это будет достигаться при выборе $0 < c_3 < 1$. Следует отметить, что при $c_3 << 1$ глубина поиска существенно увеличится, существенно замедляя поиск. Поэтому константу c_3 следует выбирать очень близкой к 1.

В таком же виде (2.19) оценочная функция определена, например,

в алгоритме симуляции отжига [133, 130], где решение строится на основании эволюции одного потенциального решения.

Для определённой в виде (2.19) функции оценки можно записать алгоритм её вычисления. Псевдокод такой процедуры вычисления оценочной функции одной особи-последовательности приведён ниже.

Алгоритм А2.2

```
ОценитьОсобь(A0, S)
{
  ИнициализацияНеопределённогоСостояния(A0);
  SV=МоделированиеРаботыИсправногоЦУ(A0, S);
  n1=ЧислоАктивированыхТриггеров(SV);
  n2=АктивностьСхемыПриМоделировании(SV);
  n3=Длина(S);
  // формула 2.19
  return Оценка=(c1*n1+c2*n2)*c3^n3;
}
```

В данном коде переменная SV является массивом, который содержит значения сигналов на всех линиях моделируемого ЦУ. Элементы этого массива дают определяют функции $g(X_j, A_0)$, а на их основании вычисляется (2.19).

Ключевым при вычислении оценок особей является использование готовых процедур моделирования работы исправных ЦУ, что существенно упрощает алгоритмическую реализацию метода.

После задания оценочной функции, путём уточнения шаблона одноуровневого ГА строится метод генерации инициализирующих последовательностей ЦУ. Его алгоритмическая реализация в виде псевдокода приведён ниже.

Алгоритм А2.3

```
ГА_ЛогическаяИнициализация(A0, Параметры)
{
  Pop_нач=ПостроениеНачальнойПопуляции(N_особ, L);
  ОценитьПопуляцию(Pop_нач, A0, N_особ);
  ВычислитьФитнессФункцию(Pop_нач, N_особ);
```

```
  Pop_тек = Pop_нач ;
  НомерПопуляции=0;
  while( НеДостигнутКритерийОстановки() )
  {
    for( int i=0 ; i<N_особ ; i++ )
    {
      Позиция=0;
      РодительА=ВыбратьРодителя(Pop_тек);
      РодительВ=ВыбратьРодителя(Pop_тек);
      if( rand() > p_скр )
       Потомок=ВыполнитьСкрещивание(РодительА,РодительБ);
      else
        Потомок=РодительА;
      if( rand() > p_мут )
        Потомок=ВыполнитьМутацию(Потомок);
      ДобавитьОсобьВПопуляцию(Pop_пром,Потомок,Позиция);
      Позиция++;
    } // конец for – построение промежуточной популяции
    ОценитьПопуляцию(Pop_пром, A_0, N_особ);
    Pop_тек=ПостроитьНовуюПопуляцию(Pop_тек, Pop_пром, N_особ);
    ВычислитьФитнессФункцию(Pop_тек, N_особ);
    НомерПопуляции++;
  } // конец while – достигнут критерий остановки
  СортироватьПопуляциюПоФитнесс(Pop_тек);
  // решение=лучшая особь в посл. популяции
  Решение=Pop_тек[0];
} // конец одноуровневого ГА построения ИдП
```

Данный псевдокод является уточнением шаблона одноуровневых методов А2.1, поэтому и смысловая нагрузка основных переменных в коде соответствует таковой в шаблоне. В данном методе использована схема ГА с совмещением циклов построения промежуточной и новой популяций.

В качестве аргументов алгоритм получает описание исследуемого ЦУ A_0, которое далее будет использоваться при оценке качества особей, и структуру с параметрами ГА.

Переменная *Позиция* показывает место в промежуточной популяции, в которое добавляется особь, построенная с помощью генети-

ческих операций. Построение промежуточной популяции происходит до заполнения всех позиций, а общее число особей в ней равно числу особей в основной популяции.

В приведённом методе требует уточнения процедура *ОценитьПопуляцию()*. В ней вычисляется оценочная функция каждой особи популяции в виде (2.19). Псевдокод реализации данной функции приведён ниже.

Алгоритм А2.4

```
ОценитьПопуляцию( Pop , A0 , Nособ )
{
  for( int i=0 ; i<Nособ ; i++ )
  {
    S = Pop[i];
    O[i]=ОценитьОсобь( A0 , S );// Алгоритм А2.2
  } // конец цикла по особям в популяции
}   // конец процедуры оценки популяции
```

Построение начальной популяции выполняется псевдослучайным методом, поскольку к настоящему времени не предложено ни одной эффективной эвристики для решения данной задачи. Псевдокод данной процедуры приведён ниже.

Алгоритм А2.5

```
ПостроениеНачальнойПопуляции( Nособ , L )
{
  Pop =ВыделениеПамяти( Nособ );
  for( int i=0 ; i<Nособ ; i++ )
  {
    Pop[i]=ПостроитьСлучайнуюПоследовательность( L );
  } // конец цикла по особям в популяции
  return Pop
}   // конец построения начальной популяции
```

Здесь переменная L показывает начальную длину особей-последовательностей.

Поскольку ГА являются эвристическими алгоритмами, большое значение приобретает этап реализации метода в виде алгоритма и проведение экспериментов по выбору значений эвристических констант или зависящих от реализации компонент.

Апробация данного метода производится в главе 5, раздел 4. Там же приводятся численные результаты машинных экспериментов (табл.5.1), позволяющие сделать выводы об его эффективности.

Основными преимуществами данного метода являются следующие:

- высокая эффективность алгоритма в терминах времени генерации и числа инициализированных элементов состояний;
- возможность работы с большими ЦУ;
- гибкость в настройке соотношения глубина поиска/время работы за счёт выбора соответствующих параметров для условий окончания работы.

С другой стороны данный подход не является точным в следующих смыслах:

- метод может показать неинициализируемость ЦУ (либо некоторых элементов состояния) даже в том случае, когда существует ИнП;
- метод может найти ИнП не наименьшей длины.

В такой реализации метод описан в [134-135].

2.4. ГА-метод построения последовательностей достижения состояния ЦУ

Одной из близких по постановке к рассмотренной выше является задача построения последовательностей достижения заданного состояния ЦУ (ПДС). Для автоматного представления ЦУ чаще используется термин установочная последовательность (УП). Данная задача также часто возникает в задачах диагностирования ЦУ.

Впервые задача построения УП сформулирована при построении диагностических экспериментов с автоматами [26]. В настоящее время развивается направление, в котором автоматные методы диагностирования ЦУ переносятся на структурный уровень [100-101], что также требует решения данной задачи для структурного уровня представления ЦУ. Такая адаптация методов становится возможной в свя-

зи с повышением вычислительной мощности рабочих станций, которое позволяет производить более глубокий поиск.

Определение 2.5 Для ЦУ A_0 с функцией перехода F последовательность $S \in \Sigma$ называется установочной в состояние $Z_{кон}$, если для $\forall \alpha \in Q$ будет выполняться: $Z_{кон} = F(\alpha, S)$.

Такое задание установочной последовательности является естественным для структурного уровня представления ЦУ при использовании трёхзначного алфавита моделирования E_3. Для автоматного задания ЦУ данное определение говорит, что начальным состоянием может являться любое состояние автомата A_0.

Возможна постановка задачи, когда задано как конечное $Z_{кон}$, так и начальное $Z_{нач}$ состояние ЦУ, либо даже множество начальных состояний $Q_{старт}$.

Определение 2.6 Для ЦУ A_0 с функцией перехода F последовательность $S \in \Sigma$ называется установочной в состояние $Z_{кон}$, если для $\forall \alpha \in Q_{старт} \subset Q$ будет выполняться: $Z_{кон} = F(\alpha, S)$.

Данное определение ближе к традиционному гиловскому определению установочной последовательности [26], поскольку в последнем задаётся именно множество возможных начальных состояний автомата.

От задачи инициализируемости данная задача отличается двумя моментами.

1) Конечное состояние $Z_{кон}$ устройства A_0 чётко специфицировано. Например, если положить $Q_{старт} = Q$ и $Z_{кон} \in Z$ задано, то получим задачу функциональной инициализируемости.
2) Возможно, задаётся начальное состояние $Z_{нач}$, из которого стартует устройство A_0. В практических задачах начальное состояние часто полностью определено, что является частным случаем при

$\left|Q_{старт}\right| = 1$.

3) В задаче построения ИнП речь идёт именно о стартовом режиме работы ЦУ, тогда как ПДС являются вспомогательными в различных алгоритмах диагностирования ЦУ.

При наличии сведений о графе переходов КА реализуемого ЦУ задача построения установочной последовательности решается с помощью диагностических экспериментов различного типа [26, 28]. Однако, как было отмечено ранее, для структурного уровня задания ЦУ данная информация, обычно, не известна.

Как и при решении задачи построения ИнП, на структурном уровне задача построения ПДС может быть решена методами, основанными на моделировании.

В [2, с.390] вводится понятие R- и S-последовательностей для элементов состояний синхронных последовательностных ЦУ.

Определение 2.7 Последовательность называется сбрасывающей R_i, если после её приложения ко входам ЦУ A_0, находящемуся в полностью неопределённом состоянии $Z_u = (uu...u)$, i-й элемент состояния $z_i \in Z$ получит значение 0 в алфавите моделирования E_3.

Определение 2.8 Последовательность называется устанавливающей S_i, если после её приложения ко входам ЦУ A, находящемуся в полностью неопределённом состоянии $Z_u = (uu...u)$, i-й элемент состояния $z_i \in Z$ получит значение 1 в алфавите моделирования E_3.

Данные R- и S-последовательности называются последовательностями типа A. Если же потребовать, что сброс (установление) значения на i-м элементе состояния достигается для ЦУ, в котором некоторые начальные элементы состояний уже фиксированы (имеют значение 0 или 1) и не изменяются, то такие последовательности называются R- и S-последовательностями типа B.

В [101] предложен генетический алгоритм построения R- и S-последовательностей. В той же работе они используются в ГА построения D-последовательностей различающих пару состояний пу-

тём их добавления в начальную популяцию, что позволяет ускорить сходимость метода.

Данные последовательности могут быть использованы для решения задачи достижимости состояний. При этом задача решается итеративным построением R- и S-последовательностей типа B для каждого элемента $z_i \in Z$ состояния ЦУ A_0, $i = \overline{1,m}$. Отметим, что в такой постановке построение R- и S-последовательностей для элемента состояния z_i должно оставлять неизменными все остальные элементы состояний $z_j \in Z$, $j = 1,..,i-1,i+1,m$.

Псевдокод алгоритмической реализации метода для задачи с заданным начальным состоянием приведён ниже.

Алгоритм А2.6

```
ГА_Достижение_Состояния(A0, Zнач, Zкон)
{
  Последовательность=∅;
  Инициализация_ЦУ(A0, Zнач);
  m=ЧислоЭлементовСостояний(A0);
  for( i=0 ; i<m ; i++ )
  {
    zi = Zкон[i];
    if( zi =1 )
      S=Построение_S_последовательности(A0, i);
    else
      if( zi =0 )
        S=Построение_R_последовательности(A0, i);
      иначе
        S = ∅;
    Последовательность=Последовательность∪S;
  }
  return Последовательность;
}
```

В данном псевдокоде переменная *Последовательность* показывает результирующую установочную последовательность.

Данный подход имеет очевидные преимущества. Первым из них является прозрачность. Второе преимущество заключается в том, что если заранее построено множество возможных R_i- и S_i-последовательностей $(i = 1,...,m; m = N_{mp} = |Z|)$ для данного ЦУ A_0 (что может быть обеспечено разработчиком ЦУ), то построение ПДС является тривиальной задачей.

Очевидными недостатком такого метода решения задачи являются как итеративность вызова алгоритма метода построения R_i- и S_i-последовательностей, так и то, что для произвольного элемента состояния z_i не известно и не гарантировано существование такой последовательности, которая не изменяет остальные элементы состояний $z_j \in Z$, $j = 1,..,i-1,i+1,m$. В последнем случае алгоритм не применим полностью.

Отметим также, что в методе не акцентируется внимание на способе построения R- и S-последовательностей, который также может иметь достаточно большую сложность.

Указанные недостатки метода с применением итеративного построения R- и S-последовательностей заставляют разработать новый метод.

Покажем конструктивное построение ГА-метода решения задачи генерации последовательностей достижения состояний. Решение строится на основании одноуровневой схемы применения шаблона метода (раздел 2.2).

Для построения ГА-метода на основании общей структуры алгоритма А2.1 зададим его компоненты. Кодирование особей и популяции, а также применяемые эволюционные операции соответствуют шаблону и ГА-методу построения инициализирующих последовательностей.

Выполним анализ и построение оценочной функции с использованием функций установки U^0 и U^1. Семантически в нашей задаче оценочная функция показывает: насколько близко текущее состояние $Z_{тек}$ моделируемого ЦУ A_0, получаемое после подачи последовательности S, находится от требуемого состояния $Z_{кон}$.

Поскольку мы рассматриваем наиболее общий случай, когда граф переходов ЦУ A_0 не известен, то для формализации оценки будем

использовать расстояние по Хэммингу для двоичного представления $Z_{тек}$ и $Z_{кон}$, показывающее число совпадений элементов состояний. Таким образом, для заданного ЦУ A_0 и заданной входной последовательности S оценка вычисляется на основании результатов исправного моделирования: $Z_{тек} = F(Z_{нач}, S)$. Тогда можно записать:

$$O(A_0, S, Z_{нач}, Z_{кон}) = O(Z_{тек}(A_0, S, Z_{нач}), Z_{кон}) = \\ = \sum_{Z_{тек}[i]=Z_{кон}[i]} 1, \tag{2.20}$$

где $Z[i]$ показывает значение i-го элемента состояния, $i = 1,..., m$.

Выражение для функции оценки (2.20) в данном случае можно также представить через функции достижения значений, введённые в данном разделе ранее. Поскольку $Z_{тек}$ определяется как достижение определённого значения 0 или 1 для элементов $g \in Z$, то значения компонент $g_{кон}$ определяют состояние $Z_{кон}$, а последнее состояние ЦУ задаётся при $j = длина(S) - 1$, то с учётом (2.11)-(2.12) можно записать:

$$O(A_0, S, Z_{нач}, Z_{кон}) = \sum_{g \in Z} ((U^0(g, X_j, A_0, Z_{нач}) = g_{кон}) \vee \\ (U^1(g, X_j, A_0, Z_{нач}) = g_{кон})), \tag{2.21}$$

где: Z - множество элементов состояний ЦУ. Видно, что в (2.21) была включёна только одна компонента из (2.19), показывающая совпадение на множестве элементов состояний Z. По аналогии с (2.19) данную оценку можно обобщить путём добавления компонент, которые показывают активность ЦУ A_0, а также длину особипоследовательности S, что должно улучшить качество оценки.

Также в оценку (2.21) может быть включён параметр управляемости элементов ЦУ, если он доступен разработчику.

Реализация вычисления функции в виде (2.21) представлена ниже.

Алгоритм А2.7

ОценитьОсобь(A_0, S, $Z_{нач}$, $Z_{кон}$)

{

  ИнициализацияНачальногоСостояния(A_0, $Z_{нач}$);

  SV =МоделированиеРаботыИсправногоЦУ(A_0, S);

  $Z_{тек}$=ОпределитьДостигнутоеСостояние(SV);

  // сравнить достигнутое состояние с требуемым

  // формула 2.20

  return Оценка=$O(A_0, Z_{тек}, Z_{кон})$;

}

Состояние $Z_{нач}$ необходимо для инициализации ЦУ A_0 всякий раз перед моделированием, которое выполняется для текущей особи-последовательности S с целью её оценки. Состояние $Z_{тек}$ является результатом такого моделирования и определяется из массива SV.

Близость рассматриваемого метода решения задачи и описанного в разделе 2.3, а также и построение оценочной функции на основании моделирования поведения одного исправного ЦУ позволяют реализовать последний метод на основании псевдокода алгоритма А2.3 путём внесения минимальных изменений.

Для этого необходимо изменить критерий остановки ГА. В методе достижения состояния завершение работы происходит при совпадении текущего состояния $Z_{тек}$ и финального $Z_{кон}$. При этом численно оценка 2.21 будет равняться $O(A_0, S, Z_{нач}, Z_{кон}) = m$, где $m = N_{тр}$ - число элементов состояния A_0.

Таким образом, ГА-метод построения ПДС включает в себя алгоритмы А2.3 – основной цикл ГА, А2.4 – оценка популяции особей, А2.5 – построение начальной популяции и А2.7 - вычисление оценки последовательности-особи. Также описание данного метода приведено в [136].

Построение ГА-метода традиционно завершается «настройкой» параметров, которая заключается в экспериментальном обосновании выбора конкретных эволюционных операторов (скрещивания, мутации, построения новой популяции), а также основных параметров (вероятности скрещивания и мутации $p_{скр}$ и $p_{мут}$, предельных зна-

чений числа общих итераций и итераций без улучшения решений и т.д.). Например, в качестве верхней оценки длины строящихся последовательностей выбирается величина $4 \cdot d$, где d - структурная последовательностная глубина ЦУ [2].

2.5. ГА-метод верификации эквивалентности поведения двух заданных ЦУ

При проектировании ЦУ с помощью САПР возникает задача проверки эквивалентности их поведения. Задача может возникать, например, в связи с оптимизацией ЦУ либо его фрагмента. Цель таких преобразований может быть различна: уменьшение площади для физической реализации, уменьшение рассеивания тепла на кристалле СБИС, либо его ограниченной территории и т.п. При этом такие оптимизирующие преобразования, обычно, затрагивают только комбинационную часть ЦУ, тогда как последовательностная структура остаётся неизменной. Вероятность ошибок при таких преобразованиях отлична от нуля, поскольку размеры обрабатываемых ЦУ включают десятки и тысячи логических вентилей. Для проектировщика важно понимание: произошёл ли этап оптимизации с ошибкой или нет?

При рассмотрении задачи эквивалентности поведения двух заданных ЦУ на практике используется ещё большее число определений эквивалентности в сравнении с задачей инициализации. Они отражают специфику, с которой работает конкретный разработчик. Для понимания проблемы и выбора конкретного определения для дальнейшей работы необходимо рассмотреть различные подходы к задаче определения эквивалентности ЦУ.

Наиболее часто задача верификации эквивалентности ставится для ЦУ, которые заданы детерминированными конечными автоматами (раздел 1). Пусть заданы два ЦУ: $A_1 = \{Z_1, X_1, Y_1, \delta_1, \lambda_1\}$ и $A_2 = \{Z_2, X_2, Y_2, \delta_2, \lambda_2\}$.

Определение 2.9. Пусть $z_1 \in Z_1$ - некоторое состояние в автомате A_1 и $z_2 \in Z_2$ состояние в автомате A_2 соответственно. Состояния z_1 и z_2 называются эквивалентными $z_1 \sim z_2$,

если не существует входной последовательности, различающей пару состояний z_1 и z_2:

$$\forall S \in X^* : \lambda_1(z_1, S) = \lambda_2(z_2, S), \qquad (2.22)$$

где λ_1 и λ_2 функции выходов автоматов A_1 и A_2 соответственно.

Определение 2.10. Пусть заданы автоматы A_1 и A_2, которые имеют цепи сброса состояний, причём z_1 и z_2 - состояния, в которые переходят автоматы A_1 и A_2 после подачи сигнала сброса соответственно. Тогда автоматы A_1 и A_2 будем называть эквивалентными по сбросу тогда и только тогда, когда соответствующие состояния z_1 и z_2 являются эквивалентными.

Для удобства наличие аппаратных цепей сброса в автоматах A_1 и A_2 можно рассматривать как приложение некоторой синхронизирующей (инициализирующей) последовательности.

В [112] даётся следующее определение эквивалентности.

Определение 2.11. Пара состояний z_1 и z_2 называется выравнивающей, если существует входная последовательность S, которая переводит два заданных ЦУ A_1 и A_2 в пару эквивалентных состояний $z_1 \sim z_2$, т.е.:

$$\delta_1(z_1, S) \sim \delta_2(z_2, S), \qquad (2.23)$$

где δ_1 и δ_2 - функции переходов ЦУ A_1 и A_2 соответственно.

Заметим, что выравнивающая последовательность в смысле определения 2.11 является частным случаем синхронизирующей последовательности.

Определение 2.12. Два заданных автомата A_1 и A_2 называются последовательностно аппаратно эквивалентными (состоят в

отношении ПАЭ) тогда и только тогда, когда существует входная последовательность S, которая является выравнивающей для любой пары состояний z_1 и z_2:

$$\exists S : \forall (z_1, z_2) \Rightarrow \delta_1(z_1, S) \sim \delta_2(z_2, S). \qquad (2.24)$$

Смысл данного определения заключается в том, что если два ЦУ A_1 и A_2 имеют универсальную синхронизирующую последовательность, т.е. состоят в отношении ПАЭ, то их дальнейшей постсинхронное поведение будет одинаковым. С использованием данного определения можно показать, что если два заданных ЦУ состоят в отношении ПАЭ, то любая синхронизирующая последовательность для них является также универсальной выравнивающей. Следовательно, если для каждого из ЦУ задана своя синхронизирующая последовательность, то проверка свойства ПАЭ сводится к проверке эквивалентности по сбросу в соответствии с определением 2.10.

Однако подход с определением отношения ПАЭ не может быть применён для дизайнов больших современных ЦУ, в которых часто свойство эквивалентности проверятся не для дизайна в целом, а для его частей после декомпозиции. При этом, обычно, не отслеживаются выходные реакции подсхем в фазе инициализации. На первый взгляд может показаться, что это даёт больше свободы в оптимизации завершённых фрагментов. Однако синхронизирующая последовательность объединенного дизайна при этом может быть разрушена, поскольку функции переходов подавтоматов могут измениться.

Развитием подхода ПАЭ, который позволяет декомпозицию ЦУ, является понятие неизменяющей замены (НЗ) [102]. Смысл подхода состоит в том, чтобы вход-выходное поведение оптимизированного ЦУ было подмножеством вход-выходного поведения изначально заданного ЦУ. Это свойство позволит окружению фрагмента дизайна, который подвергся оптимизации, не заметить различий в его поведении в сравнении с оригинальным фрагментом.

Определение 2.13. ЦУ A_2 называется неизменяющей заменой для автомата ЦУ A_1, если для данного состояния z_2 в A_2 и любой конечной входной последовательности $S \in X^*$

существует состояние z_1 в ЦУ A_1 такое, что выходные поведения при старте из z_1 и z_2 соответственно совпадают, т.е.:

$$\lambda(z_1, S) = \lambda(z_2, S), \qquad (2.25)$$

где λ_1 и λ_2 функции выходов ЦУ A_1 и A_2 соответственно.

Свойство НЗ очень удобно для разработчика, однако его проверка является крайне сложной задачей. Это связано с тем, что в процессе такой проверки необходимо произвести перенумерацию всех внутренних состояний ЦУ, что при их большом размере является крайне сложным.

Попытка упростить проверку свойства НЗ привели исследователей к понятию трёхзначной неизменяющей замены (ТНЗ). Упрощение связано с тем, что моделирование поведения ЦУ выполняется в трёхзначном алфавите E_3.

Определение 2.14. Значение сигнала v_1 называется покрывающим для значения сигнала v_2 , если пара значений (v_1, v_2) принимает только следующие значения: $(0,0)$, $(1,1)$, (u,u), $(u,0)$ и $(u,1)$, где $1,0,u \in E_3$.

Видно, что для пары значений (v_1, v_2) запрещены следующие комбинации: $(0,1)$, $(1,0)$, $(0,u)$ и $(1,u)$.

Определение аналогичное 2.14 даётся для векторов значений.

Определение 2.15. Вектор значений V_1 называется покрывающим для вектора значений V_2, если каждый бит в векторе V_1 является покрывающим для соответствующего бита в векторе V_2 в смысле определения 2.14.

Трёхначным состоянием ЦУ называется вектор состояний, в котором каждый разряд принимает значение 0, 1 или u из E_3. Обозначим $z_u = (uu...u)$ полностью неопределенное состояние, когда все

разряды принимают неопределённое значение u.

Если z - состояние, из которого начинает работу автомат A_1, на вход которого подаётся последовательность S, то его выходные реакции обозначим $A(z,S)$. Для ЦУ, стартующего из полностью неопределённого состояния z_u, выходные реакции обозначаются $A(z_u,S)$.

Определение 2.16. ЦУ (автомат) A_1 называется трёхзначной неизменяющей заменой для ЦУ (автомата) A_2 тогда и только тогда, когда для произвольной последовательности S выходные реакции $A_1(z_u,S)$ покрывают выходные реакции $A_2(z_u,S)$.

Также говорят, что ЦУ A_1 и A_2 состоят в отношении трёхзначной неизменяющей замены.

Видно, что определение эквивалентности специально сужается для таких ЦУ, которые начинают работу в полностью неопределённом состоянии z_u, что является очень распространённым случаем на практике.

Поясним данное определение. Пусть A_1 и A_2 исходное и преобразованное (оптимизированное) ЦУ соответственно. Пусть на их входы подаётся последовательность S (в такой входной последовательности некоторые значения могут не иметь фиксированные значения 0 или 1, т.е. быть равными $u \in E_3$). Если некоторый выход исходного ЦУ A_1 принимает определённое значение 0 или $1 \in E_3$, но не значение $u \in E_3$, то соответствующий выход преобразованного ЦУ A_2 обязан принимать такое же значение. Если же выход ЦУ A_1 принимает неопределённое значение $u \in E_3$, то соответствующий выход ЦУ A_2 может принимать как значение u, так и уточняющие его значения 0 или $1 \in E_3$ без потери свойства трёхзначной неизменяющей замены.

Для свойства ТНЗ двух ЦУ A_1 и A_2 может быть показано, что если последовательность S является инициализирующеё для ЦУ A_1, то она также является инициализирующей для ЦУ A_2. Однако это выходит за рамки данного исследования.

Определение 2.17. ЦУ A_1 является трёхзначным эквивалентом (ТЭ) для ЦУ A_2 тогда и только тогда, когда при их при старте из неопределённого состояния x для любой входной последовательности S пары выходных реакций на неё $(A_1(z_u, S), A_2(z_u, S))$ принадлежат множеству $\{(0,0),(1,1),(u,u)\}$.

Таким образом определяется, что два ЦУ A_1 и A_2 состоят в отношении ТЭ при использовании алфавита моделирования E_3 тогда и только тогда, когда их выходные реакции полностью совпадают, при этом оба ЦУ начинают работу из полностью неопределённого состояния $z_u = (uu...u)$. При этом пары выходных значений не должны быть равны: $(0,1)$, $(1,0)$, $(0,u)$, $(1,u)$, $(u,0)$, $(u,1)$. Данное определение является очевидным расширением обычного определения эквивалентности для алфавита моделирования E_3.

Свойства ТНЗ и ТЭ являются компромиссом между точностью моделирования в многозначных логиках, которая является приемлемой с практической точки зрения, и возможностью их проверки. С точки зрения разработчиков свойства эквивалентности, определяемые в трёхзначных логиках, обладают следующими преимуществами.

1. Свойства ТНЗ и ТЭ легче проверять путём моделирования работы ЦУ, чем формальным доказательством эквивалентности. Во первых, для формального доказательства эквивалентности двух заданных ЦУ в смысле определений 2.4 или 2.6 необходимо рассмотреть очень большое число пар состояний, для каждой из которых следует выполнять проверку. Во вторых, для оптимизированного некоторой процедурой ЦУ вообще может не существовать формальной модели, т.е. применение определений 2.10 или 2.12 невозможно.
2. Проверка свойств 2.16 или 2.17 на основании моделирования использует 3-значную логику. Процедуры такого моделирования хорошо разработаны и присутствуют в любой САПР соответствующего назначения.

Определение ТНЗ обладает ещё двумя положительными свойствами с точки зрения практики их применения в процессе дизайна ЦУ [105].

1) Свойство композиции. Пусть задано ЦУ на логическом уровне описания (цифровая схема). Данное ЦУ можно рассматривать как состоящее из некоторого числа подсхем. Если некоторые (или даже все) подсхемы будут оптимизированы с выполнением свойства ТНЗ, то новое ЦУ, составленное из таких подсхем, будет находиться с оригинальным ЦУ в отношении ТНЗ.
2) Свойство сохранения инициализирующей последовательности. Пусть ЦУ A_2 является трёхзначной неизменяющей заменой для ЦУ A_1, а также входная последовательность S является инициализирующей последовательностью для A_1. Тогда данная последовательность S будет инициализирующей последовательностью для ЦУ A_2.

Два указанных свойства открывают очень широкие возможности для оптимизации разрабатываемого дизайна. Например, в некотором большом ЦУ можно выделить его структурные элементы и независимо производить их оптимизацию с учётом свойства трёхзначной неизменяющей замены. Соединяя затем оптимизированные фрагменты, будет получено ЦУ, для которого инициализирующая последовательность исходного ЦУ также будет являться инициализирующей.

Данные определения (2.16 и 2.17) эквивалентности в трёхзначных логиках могут быть эффективно использованы, например, совместно с моделью склеенных ЦУ (рис.2.9), используемой для верификации эквивалентности поведения. Задача сводится к построению теста для неисправности *const*0 на выходе объединяющего элемента Исключающее-ИЛИ.

Определение 2.18. Неисправность *const*0 является непроверяемой, если не существует такой входной последовательности S, что выход ЦУ $g(z_u, S) = 1$, или другими словами пара $(A_1(z_u, S), A_2(z_u, S))$ не принимает значения (0,1) или (1,0).

Однако свойство непроверяемости неисправности *const*0 на выходе объединённого ЦУ является необходимым условием для свойства 3-значной неизменяющей замены. Т.е., если неисправность *const*0 на выходе объединённого ЦУ является непроверяемой, то может су-

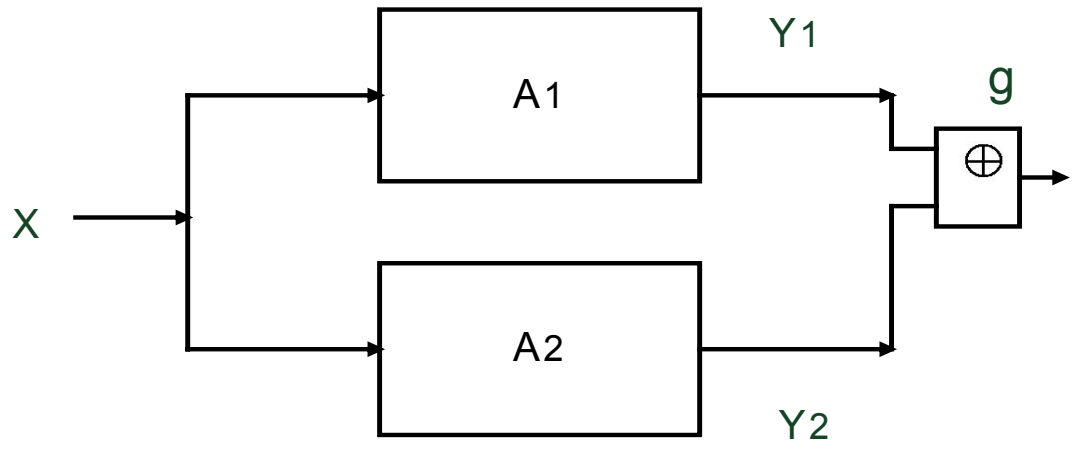


Рис.2.9. Модель склеивания при верификации поведения двух ЦУ.

ществовать некоторая входная последовательность S такая, что пара значений $(A_1(z_u,S),A_2(z_u,S))$ будет принадлежать множеству $\{(0,u),(1,u)\}$, что, очевидно, удовлетворяет определению 2.16.

Свойство 2.1. ЦУ A_2 является ТНЗ для ЦУ A_1 тогда и только тогда, когда не существует входной последовательности S и состояния z таких, что пара значений $(A_1(z_u,S),A_2(z,S))$ принадлежит множеству $\{(0,1),(1,0)\}$.

Если состояние z из определения выше является полностью неопределённым состоянием, то последовательность S будет тестом для неисправности *const*0 в модели объединённых ЦУ. Поскольку с помощью такой последовательности S по данному определению показывается, что ЦУ A_1 и A_2 не состоят в отношении ТНЗ, то последовательность S будет являться различающей в смысле данного свойства.

Аналогично, для того чтобы показать трёхзначную неэквивалентность двух заданных ЦУ, необходимо строить такую последовательность S, которая для них порождает одну из шести запрещённых выходных комбинаций (определение 2.11): $(0,1)$, $(1,0)$, $(0,u)$, $(1,u)$, $(u,0)$, $(u,1)$.

Последние два замечания позволяют переформулировать задачу верификации эквивалентности: вместо доказательства эквивалентности двух заданных ЦУ следует строить последовательности, которые показывают их неэквивалентность в смысле заданных определений. В качестве определений эквивалентности выбираются определения 2.16 и 2.17.

Такая переформулировка проблемы имеет ряд преимуществ с точки зрения построения метода решения задачи верификации эквивалентности.

1) При построении решения можно перейти от автоматного задания ЦУ как объектов верификации к структурному их заданию на логическом уровне.
2) Описание ЦУ на структурном уровне даёт разработчику гораздо больше информации, чем автоматное описание. Это связано с тем, что на этом уровне детализации считается доступной информация о поведении всех линий ЦУ во время моделирования, а не только выходные реакции, как для автоматного задания.
3) Нет необходимости проводить формальное доказательство эквивалентности заданных ЦУ, что является задачей с повышенными временными требованиями.
4) Подход является конструктивным в том смысле, что он определяет те свойства последовательностей, претендующих быть решением, которые необходимо проверить.
5) Для проверки свойств последовательностей быть различающими используется моделирование в 3-значных логиках.

С точки зрения разработчика такой подход также не является противоречивым: формальное доказательство эквивалентности ЦУ до и после оптимизации не является самоцелью, гораздо более ценной является информация об эквивалентности/неэквивалентности их поведения в результате такого преобразования.

Таким образом, задача верификации эквивалентности поведения двух заданных ЦУ сводится к задаче построения некоторой входной ИдП. Семантическая нагрузка свойств строящейся последовательности формулируется следующим образом:

- последовательность должна обнаружить различные выходные реакции хотя бы на одном из внешних выходов $y_1, y_2, ..., y_n$ двух верифицируемых ЦУ A_1 и A_2;

- если не удалось этого сделать, то различие в поведении следует искать на линиях состояний $z_1, z_2, ..., z_m$ анализируемых ЦУ; данное различие будет показывать, что ЦУ находятся в разных состояниях и данное различие скорее должно проявиться на внешних выходах;
- если несоответствие не обнаружено и на линиях состояний, то его следует искать на внутренних линиях анализируемых ЦУ, предполагая, что это позволит различию поведения распространиться сначала на внутренние состояния, а затем на внешние выходы ЦУ.

Последнее условие требует отдельного замечания. Оптимизация ЦУ на вентильном уровне описания в средах САПР обычно производится для некоторого комбинационного фрагмента. При этом последовательностная структура ЦУ не изменяется. Однако невозможно точно соотнести внутреннюю линию комбинационного фрагмента до и после преобразования. Это решается установкой контрольных точек на выходах преобразуемых фрагментов ЦУ.

Поскольку решением задачи верификации эквивалентности является ровно одна последовательность, а не их набор, то ГА-метод решения строится по одноуровневой схеме (рис.2.1) и структурно соответствует шаблону – Алгоритм А2.1 [137]. Для построения на основе шаблона ГА-метода генерации верифицирующих эквивалентность последовательностей доопределим его компоненты.

В качестве особи выбирается одиночная двоичная последовательность (рис.2.4). Для инициализирующей последовательности нет эвристик, показывающих её начальную, либо конечную длину. Поэтому длина особей в алгоритме не ограничена. В начальной популяции длина особей принимается равной единице $L = 1$. Длина последовательностей в популяции растёт по поколениям.

Традиционно для методов данного типа функция оценки строится на основании моделирования поведения ЦУ. В данном методе для вычисления оценки качества особи требуется выполнить моделирование двух верифицируемых ЦУ: A_1 и A_2. Предполагается, что данные ЦУ имеют одинаковую последовательностную структуру и минимальные различия в комбинационных блоках.

Семантика качества входной последовательности приведена выше. При применении «наивных методов» формализации многокритериальной оптимизации её формализация для двух заданных ЦУ A_1 и

A_2 и входной последовательности-особи $S = (X_1,...,X_n)$ задаётся выражением:

$$O(A_1, A_2, S) = \sum_{j=1}^{\text{длина } S} O(A_1, A_2, X_j) =$$

$$= \sum_{j=1}^{\text{длина } S} (c_1 \cdot n_1(A_1, A_2, X_j) + c_2 \cdot n_2(A_1, A_2, X_j) + \\ + c_3 \cdot n_3(A_1, A_2, X_j)), \qquad (2.26)$$

где:

- X_j - j-й входной набор последовательности S;

- c_1-c_3 - нормализующие константы, которые уравновешивают различие поведений заданных ЦУ на внешних выходах, линиях состояний и внутренних линиях (контрольных точках);

- n_1-n_3 - числовые параметры, определяемые по результатам моделирования работы ЦУ A_1 и A_2.

Смысл данных параметров исходит из семантики качества последовательности и заключается в следующем:

- $n_1(A_1, A_2, X_j)$ показывает число различий на множестве внешних выходов двух ЦУ A_1 и A_2 по результатам моделирования на j-м входном наборе последовательности S:

$$n_1(A_1, A_2, X_j) = \sum_{g \in Y} r(g, X_j, A_1, A_2), \qquad (2.27)$$

где Y - множество внешних выходов ЦУ A_1 и A_2;

- $n_2(A_1, A_2, X_j)$ показывает число различий на множестве линий состояний двух ЦУ A_1 и A_2 по результатам моделирования на j-м входном наборе последовательности S:

$$n_2(A_1, A_2, X_j) = \sum_{g \in Z} r(g, X_j, A_1, A_2), \qquad (2.28)$$

где Z - множество линий состояний ЦУ A_1 и A_2;

- $n_3(A_1, A_2, X_j)$ показывает число различий на множестве выходов комбинационных блоков (множестве контрольных точек) двух ЦУ A_1 и A_2 по результатам моделирования на j-м входном наборе последовательности S:

$$n_3(A_1, A_2, X_j) = \sum_{g \in G} r(g, X_j, A_1, A_2), \tag{2.29}$$

где G - множество линий комбинационных блоков ЦУ A_1 и A_2.

Функция $r()$ является функцией различия поведения устройств A_1 и A_2, вычисляется на основании моделирования. При использовании двухзначного алфавита моделирования E_2 и определяется согласно формуле:

$$r_2(g, X_j, A_1, A_2) = g(X_j, A_1) \oplus g(X_j, A_2), \tag{2.30}$$

где: 2 – индекс в $r()$, показывающий значность алфавита моделирования; $g(X_j, A_k)$ - выходное значение на выходе комбинационного блока g при завершении моделирования ЦУ $A_k, k = \overline{1,2}$ на входном наборе X_j.

Функция различия $r_2()$ для двухзначного алфавита моделирования легко выражается через введённые ранее функции достижения U^0 и U^1. Будем иметь:

$$\begin{aligned} r_2(g, X_j, A_1, A_2) = (U^0(g, X_j, A_1) = U^1(g, X_j, A_2)) \vee \\ \vee (U^1(g, X_j, A_1) = U^0(g, X_j, A_2)), \end{aligned} \tag{2.31}$$

Видно, что сравнение в (2.31) возможно проводить только по тем элементам g, соответствие которых в устройствах A_1 и A_2 не нарушено процедурой оптимизации.

Активность на выходе различных компонент ЦУ по разному может влиять на активность его выходов. Влияние значения на выходе

элемента g_i на внешний выход ЦУ называется наблюдаемостью и задаётся численным значением I_g (табл.2.1). С учётом данного параметра (2.31) можно записать в виде:

$$r_2(g, X_j, A_1, A_2) = I_g \cdot (U^0(g, X_j, A_1) = U^1(g, X_j, A_2)) \vee \\ \vee (U^1(g, X_j, A_1) = U^0(g, X_j, A_2))). \quad (2.32)$$

Для больших ЦУ информация о наблюдаемости элементов может быть недоступна, однако её наличие должно повышать качество оценки последовательности. В дальнейшем мы будем использовать функцию $r()$ в виде (2.31) без учёта параметра наблюдаемости I. Однако будем считать, что если он доступен, то может быть учтён в виде (2.32). Такое допущение не нарушит общности рассуждений, поскольку в обоих этих случаях вид формулы оценки в виде (2.26) не изменится.

Вычисление значений функции (2.26) и входящих в неё компонент (2.27)-(2.29) выполняется после каждого такта модельного времени путём сравнения массивов значений сигналов SV на линиях ЦУ A_1 и A_2.

Функция $r_2()$ (2.30) может принимать два значения и определена в двоичном алфавите E_2. Однако разрабатываемый метод ориентируется на определения эквивалентности устройств, которые заданы в 3-значных логиках. Определим данную функцию для заданных определений эквивалентности 2.16 и 2.17.

1) Для алгоритма верификации ТНЗ функция $r()$ будет принимать значений 1, если пара выходов элементов $g_1(X_j, A_1)$ и $g_2(X_j, A_2)$ принадлежит запрещённому множеству $D_1 = \{(0,1),(1,0)\}$:

$$r_3^{'}(g, X_j, A_1, A_2) = \begin{cases} 1, \textit{если } (g(X_j, A_1), g(X_j, A_2)) \in D_1; \\ 0, \textit{в противном случае.} \end{cases} \quad (2.33)$$

2) Для алгоритма верификации трёхзначной эквивалентности функция $r()$ будет принимать значений 1, если пара выходов элементов

$g_1(X_j, A_1)$ и $g_2(X_j, A_2)$ принадлежит запрещённому множеству $D_2 = \{(0,1),(1,0),(u,0),(u,1),(0,u),(1,u)\}$:

$$r_3''(g, X_j, A_1, A_2) = \begin{cases} 1, \textit{если } (g(X_j, A_1), g(X_j, A_2)) \in D_2; \\ 0, \textit{в противном случае.} \end{cases} \qquad (2.34)$$

Видно, что аналогично $r_2()$ в (2.32) функции различия в трёзначных логиках (2.33) и (2.34) легко могут быть выражены через функции достижения U^0 и U^1.

Таким образом, применяя функции различия в виде (2.33) либо (2.34) в функции оценки (2.26) мы будем строить различающие последовательности для определений ТНЗ и ТЭ соответственно.

Как было сказано выше, константы c_1, c_2, c_3 определяют вес (т.е. важность) каждого из параметров n_1, n_2 и n_3 в достижении цели.

Видно, что оценочная функция (2.26) является многокритериальной, и, как в случае построения инициализирующих последовательностей, строится по аддитивному принципу.

Очевидно, что различие на внешних выходах сравниваемых ЦУ (параметр n_1) имеет наибольший вес в функции оценки, поскольку ненулевое значение данного параметра говорит о различии внешнего поведения ЦУ, т.е. означает решение задачи.

Следующим по важности является параметр n_2. Его ненулевое значение показывает различие поведения на линиях состояний. В автоматной нотации это означает, что автоматы A_1 и A_2 находятся в разных внутренних состояниях. В этом случае вероятность распространения различия в поведении двух заданных ЦУ на внешние выходы гораздо выше, чем в случае при $n_1 = 0$.

Наименьший вес имеет параметр n_3, ненулевое значение которого показывает различное поведение на некоторых комбинационных блоках внутри сравниваемых ЦУ. Пренебрегать данным параметром нельзя в том случае, если и $n_1 = 0$ и $n_2 = 0$. В этом случае ненулевое значение n_3 повышает вероятность последовательного распространения различия в поведении сначала на элементы состояний, а затем на

внешние выходы сравниваемых ЦУ.

Выбор конкретных значений констант описан в главе 5 (раздел 5.4) при апробации метода.

Псевдокод алгоритмической реализации ГА-метода верификации эквивалентности при определённых компонентах приведён ниже.

Алгоритм А2.8

```
ГА_ВерификацияЭквивалентности(A1, A2,Параметры)
{
  Pop_нач=ПостроениеНачальнойПопуляции(N_особ, L);
  ОценитьПопуляцию(Pop_нач, A1, A2, N_особ);
  Pop_тек = Pop_нач;
  НомерПопуляции=0;
  while( НеДостигнутКритерийОстановки() )
  {
    for( int i=0 ; i<N_особ ; i++ )
    {
      РодительА=ВыбратьРодителя(Pop_тек);
      РодительВ=ВыбратьРодителя(Pop_тек);
      if( rand() > p_скр )
       Потомок=ВыполнитьСкрещивание(РодительА,РодительБ);
      else
        Потомок=РодительА;
      if( rand() > p_мут )
        Потомок=ВыполнитьМутацию(Потомок);
      ДобавитьОсобьВПопуляцию(Pop_тек,Потомок);
    } // конец for – построение промежуточной популяции
    ОценитьПопуляцию(Pop_тек, A1, A2, N_особ*2);
    Pop_тек=УсечениеПопуляции(Pop_тек, N_особ, Fac);
    НомерПопуляции++;
  } // конец while – достигнут критерий остановки
  СортироватьПопуляциюПоОценке(Pop_тек);
  // решение=лучшая особь в посл. популяции
  Решение=ТекущаяПопуляция[0];
} // конец одноуровневого ГА верификации эквивалентности
```

Дадим пояснения к псевдокоду. Метод в качестве входных параметров получает описание двух сравниваемых ЦУ A_1 и A_2, а также следующие неуказанные в коде параметры:

- $N_{особ}$ – количество особей в популяции;
- $N_{пок}$ – максимальное число циклов построения новых популяций;
- $N_{бу}$ – построение новых популяций будет прервано, если оценка лучшей особи не улучшается заданное число раз;
- Fac - фактор обновления популяции, показывает долю популяции, которая будет заменена на новые особи в процессе конструирования популяции для следующей итерации.

Смысловая нагрузка переменных и процедур в реализации соответствует шаблону одноуровневых ГА-методов. В реализации применяются критерии остановки работы, перечисленные при построении шаблона одноуровневого ГА построения ИдП (раздел 2.2.1).

В данной реализации применена схема ГА без построения промежуточной популяции. Также реализовано неявное вычисление фитнесс-функции при реализации операции выбора особей в качестве родителей. В остальном данная реализация совпадает с реализацией ГА-метода построения инициализирующих последовательностей.

Наиболее существенным отличием приведённой реализации от предыдущих одноуровневых ГА-ориентированных методов построения ИдП является необходимость работы с двумя верифицируемыми ЦУ A_1 и A_2.

Функция оценки всех особей в популяции соответствует псевдокоду алгоритма А2.4 с учётом необходимости моделирования поведения двух ЦУ. Псевдокод процедуры оценки одной последовательности-особи приведён ниже.

Алгоритм А2.9

```
ОценитьОсобь(A1, A2, S)
{
  Оценка=0.0;
  Длина=ДлинаПоследовательности(S);
  for( int i=0 ; i<Длина ; i++ )
  {
    ТекущееРазличие=0.0;
    Xi=S[i];
    SV1=ИсправноеМоделирование(A1, Xi);
    SV2=ИсправноеМоделирование(A2, Xi);
```

```
    n1=СравнениеПоведения(SV1, SV2);
    n2=СравнениеПоведения(SV1, SV2);
    n3=СравнениеПоведения(SV1, SV2);
    ТекущееРазличие=cc1·n1+c2·n2+c3·n3;
    Оценка=Оценка+ТекущееРазличие; // формула 2.26
  } // конец цикла по длине последовательности
  return O=Оценка;
} // конец – вычисление оценки особи
```

В данном псевдокоде:

- *ТекущееРазличие* – переменная, показывающая различие в поведении схем на текущем входом наборе X_i;
- *Оценка* – численное значение оценочной функции в соответствии с (2.26).

Видно, что описанный алгоритм в целом производит верификацию эквивалентности поведения двух заданных ЦУ A_1 и A_2 для последовательностей-особей, которые в пространстве поиска стремятся к тем участкам, где выше оценочная функция (2.26). Однако число итераций алгоритма ограничено сверху и в пространстве поиска он рассмотрит только конечное число точек, поэтому алгоритм не гарантирует нахождения точки с максимальным значением оценки.

Отсюда следует важное замечание. Если алгоритм закончит работу и не найдёт различающую последовательность, то это, вообще говоря, не означает эквивалентность поведения устройств A_1 и A_2. Возможно, среди нерассмотренных последовательностей есть такая, которая покажет неэквивалентность устройств A_1 и A_2. Однако, поскольку поиск шел в направлении последовательностей, которые продуцируют возможно большее различие во внутреннем поведении анализируемых ЦУ A_1 и A_2, то отрицательный результат поиска говорит о высокой вероятности эквивалентности поведения устройств A_1 и A_2.

Апробация метода [137] и описание методики численные эксперименты приведены в разделе 5.4, а результаты экспериментов в табл.5.2. Приведённые числовые данные показывают высокую эффективность предложенного алгоритма: число экспериментов, в которых удалось построить различающую функцию, составило 95.77%,

что является очень высоки показателем. Число экспериментов, в которых различие функционирования не проявилось даже внутри схемы, составило менее 2%.

Глава 3

ДВУХУРОВНЕВЫЕ ГЕНЕТИЧЕСКИЕ АЛГОРИТМЫ ПОСТРОЕНИЯ ВХОДНЫХ ИДЕНТИФИЦИРУЮЩИХ ПОСЛЕДОВАТЕЛЬНОСТЕЙ ЦУ

3.1. Шаблон двухуровневых ГА-методов построения идентифицирующих последовательностей

Одноуровневые ГА-методы построения ИдП (глава 2) следует рассматривать как первый этап в применении ГА к решению задач технической диагностики. Однако целый ряд задач не может быть сведён к однократному вызову ГА поиска в силу того, что решение задачи требует поиска нескольких промежуточных решений. Поскольку в процессе поиска в разное время ставятся различные подзадачи, то в оценочной функции появляется параметр (параметры), который зависит от текущей локальной *Цели* (рис.2.2). В силу специфики исследования в данном разделе мы будем рассматривать случаи, когда поиск решения для локальных целей сводится к вызову процедуры в той или иной степени использующей эволюционный поиск на основе ГА. Такую схему построения решения задачи будем называть двухуровневой схемой применения ГА, а разработанные на её основе методы – двухуровневыми ГА-методами.

Также в данном разделе будет показано, что такие методы в качестве нижнего уровня используют одноуровневые ГА-методы, описанные в предыдущем разделе, в частности методы верификации эквивалентности поведения ЦУ и метод достижение заданных состояний в ЦУ

В данном разделе мы не будем касаться тех методов, в которых ГА либо детерминированный метод могут использоваться попеременно для достижения одной цели. В качестве такого примера можно

отметить те методы, где на начальном этапе в качестве стратегии поиска используется ГА, а в случае отсутствия прогресса происходит переход к некоторому детерминированному методу. Это связано с тем, что такие методы показывают, скорее, возможный контекст применения ГА, однако не касаются непосредственно разработки ГА-методов построения ИдП.

Разработаем шаблон верхнего уровня двухуровневых методов построения ИдП, в котором в качестве эволюционной поисковой процедуры выступает ГА.

Укрупнённая блок-схема двухуровневых ГА-методов приведена на рис.2.2. Центральным элементом всех методов, основанных на данной схеме, является объект *Цель*. Природа данного объекта уточняется при построении конкретного метода. Главный цикл верхнего уровня таких методов включает два шага:

- поиск текущей *Цели*; если невозможно найти такую цель, то работа метода завершается;
- поиске решения, которое ведёт к достижению текущей *Цели*.

При детализации данной схемы применительно к задачам построения ИдП в качестве метода достижения текущей *Цели* выбирается эволюционный поиск основанный на ГА. Вообще говоря, поиск *Цели* также может выполняться как с помощью ГА, так любого другого ЭА. Однако определяющим для нас будет являться именно поиск решения с помощью ГА, а его использование в другом месте методов – второстепенным.

Пусть задано ЦУ A_0. Шаблон верхнего уровня для двухуровневых ГА-методов построения ИдП может быть представлен следующим образом.

Алгоритм А3.1

```
Двухуровневый_ГА_построения_ИдП(A0, Параметры)
{
  ПредварительнаяОбработка(A0);
  while( НеДостигнутКритерийОстановки() )
  {
    Цель=ВыборТекущейЦели(A0, Параметры);
    if( Цель==НетЦели )
      return; // нет цели – завершение работы алгоритма
    else
```

```
    {
      // цель найдена-вызов ГА поиска локального решения
      S=ГА_Построение_ИдП(A0,Цель,НачальнаяПопуляция);
      if( S==NULL ) // последовательность не построена
        ОтметитьКакНедостижимую(Цель);
      else
      {
        ДобавитьВТест(S);
        ДополнительнаяПроверка(A0,S);
      }  // конец else – последовательность построена
    }    // конец else – выбрали цель
  }      // конец while – не достигли критерия остановки
}        // конец двухуровневого ГА-метода
```

Предварительная обработка, представленная в коде одноименной процедурой, может включать в себя построение множества промежуточных *Целей*, подготовку структуры данных для работы с ними в дальнейшем (списки, массивы и т.д.), а также вычисление статических параметров оценочных функций.

Если в итерации текущая *Цель* найдена, то вызывается ГА-метод, который строит промежуточную последовательность S. При этом структурно такой ГА является одноуровневым и соответствует шаблону метода А2.1. Начальная популяция для ГА нижнего уровня может строиться в процессе поиска *Цели* на верхнем уровне. Такой подход реализован, например в [52, 138, 48].

После построения промежуточной последовательности S и в зависимости от конкретного метода может быть выполнено дополнительное моделирование [48-49]. Введение такой эвристики связано с тем, что построенная для некоторой локальной цели промежуточная последовательность S может обладать более широкими идентифицирующими свойствами. Именно тот факт и устанавливается в процедуре *ДополнительнаяПроверка()*.

Результирующее решение строится по аддитивному принципу: итоговая последовательность состоит из совокупности промежуточных последовательностей. К итоговой последовательности далее могут применяться различные процедуры оптимизации.

Данный шаблон является обобщающим для достаточно широкого класса методов. Применение различных типов объектов *Цель* и различных процедур ГА будет порождать целое множество различных

методов.

Далее в данном разделе при разработке соответствующих методов будет показано, что в качестве объекта *Цель* может выступать как некоторая неисправность, так и множество неисправностей.

Видно, что в представленном шаблоне верхнего уровня конкретная реализация ГА нижнего уровня не приводится. В качестве таких процедур будут использованы ГА-методы верификации эквивалентности поведения двух заданных ЦУ(методы А2.8-А2.9) и достижения состояния в ЦУ (А2.6-А2.7), разработанные в разделе 2.

В данной главе на основе представленного шаблона А3.1 будет показано построение следующих двухуровневых ГА-методов:

- метод построения проверяющих тестов с верификацией поведения неисправного ЦУ; в данном методе на верхнем уровне в качестве объекта *Цель* выбирается некоторая непроверенная неисправность $f_i \in F$; на нижнем уровне ГА ищет такую последовательность, которая будет показывать различие поведения исправного устройства A_0 и неисправного устройства A_i (соответствующего текущей целевой неисправности) на выходе ЦУ;
- метод построения проверяющих тестов с подтверждением состояний; верхний уровень метода соответствует предыдущему случаю; на нижнем уровне для целевой неисправности f_i сначала происходит распространение её влияния на внешние выходы и продвижение назад в пределах текущего такта модельного времени с получением состояния ЦУ Z_i, которое обеспечивает распространение влияния неисправности; далее ГА ищет такую последовательность, которая переводит исправное и неисправное ЦУ в состояние Z_i;
- метод построения диагностических тестов; здесь в качестве объекта *Цель* верхнего уровня выбирается множество неразличимых к текущему моменту времени неисправностей F_i; на нижнем уровне для данного множества ГА строит различающую последовательность.

Поскольку в шаблоне А3.1 показывается только место процедур ГА, то без потери общности данный шаблон может быть применён к случаю, когда поиск решения на нижнем уровне будет осуществляться с помощью других ЭА, например СО [130]. Однако разработка таких методов выходит за рамки данной работы.

3.2. Двухуровневый ГА-метод построения проверяющих тестов ЦУ с активизацией неисправностей

Автоматизированное построение тестов ЦУ является одной из наиболее часто рассматриваемых задач в технической диагностике [30-40].

Пусть задано исправное ЦУ A_0. Будем в качестве модели использовать синхронные последовательностные ЦУ. Также задано множество неисправностей $F = \{f_1,..., f_n\}$, причём считается, что данное множество конечно: $|F| = n < \infty$. Неисправности множества F порождают множество неисправных ЦУ $A = \{A_1,..., A_n\}$. В качестве модели неисправности может выступать любая такая модель, для которой разработаны методы её моделирования. Множество неисправных ЦУ не строится явно, а будет порождаться путём внесения влияния неисправностей в процессе моделирования ЦУ.

Требуется определить исправность заданного ЦУ $A_i \in A$, т.е. проверить соответствует ли его поведение A_0.

Определение 3.1. Входная последовательность S, выходные реакции на которую заданных устройств A_0 и A_i различны, называется проверяющей, т.е. $A_0(S) \neq A_i(S)$.

Также такая последовательность S называется обнаруживающей неисправность $f_i \in F$, которая порождает неисправное устройство A_i.

Определение 3.2. Входная последовательность S называется тестом (полным тестом), проверяющим множество A, если она проверяет каждое ЦУ $A_i \in A$, $i = \overline{1,n}$.

В этом случае говорят также, что S является тестом для множества неисправностей F.

Определение 3.3. Эффективной длиной $l_{эф}$ последовательности S, которая является проверяющей для неисправности f_i, на-

зывается минимальный такт модельного времени t, для которого выходные реакции устройств A_0 и A_i станут различны.

Введение понятия эффективной длины последовательности для ГА-методов построения тестов связано с тем, что в процессе применения эволюционных операций длина l последовательности S может стать достаточно большой. При этом может оказаться, что эффективная длина $l_{эф}$ меньше, чем длина особи S. В этом случае необходимо отбросить все входные наборы в последовательности S, которые соответствуют тактам времени от $l_{эф}+1$ до l.

Задача тестирования множества неисправностей F может быть представлена как задача итеративного поиска тестовой последовательности S_i для каждой неисправности $f_i \in F$, $i = 1,...,|F|$. В силу специфики данной работы мы рассматриваем только те методы, которые для построения теста неисправности f_i в той или иной мере используют поиск основанный на ГА.

Такой итеративный подход соответствует двухуровневой схеме применения ГА (рис.2.2) и реализуется на основе шаблона А3.1. При этом:

- неисправность $f_i \in F$ определяет объект *Цель* метода;
- фаза выбора неисправности соответствует верхнему уровню метода;
- построение теста неисправности f_i с помощью ГА определяет нижний уровень метода.

Тогда алгоритм работы верхнего уровня ГА-метода построения тестов можем быть представлен в виде укрупнённой блок-схемы, приведённой на рис.3.1.

Данная блок-схема позволяет также реализовывать различные стратегии построения тестов. Рассмотрим один из наиболее популярных подходов, который основан на предварительной активизации неисправностей.

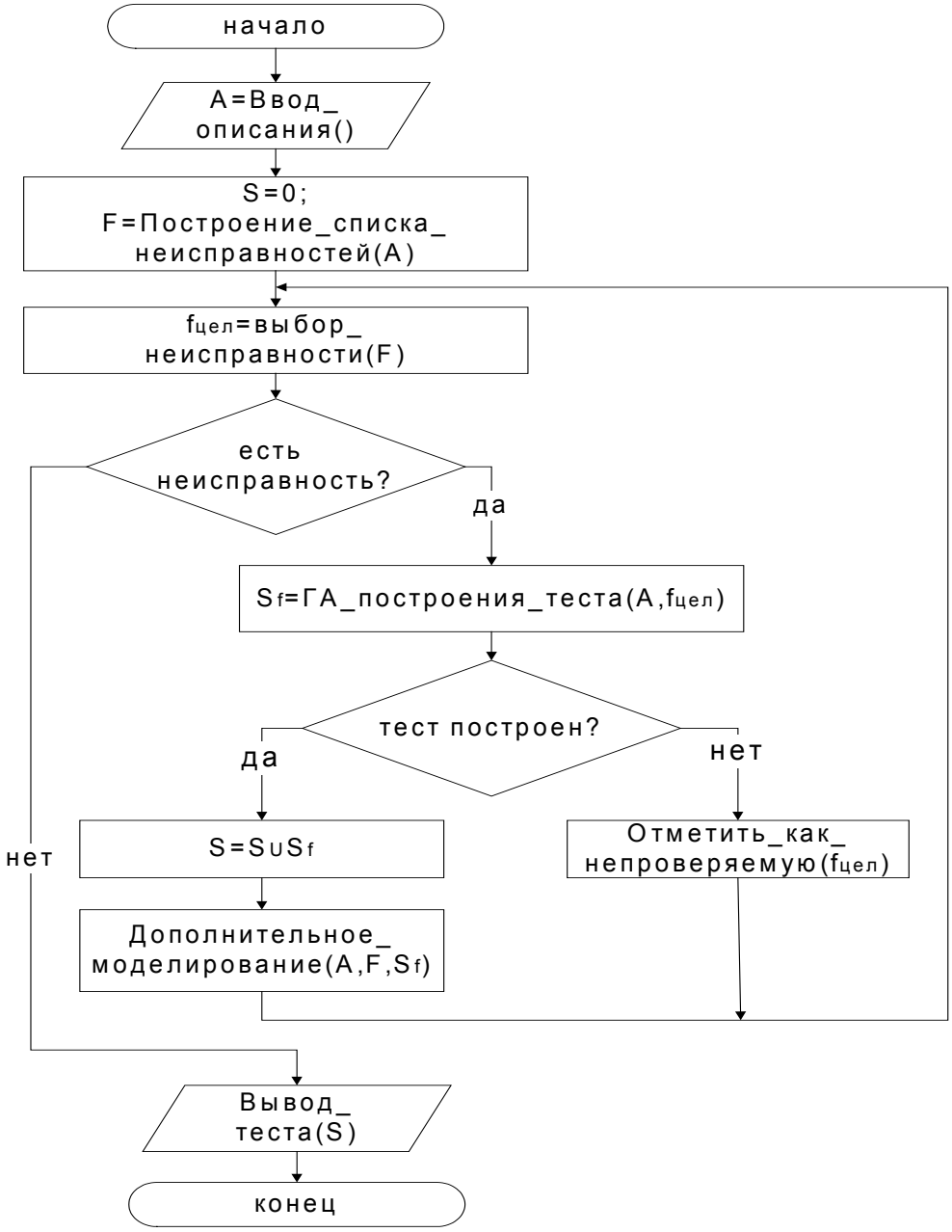


Рис.3.1. Укрупнённая блок-схема верхнего уровня для двухуровневых ГА-методов построения тестов.

Сложность задачи построения тестов обуславливается, в основном, тем, что в процессе проведения экспериментов с устройствами A_0 и A_i (исправным и неисправным) не всегда возможно однозначно восстановить их поведение. Такая неопределённость выходных сигналов, в свою очередь, связана с неопределённостью начальных со-

стояний данных ЦУ. Т.е. задача существенно упростилась бы в том случае, если бы удалось различие поведения двух сравниваемых ЦУ A_0 и A_i сначала распространить на их элементы состояний.

Определение 3.4. Пусть задано исправное ЦУ A_0, неисправность $f_i \in F$, порождающая неисправное устройство A_i, и некоторая входная последовательность $S_{акт}$. Говорят, что неисправность f_i активизирована последовательностью $S_{акт}$, если после подачи данной последовательности на устройства A_0 и A_i, начинающие работу из полностью неопределённого состояния Z_u, существует хотя бы одна линия состояния $z_j \in Z$, для которой различны значения в устройствах A_0 и A_i. При этом выходные реакции двух ЦУ одинаковы: $A_0(Z_u, S_{акт}) = A_i(Z_u, S_{акт})$.

Видно, что для автоматного задания ЦУ активизация неисправности соответствует случаю, когда два сравниваемых ЦУ перешли в различные состояния.

Определение 3.5. Эффективной длиной последовательности $S_{акт}$, которая является активизирующей для неисправности f_i, называется минимальный такт модельного времени t, при котором для исправного и неисправного ЦУ A_0 и A_i будет найдено различие на линиях состояний.

Очевидно, что если некоторая неисправность f_i активизируется последовательностью $S_{акт}$, то задача построения теста для неё существенно упрощается. При этом дальнейшее конструирование тестовой последовательности для f_i следует начинать с $S_{акт}$.

Таким образом, если разбить задачу тестирования на две подзадачи (фазы), активизация неисправности и последующее обнаружение, то каждую подзадачу можно отобразить на свой уровень:

- активизация влияния неисправности $f_i \in F$ и её выбор в качестве *Цели* – верхний уровень;

- распространения влияния f_i на внешние выходы ЦУ – нижний уровень.

При таком разбиении на подзадачи происходит дальнейшее уточнение блок-схемы, приведённой на рис.3.1.

Опишем работу метода, который соответствует верхнему уровню. Здесь суть заключается в выборе некоторой неисправности (далее будем называть её целевой) $f_{цел} \in F'$, для которой удалось распространить её влияние на элементы состояний в схеме, где F' - текущее множество непроверенных неисправностей. В начале работы метода $F' = F$.

Построение активизирующей последовательности происходит путём псевдослучайной генерации входных последовательностей и их моделировании на множестве F'. Если при этом в некоторый момент времени обнаружится, что последовательность $S_{вх}$ проверила некоторую(ые) неисправность из F', то последовательность $S_{вх}$ добавляется в финальный тест, а проверенные неисправности удаляются из множества F'.

Если по результатам моделирования некоторой неисправности $f_i \in F'$ видно, что данная последовательность является активизирующей, то данная неисправность выбирается в качестве целевой $f_{цел} = f_i$ и вместе с активизирующей последовательностью $S_{акт}$ они передаются в качестве аргументов для ГА построения теста одной неисправности на нижний уровень метода.

Если ГА для целевой неисправности $f_{цел}$ сумел построить тест $S_{тест}$, то он добавляется в тестовую последовательность, а неисправность $f_{цел}$ удаляется из F'. Если же ГА на нижнем уровне не смог построить тест, то текущая целевая неисправность $f_{цел}$ отмечается как непроверяемая (с помощью данного алгоритма) и не будет выбираться в качестве целевой в дальнейшем.

Дополнительно после построения теста $S_{тест}$ для заданной целевой неисправности $f_{цел}$ производится моделирование на множестве ещё непроверенных неисправностей F'. Если последовательность $S_{тест}$ обнаружила некоторые неисправности из F', то они оттуда удаляются.

Псевдокод верхнего уровня ГА-метода построения тестов приведён ниже.

Алгоритм А3.2

```
ГенерацияПроверяющихТестов_ВерхнийУровень( A0 , Параметры)
{
  F =ПостроениеПолногоСпискаНеисправностей( A0 );
  F'= F ; // текущий список непроверенных неисправностей
  while( НеДостигнутКритерийОстановки() )
  {
    fцел =ВыборЦелевойНеисправностиИзСписка( A0 , F' ,параметры);
    if( fцел ==NULL )
      return; // возврат – целевая неисправность не найдена
    else
    {
      Sтест =ГА_ПостроениеТеста( A0 , fцел ,НачальнаяПопуляция);
      if( Sтест ==NULL ) // тест для неисправности не построен
      {
        ОтметитьКакНепроверяемую( fцел );
        F'=УдалитьИзСписка( F' , fцел );
      }
      else
      {
        ДобавитьВТест( Sтест );
        МоделированиеСНеисправностями( A0 , F' , Sтест );
        F'=ОбновитьМножество( F');
      }  // конец else – тест целефой неисправности построен
    }    // конец else – активизировали неисправность
  }      // конец while – не достигли критерия остановки
}        // конец алгоритма
```

Для проверки завершения работы в функции *НеДостигнутКритерий-Остановки()* метода используются следующие критерии:

- список непроверенных неисправностей F' пуст: $F'=\varnothing$;
- достигнуто предельное число итераций верхнего уровня;
- время работы алгоритма достигло предельного.

При выполнении хотя бы одного из критериев работа метода прекращается. Такой подход позволяет разработчику путём изменения соответствующих входных параметров метода изменять глубину поиска, придавая гибкость в выборе баланса между скоростью работы алгоритма и его точностью.

Результатом работы метода будет тестовая последовательность, которая состоит из набора последовательностей $S_{тест}$, построенных на разных итерациях.

Функция *ОбновитьМножество()* актуализирует множество F' путём удаления всех неисправностей, которые были проверены во время дополнительного моделирования с неисправностями.

Рассмотрим теперь подробнее процесс построения активизирующей последовательности и выбора целевой неисправности. Псевдокод данной части метода приведён ниже.

Алгоритм А3.3

```
ВыборЦелевойНеисправностиИзСписка(A0, F', Параметры)
{
  L = Lнач;
  while( НеДостиглиМаксимальногоЧислаИтераций() )
  {
    Pop = СлучайнаяГенерация(L);
    for( int i=0 ; i<Nособ; i++ )
    {
      МоделированиеСНеисправностями(A0, F', Pop[i]);
      if( ЕстьПроверенныеНеисправности() )
      {
        F'=ОбновитьМножество(F');
        ДобавитьВТест(Pop[i]);
      }
    }
    if( f =ЕстьАктивизированнаяНеисправность() )
      return f, Sакт;
    L=УвеличитьДлину(L);
  }
  return NULL; // не смогли активизировать
                  неисправность
}  // конец выбора целевой неисправности
```

В данной процедуре используется термин *Популяция* для обозначения набора последовательностей и унификации с нижним уровнем

метода, несмотря на то, что здесь (на верхнем уровне) не происходит эволюции особей. Кодирование особей и популяций соответствует выбранному в разделе 2.

В процедуре реализуется итеративная генерация набора случайных входных последовательностей, называемых популяцией Pop, размерность которой $N_{особ}$. После построения набора последовательностей (особей) для каждой из них $Pop[i]$, $i = \overline{1, N_{особ}}$ выполняется моделирование с неисправностями на множестве F'. Если некоторая последовательность $S = Pop[i]$ проверит новые неисправности, то она добавляется в тест, а множество F' обновляется путем удаления обнаруженных неисправностей. То, что в процессе псевдослучайной генерации последовательностей происходит обнаружение неисправностей, характерно для начальной стадии работы метода.

Далее в множестве F' происходит поиск активизированной неисправности в соответствии с определением 3.4. Если такая неисправность найдена, то она выбирается в качестве целевой $f_{цел}$ и процедура заканчивает работу. При этом популяция Pop, в которую входит активизирующая последовательность $S_{акт}$ становится начальной популяцией $Pop_{нач}$ для нижнего уровня алгоритма.

Если же не удалось найти активизированную неисправность, то увеличивается начальная длина L генерируемых последовательностей и происходит переход на новую итерацию. При этом их длина изменяется от начального значения $L_{нач}$ в соответствии со стратегией, реализованной в функции *УвеличитьДлину()*.

Итерации повторяются не более чем предварительно заданное число раз.

В настоящее время не предложено общей эвристики, позволяющей ускорить активизацию неисправностей в данной фазе метода ни для автоматного, ни для структурного уровней представления ЦУ. Для улучшения эффективности псевдослучайной генерации наиболее часто используется следующие эвристики:

- в качестве текущего набора потенциальных активизирующих последовательностей используют последовательности, полученные из тестовой последовательности последней итерации ГА на нижнем уровне, к которым применены операторы мутации [138];
- в качестве оценки длины последовательностей для псевдослучай-

ной генерации используют длину тестовой последовательности последней итерации ГА плюс 1 [139].

Опишем теперь подробно нижний уровень метода. Здесь для заданной целевой неисправности $f_{цел}$ происходит построение тестовой последовательности $S_{тест}$. Именно здесь реализуется генетический алгоритм поиска такого решения.

Все основные компоненты соответствуют описанным в главе 2 при описании шаблона одноуровневых ГА-методов: особью в данном ГА является входная двоичная последовательность $S = (X_1, X_2, ..., X_l)$; кодирование особей и популяций соответствует представленному на рис.2.3-2.4; множество эволюционных операций представлено на рис.2.5-2.8. При этом начальная популяция ГА $Pop_{нач}$ является заданной – она передаётся с верхнего уровня алгоритма.

Семантическая нагрузка качества особей заключается в следующем. Оценка данной входной последовательности S тем выше, чем большую различающую активность для устройств A_0 и A_f она произведёт в процессе моделирования, где A_f соответствует устройству в присутствии целевой неисправности. Формально такая зависимость представлена следующей формулой:

$$O(A_0, f_{цел}, S) = \sum_{i=1}^{длина(S)} \sum_{g \in G} r(g, X_i, A_1, A_f), \qquad (3.1)$$

где:

- X_i - i-й входной набор последовательности S;
- $r(g, X_i, A_1, A_f)$ - функция различия поведения устройств (2.30), определяющая различающую активность для входного набора X_i последовательности S и заданной неисправности $f_{цел}$.

Видно также, что внутренняя сумма в (3.1) полностью соответствует различающей активности (2.29) для одного входного набора.

Таким образом, для вычисления оценки (3.1) конкретной особи S, необходимо выполнить явное моделирование поведения двух ЦУ: исправного A и неисправного A_f.

В качестве функционала $O(A_0, f_{цел}, S)$ в (3.1) часто используется не суммирование, а выбор максимального значения [48]. Тогда функция оценки принимает следующий вид:

$$O(A_0, f_{цел}, S) = \max_i \sum_{g \in G} r(g, X_i, A_1, A_f). \qquad (3.2)$$

В табл. 2.2 различающей активности соответствуют параметры $R\_N_{вых}$, $R\_N_{тр}$, $R\_N_{эл}$, $R\_N_{к.т.}$. Поскольку предполагается, что разработчику при моделировании доступно для наблюдения поведение всех логических элементов, то параметр $R\_N_{к.т.}$ (различающая активность на множестве контрольных точек), обычно, не учитывается. Тогда (3.2) может быть записана в виде (2.26). Из данной формулы можно исключить параметр различия на внешних выходах верифицируемых ЦУ A_0 и A_f, поскольку наличие хотя бы одного различия $R\_N_{вых} > 0$ говорит о проверяемости неисправности $f_{цел}$. Причём такая информация всегда доступна после моделирования особи-последовательности.

С этими упрощениями оценка для одного входного вектора последовательности примет вид.

$$O(A_0, f_{цел}, X_i) = c_1 \cdot n_1(X_j, A_1, A_f) + c_2 \cdot n_1(X_j, A_1, A_f), \qquad (3.3)$$

где:

- c_1 и c_2 - нормирующие константы;
- функции различия n_1 и n_2 соответствуют функциям (2.27) и (2.28) с тем различием, что в качестве первого ЦУ A_1 выступает исправное ЦУ A_0, а качестве второго ЦУ A_2 - устройство A_f.

Также в реализациях для улучшения оценки особи-последовательности в виде (3.3) может быть учтён параметр наблюдаемости I_i элемента g_i путём использования функции различия в виде (2.32).

Таким образом, и в данном методе функция оценки выражается через функции достижения значений U^0 и U^1, введённые в преды-

дущем разделе.

Задание компонент ГА позволяет описать его работу. Псевдокод ГА-метода построения теста заданной целевой неисправности приведён ниже.

Алгоритм А3.4

```
ГА_ПостроениеТеста( A0 , fцел , Popнач )
{
  i = 0 ; //счётчик числа поколений
  Popi = Popнач ;
  ОценитьПопуляцию( A0 , fцел , Popi );
  while( НеДостигнутКритерийОстановки() )
  {
    Popпром =∅;
    Позиция=0;
    for( int j=0 ; j<ЧислоНовыхОсобей ; j++ )
    {
      РодительА=ВыбратьРодителя( Popi );
      РодительВ=ВыбратьРодителя( Popi );
      if( random< pскр )
       Потомок=ВыполнитьСкрещивание(РодительА,РодительВ);
      else
        Потомок= РодительА;
      if( random< pмут )
        Потомок=ВыполнитьМутацию(РодительА, РодительВ);
      ДобавитьВПромежуточную
      Популяцию( Popпром ,Потомок,Позиция);
      Позиция++;
    }
    ОценитьПопуляцию( A0 , fцел , Popпром );
    for( int j=0 ; j< Nособ ; j++ )
    {
      if( Popпром[j] обнаруживает fцел )
        return Sтест = Popпром[j];
    }
    i++; // увеличение счётчика числа поколений
    //если тест не найден – построение новой популяции
    ПостроитьНовуюПопуляцию( Popi , Popпром );
```

```
  } // конец цикла по поколениям
  return NULL;//после завершения всех попыток
}   // конец построения теста одной неисправности
```

В данном псевдокоде используются следующие обозначения:

- A_0 - заданное ЦУ, используется для исправного моделирования и моделирования неисправного ЦУ путём внесения заданной неисправности $f_{цел}$;
- $p_{скр}$ и $P_{рмут}$ - вероятности применения операторов скрещивания и мутации соответственно;
- Pop_i - популяция с номером i;
- $Pop_{пром}$ - промежуточная популяция;
- $Pop[i]$ - i-й элемент популяции (особь с номером i);
- *ОценитьПопуляцию()* - функция, в которой для каждой особи заданной популяции вычисляется оценка (3.2);
- *Позиция* – показывает в промежуточной популяции $Pop_{пром}$ текущую позицию, в которую будет добавляться новая особь.

Другие функции в псевдокоде (*ПостроитьНовуюПопуляцию()*, *ВыполнитьСкрещивание()*, *ВыполнитьМутацию()* и т.д.) и переменные (*РодительА*, *РодительБ*, *Потомок* и т.д.) имеют стандартную семантику ГА.

Если пройдено заданное число поколений, а тест для неисправности $f_{цел}$ не построен, то на верхний уровень передаётся пустой указатель, неисправность будет отмечена как непроверяемая и не будет рассматриваться в дальнейшем при выборе *Цели*.

Видно, что метод А3.4 нижнего уровня метода построения тестов практически полностью совпадает с методом верификации эквивалентности двух заданных ЦУ А2.8. Отличие заключается лишь в задании второго устройства для верификации: в методе А2.8 оно задано явно, тогда как в методе А3.4 поведение устройства задаётся путём внесении влияния неисправности. Данный различающий момент можно рассматривать лишь как технический в смысле уточнения метода А2.9 вычисления оценки особи. Тогда можно говорить, что двухуровневый ГА-метод построения тестов на нижнем уровне использует алгоритм верификации двух ЦУ.

Следовательно, двухуровневый ГА-метод построения тестов с

активизацией неисправностей включает в себя следующие компоненты: алгоритмы А3.2-А3.3 работы верхнего и нижнего уровней, А2.4 оценки популяции, оценочную функцию особи в виде (3.1).

В целом описанный двухуровневый ГА-метод построения тестов с активизацией неисправностей имеет общие черты с экспериментами над автоматами по Гиллу. Главное отличие заключается в отсутствии структурированности при построении теста целевой неисправности: в экспериментах с автоматами происходит поиск некоторого пути по автомату, который и покажет их различное выходное поведение. В данном же методе обязательная фиксация различия поведений устройств происходит только в один момент времени – при активизации неисправности, а дальнейшая последовательность строится целиком с помощью эволюционных операций, а не путём прохождения фиксированного набора вершин. При этом проверяющие свойств последовательности из диагностического эксперимента неявно фиксируются в оценочной функции.

Видно также, что рассмотренный двухуровневый ГА построения тестов не накладывает никаких ограничений на модель неисправностей. Он может быть легко распространён на случаи любых других типов неисправностей, для которых существуют методы логического моделирования с неисправностями (задержка распространения сигналов, перекрёстные неисправности, кратные неисправности и т.д.). При этом семантика построения оценочной функции особей-последовательностей и её представление в виде (3.1) и (3.3) останутся прежними. Уточнению подлежат лишь входящие в них функции различия поведений исправного и неисправного ЦУ (2.27)-(2.29). В этом смысле описанный ГА-метод следует рассматривать как обобщение методов [138, 48] на все такие типы неисправностей.

Апробация метода производилась с использованием модели одиночных константных неисправностей. Для вычисления оценок особей-последовательностей в виде (3.1)-(3.2) можно использовать любые известные алгоритмы логического моделирования исправных и неисправных ЦУ. При реализации нами использовался алгоритм, подробно описанный в [8]. Результаты апробации приведены в разделе 5.4.

3.3. Гибридный двухуровневый ГА-метод построения тестов ЦУ с подтверждением состояний

В предыдущем разделе описан двухуровневый ГА-метод построения проверяющих тестов, который на нижнем уровне использует один из одноуровневых методов раздела 2, в частности метод верификации эквивалентности двух заданных ЦУ - А2.8 (раздел 2.5).

Однако построение двухуровневого ГА-метода генерации тестов возможно с использованием на нижнем уровне метода достижения состояний в ЦУ (раздел 2.4). Кроме иллюстрации технической возможности такого конструирования данный двухуровневый метод имеет и практические предпосылки, которые заключаются в следующем.

В детерминированных алгоритмах построения тестов [23, 140-141] каждая неисправность первоначально должна быть активирована, а её влияние необходимо транспортировать на внешние выходы анализируемого ЦУ. При этом в последовательностных ЦУ это может потребовать от устройства находиться в некотором определённом состоянии для того, чтобы такая активация/распространение были возможны. Для этого требуется разработка процедур обратного распространения, которые для логического уровня представления последовательностных ЦУ усложняются двумя факторами:

- обратное распространение может потребоваться для нескольких тактов модельного времени;
- для многих логических элементов возникает проблема неоднозначности значений входов, заставляющая рассматривать множество вариантов.

Дополнительные ограничения могут возникать из технологических особенностей: частичное или полное использование сканирующих регистров для установки начального состояния, предопределённость начального состояния, ограничение на число тактов для достижения состояния и т.д.

В противоположность обратному распространению в подходах идентификации основанных на моделировании [142] влияние неисправности всегда распространяется только в прямом направлении. Оно заключается в генерации пробной последовательности и моделировании поведения ЦУ путём её приложения ко входам устройства. К этой же группе относятся методы, которые основаны на ГА [143, 48]

– раздел 3.1.

Детерминированные алгоритмы чаще используются в схемах-контроллерах, тогда как подходы основанные на моделировании - для устройств обработки данных.

Развитие сложных комплексных СБИС потребовало объединения этих двух направлений. В настоящее время известны методы построения тестов, которые совмещают детерминированный и основанный на моделировании подходы. Например, в [55] метод начинает работу с ГА. Если же в течении определённого времени не происходит улучшения ситуации, то алгоритм переключается на детерминированный метод построения тестов.

Другим способом объединения двух указанных подходов является подтверждение состояний ЦУ при активизации неисправностей в детерминированных методах, которое реализуется, в свою очередь, основанным на моделировании алгоритмом.

Покажем конструктивно построение такого ГА-метода генерации тестов.

Постановка задачи совпадает с разделом 3.2, рассматриваются одиночные константные неисправности.

Описываемый метод построения тестов имеет с методом раздела 3.2 общую стратегию верхнего уровня, которая заключается в выборе из списка непроверенных неисправностей F' одной целевой неисправности $f_{цел}$, для которой на нижнем уровне будет построен тест. Таким образом блок-схема верхнего уровня разрабатываемого метода соответствует приведённой на рис.3.1, а его реализация происходит на основании метода А3.2. Изменения буду касаться процедуры выбора целевой неисправности. Поскольку фаза активизации неисправности переместится на нижний уровень, то здесь в качестве целевой выбирается любая непроверенная к текущему моменту времени неисправность $f_{цел} = f' \in F'$.

Для иллюстрации места ГА в методе построения теста одной неисправности нижнего уровня рассмотрим рис.3.2. Здесь изображены три такта времени работы ЦУ: $t-1$, t, $t+1$. Полностью поведение комбинационного блока (КБ) в произвольный момент времени определяют значения на его входах X и псевдовходах Z. Для КБ в момент времени модельного t это векторы X_t и Z_t.

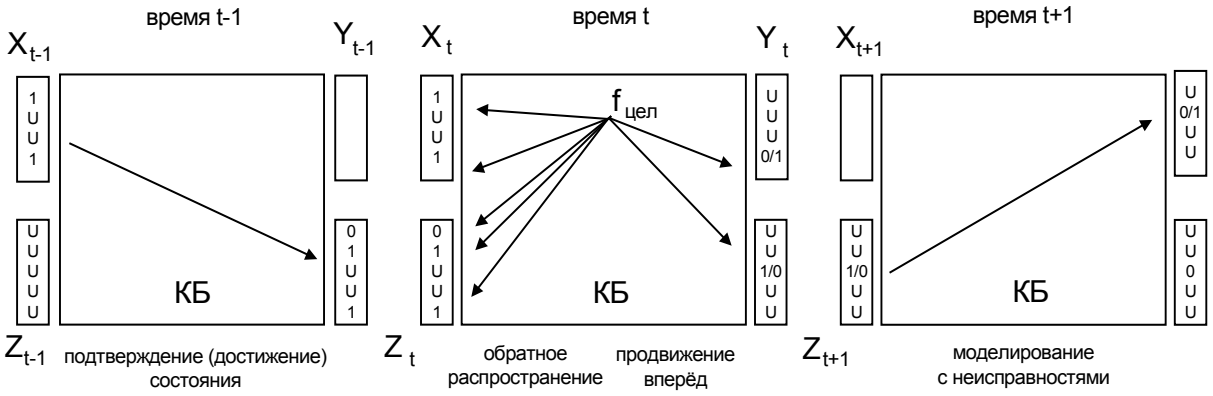


Рис.3.2. Принцип работы метода построения тестов с подтверждением состояний.

Начальным на рисунке является момент времени t. Именно в этот такт происходит внесение влияния неисправности $f_{цел}$. В данном такте времени t реализуется метод существенного пути построения тестов [29]. Здесь выполняются две следующие фазы работы данного метода.

Обратное распространение. Оно выполняется исходя из условия проявления неисправности: на линии в месте неисправности необходимо обеспечить различные значения в исправном A_0 и неисправном A_f устройствах. При этом будут получены некоторые значения (возможно частично определённые) для входов и псевдовходов ЦУ. Для примера на рисунке имеем: $X_t = (1,u,u,1)$ и $Z_t = (0,1,u,u,1)$.

Реализация фазы обратного распространения осуществляется любым соответствующим структурным методом. Поскольку обработка ЦУ производится в условиях частичной неопределенности в алфавите E_3, то может быть применён метод для частично определённых булевых функций, например, из [144].

Продвижение вперёд: необходимо выполнить продвижение различия поведения исправного и неисправного ЦУ на внешние выходы/псевдовыходы. Для нашего примера имеем: $X_{t+1} = (u,u,u,0/1)$ и $Z_{t+1} = (u,u,1/0,u,u)$. Здесь 0/1 для краткости обозначает значение сигнала на линии в исправном A_0 и неисправном A_f устройствах.

Фаза продвижения вперёд может потребовать несколько тактов

модельного времени для распространения различия на внешние выходы схемы в соответствии с определением 3.1. Дополнительного продвижения вперёд в тактах времени $t+1$ и далее может не потребоваться в том случае, если в момент времени t различие в поведении исправного ЦУ A_0 и неисправного A_f распространилось на внешние выходы. На рисунке $X_{t+1} = (u,u,u,0/1)$ и, следовательно, нет необходимости выполнять дополнительное распространение вперёд для времени $t+1,...$ Если бы в векторе X_{t+1} не было различных значений для A_0 и A_f, то дополнительное продвижение вперёд необходимо (как это показано на рис.3.2). На рисунке изображён только один дополнительный такт $t+1$. Продвижение вперёд также может уточнять значения векторов X_t и Z_t.

Центральным местом метода является подтверждение состояний Z_t, полученного в фазе обратного распространения. Оно заключается в том, чтобы получить заданное состояние Z_t в паре устройств A_0 и A_f, стартуя из полностью неопределённого состояния $Z_u = (u,u,...,u)$.

Вообще говоря, построить последовательность S, переводящую устройства A_0 и A_f в состояние Z_t из Z_u можно также с помощью процедуры обратного распространения, применяя её итеративно для тактов времени $t-1$, $t-2$ и т.д. до выполнения условия $Z_{t-i} = Z_u$. Однако сложность здесь заключается именно в том, что обратное распространение выполняется через несколько тактов модельного времени. Как отмечалось ранее, структурные методы в этом случае не позволяют обрабатывать большие ЦУ.

С другой стороны, задача подтверждения состояния в рассматриваемом методе фактически является задачей построения ПДА (раздел 2.4). В этом случае параметрами данного ГА-метода являются: $Z_{нач} = Z_u$, $Z_{кон} = Z_t$.

Таким образом, в рассматриваемом методе построения тестов для достижения одной цели, определённой на верхнем уровне, вызываются два различных метода на нижнем уровне: метод существенного пути и ГА-метод достижения состояния. Поэтому описываемый метод мы называем гибридным двухуровневым.

Для описания метода будем считать, что у нас имеется процеду-

ра, реализующая метод существенного пути, в которой выполняются этапы продвижения вперёд и продвижения назад. Они заключаются в том, что для заданного устройства A_0 и неисправности в нём $f_{цел}$ порождаются входные наборы X_t и $Z_t \neq Z_u$, обеспечивающие распространение влияния неисправности на внешние выходы устройства. Известно [29, 1], что в методах данного класса при выполнении условий транспортировки влияния неисправности возможно формирование нескольких альтернатив для значений сигналов на некоторых линиях ЦУ. С другой стороны, при выполнении процедуры обратного распространения возможны противоречия в формировании таких значений, которые необходимы для выполнения условий активизации и распространения. В этом случае в методе происходит откат до места выбора последней альтернативы. Таким образом, результатом рассмотрения каждой такой альтернативы являются входной набор X_t и состояние $Z_t \neq Z_u$ в том случае, если удалось произвести распространение влияния неисправности на внешние выходы ЦУ. В противном случает метод существенного пути возвратит признак непроверяемости неисправности. Будем считать, что реализация метода существенного пути формирует множество таких альтернатив, которые доступны для рассмотрения далее. Таким образом в описываемом методе построения тестов на нижнем уровне формируется цикл по всем возможным таким альтернативам, представленным в виде наборов (X_t, Z_t).

Тогда работу метода на нижнем уровне можно представить в виде следующего псевдокода.

Алгоритм А3.5

```
Гибридное_ПостроениеТеста( A0 , fцел )
{
  МетодСущественногоПути( A0 , fцел );
  while( есть нерассмотренные состояния  Zt ≠ Zu  )
  {
    // вызов ГА-метода раздела 2.3
    ГА_Достижение_Состояния( A0 , Zu , Zt );
    if( последовательность построена )
    {
      S =СформироватьТест();
```

```
      ДобавитьВТест(S);
      return; // тест построен-возврат на верхний уровень
    }
    else
      continue; // переход к выбору другой альтернативы
  }//конец есть варианты в методе существенного пути
  // если все варианты рассмотрены, а тест не построен
  ОтметитьКакНепроверяемую(f_цел);
  return;
} // конец алгоритма
```

Процедура *МетодСущественногоПути()* выполняет в нашем методе фазу продвижения вперёд и обратного распространения для заданной неисправности и формирует множество альтернатив (X_t, Z_t).

При рассмотрении каждой из альтернатив происходит вызов ГА-метода достижения состояний. Условием его вызова является тот факт, что для активизации неисправности необходимы определённые значения в векторе состояния ЦУ Z_t. Это выполняется при условии $Z_t \neq Z_u$. Отличием данной процедуры от метода раздела 2.4 является то, что построенная ПДС должна обеспечивать необходимые состояния как в исправном A_0, так и в неисправном A_f ЦУ.

Если ГА построил ПДС для Z_t, то формируется тестовая последовательность S для $f_{цел}$ (функция *СформироватьТест()*). Тест состоит из следующих частей:

- последовательность достижения состояния для Z_t;
- входной набор X_t;
- последовательность распространения влияния неисправности.

Если рассмотрены все альтернативы и тест для неисправности $f_{цел}$ не построен, либо множество альтернатив было пусто, то данная неисправность отмечается как непроверяемая (функция *ОтметитьКакНепроверяемую()*) и не рассматривается в дальнейшем на верхнем уровне метода.

В том случае, если метод не построил тестовую последовательность для некоторой неисправности f_i, это не говорит о том, что такая последовательность не существует на самом деле. Поскольку нижний уровень метода фактически состоит из двух алгоритмов, то

каждый из них может стать причиной отрицательного результата:

- метод существенного пути не позволил выполнить распространение влияния неисправности ввиду собственных ограничений;
- ГА не построил ПДС ввиду вероятностного характера, либо недостаточной глубины поиска.

Если произвести сравнение данного метода построения тестов и метода из раздела 3.2, то хорошо видно место ГА-методов построения последовательностей. Рассмотрим рис.3.2. Его можно рассматривать как изображение обобщённого подхода к решению задачи построения теста одной неисправности $f_{цел}$ для последовательностных ЦУ, который включает фазы активизации неисправности $f_{цел}$ и распространения её влияния на внешние выходы.

В двухуровневом ГА-методе построения тестов (раздел 3.2) происходит псевдослучайная генерация последовательностей, активизирующих неисправность. Т.е. такая генерация должна построить состояние Z_{t+1} (в такте времени t на рисунке) с различными значениями в исправном и неисправном ЦУ, что выполняется в такты времени до t включительно. Дальнейшее распространение влияния неисправности происходит с помощью процедуры ГА. Таким образом ГА работает в тактах времени $t+1$ и далее на рисунке.

В гибридном методе, который рассматривается в данном разделе, активизация и распространение влияния неисправности происходит с помощью метода существенного пути: такты времени t, $t+1$ и т.д. Тогда как ГА необходим для достижения состояния Z_t и работает по такт времени $t-1$ включительно.

Рассматриваемый метод использует 3-значный алфавит моделирования E_3 и, следовательно, одномерную активизацию неисправности. Очевидно, что ещё большей эффективности метода можно достичь при использовании многозначных алфавитов. Использование шестизначного алфавита T_6 позволит проводить многомерную активизацию путей распространения неисправностей, например, с помощью D-алгоритма. Использование многозначных логик является одним из перспективных направлений при построении подобных методов.

3.4. Двухуровневый ГА-метод построения диагностических тестов ЦУ

Задача построения диагностических тестов (ДТ) рассматривается гораздо реже, чем задача построения проверяющих тестов. Это связано с тем, что задача являет существенно более сложной. В [145] показывается, что данная задача эквивалентна нахождению максимальной клики заданного графа, которая является NP-полной.

Как и в случае задачи построения проверяющих тестов, разработанные алгоритмы имеют ограничения по практической применимости. Например, в [146] описан подход, который гарантирует построение диагностической последовательности, если она существует. Однако для схем большой размерности данные методы не в состоянии поострить полное дерево решений ввиду его размера и ограничений памяти, что делает их неприменимыми для таких схем.

В данном разделе будет показано, как данная задача может быть решена с помощью ГА [147]. При этом ограничения на размер схемы будут сняты ввиду отсутствия необходимости строить решающие деревья большой размерности.

В качестве модели будем использовать синхронные последовательностные ЦУ, а в качестве моделей неисправностей выберем одиночные константные неисправности. Замечания о возможности распространения метода на другие типы неисправностей будут даны далее.

Определение 3.6. Пусть задано исправное ЦУ A_0 и класс неисправностей $F = \{f_1,..., f_n\}$, который соответственно порождает класс неисправных устройств $A = \{A_1,..., A_n\}$. Входная последовательность S, которая может отличить поведение произвольного устройства A_i от поведения исправного A_0, а также поведения всех остальных неисправных устройств, называется диагностической.

Фактически, диагностическая последовательность позволяет разбить весь класс неисправностей на классы эквивалентных неисправностей, при это мощность каждого такого класса равна единице [146, 148]. Задачу в такой постановке также часто называют задачей о ло-

кализации неисправностей, подчёркивая тот момент, что выделение неисправности в отдельный класс неотличимости точно указывает на место в ЦУ (линию в схеме) возникновения неисправности.

Дадим несколько определений, на основе которых будет строиться метод решения задачи.

Определение 3.7. Для заданного устройства A_0 и двух неисправностей f_1 и f_2 последовательность S называется различающей, если выходные реакции $A_1(S)$ и $A_2(S)$ различны хотя бы для одного входного вектора.

Определение 3.8. Все неисправности $f_1,...,f_n$, которые не различаются заданной входной последовательностью S для заданного устройства A_0, принадлежат к одному классу неразличимости относительно последовательности S.

Определение 3.9. Для заданного устройства A_0 и заданного класса неисправностей $F=\{f_1,...,f_n\}$ последовательность S называется различающей, если существует хотя бы одна неисправность $f_j \in F$ такая, что $A_j(S) \neq A_i(S)$ (для $i=\overline{1,n}$ кроме $i=j$) хотя бы для одного входного вектора:

$$\exists f_j \in F : A_j(S) \neq A_i(S), i=\overline{1,n}, i \neq j. \qquad (3.4)$$

Последнее определение показывает, что при существовании последовательности S с указанными свойствами, класс неисправностей F разбивается на два класса неразличимых неисправностей: в первый класс входят все неисправности A_j, для которых выполнено условие в определении, во второй класс неотличимости – все остальные неисправности. Часто такую последовательность S называют разбивающей, поскольку она делит один класс неотличимых неисправностей F на несколько. Неформально такую последовательность также можно назвать диагностической. Также последнее определение является конструктивным, поскольку показывает, выполнение какого свойства позволят назвать последовательность диагностической.

Именно на основании определения 3.9 будет строиться наш метод.

Цель метода – построение такой входной последовательности, которая способна разбить исходный класс одиночных константных неисправностей $F = \{f_1,...,f_n\}$ на максимальное число классов неразличимых неисправностей.

Вначале работы метода все неисправности принадлежат одному классу неразличимости: $f_i \in F, i = \overline{1,n}$. Необходимо построить различающую последовательность, которая позволяет разбить этот класс на два или более класса неразличимых неисправностей $F_1,...,F_j$.

Обозначим через F' множество всех классов неразличимых неисправностей для текущей итерации алгоритма: $F' = \{F_1,...,F_j\}$. Далее необходимо строить различающие последовательности для каждого класса $F_k \in F'$. На конечном этапе работы алгоритма последовательности будут строиться для классов, которые содержат только по две неисправности: $|F_k| = 2$.

Рассмотрим пример, приведённый на рис.3.3. В начале работы алгоритма все неисправности принадлежат одному множеству неразличимости $F_0 = \{f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8\}$. Поскольку в этот момент оно является единственным, то множество F' содержит единственный элемент: $F' = \{F_0\}$.

Пусть далее удалось построить различающую последовательность S_1, которая разбила F_0 на два множества, которые соответственно содержат: $F_1 = \{f_1, f_4, f_5\}$ и $F_2 = \{f_2, f_3, f_6, f_7, f_8,\}$. Тогда на этом этапе множество F' будет содержать два элемента: $F' = \{F_1, F_2\}$, а итоговый тест – одну последовательность: $S = \{S_1\}$.

После этого для множества F_1 удалось построить различающую последовательность S_2, которая также разбила его на два следующих: $F_3 = \{f_1\}$ и $F_4 = \{f_4, f_5\}$.

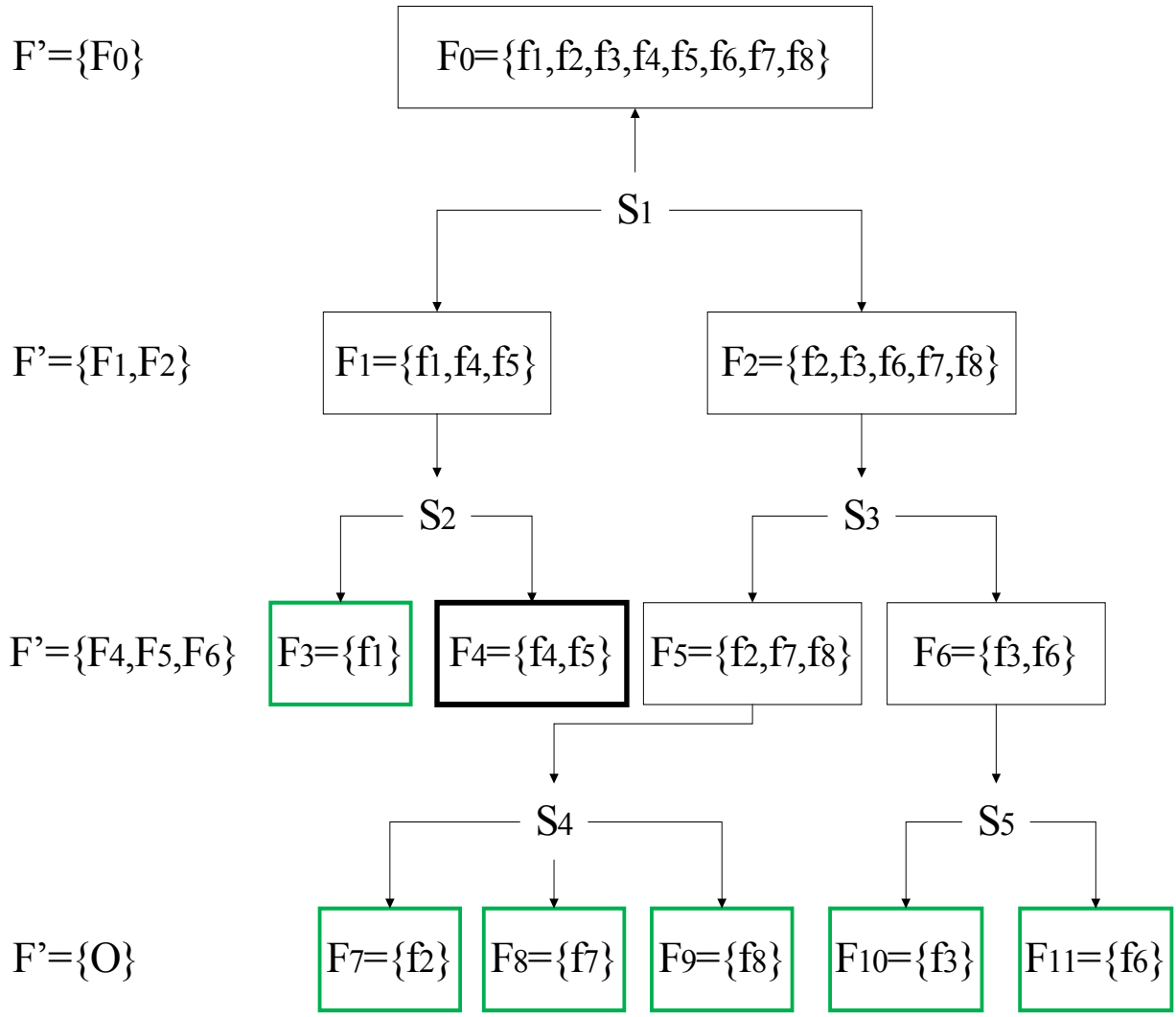


Рис.3.3. Пример построения дерева решений в методе генерации диагностических тестов.

В свою очередь для множества F_2 построенная последовательность S_3 является различающей, разбивая его на два следующих множества: $F_5 = \{f_2, f_7, f_8\}$ и $F_6 = \{f_3, f_6\}$. Тогда на этом этапе множество F' содержит три элемента: $F' = \{F_4, F_5, F_6\}$. Сюда не следует включать элемент $F_3 = \{f_1\}$, поскольку он содержит только одну неисправность. Последовательности S_2 и S_3 добавляются в итоговый тест: $S = \{S_1, S_2, S_3\}$.

Продолжая рассуждения таким образом, предположим, что для

множеств F_5 и F_6 удалось построить различающие последовательности S_4 и S_5 соответственно, причём данные классы неисправностей были разбиты на классы мощности 1: $F_7 = \{f_2\}$, $F_8 = \{f_7\}$, $F_9 = \{f_8\}$, $F_{10} = \{f_3\}$, $F_{11} = \{f_6\}$. В дереве решений данные множества будут являться листьями.

С другой стороны, для множества неисправностей F_4 построить различающую последовательность не удалось – на рис.3.3 отмечено жирным шрифтом.

Результатом работы алгоритма будет итоговый диагностический тест $S = \{S_1, S_2, S_3, S_4, S_5\}$. При его приложении будут построены следующие классы:

- шесть классов неразличимых неисправностей мощности один: $F_3 = \{f_1\}, F_7 = \{f_2\}$, $F_8 = \{f_7\}$, $F_9 = \{f_8\}$, $F_{10} = \{f_3\}$, $F_{11} = \{f_6\}$;
- один класс неразличимых неисправностей мощности два: $F_4 = \{f_4, f_5\}$.

В данном примере видно, что итоговый тест не является полным, поскольку не все итоговые классы содержат ровно одну неисправность.

Таким образом, на практике результатом работы метода является диагностический тест, который состоит из набора разбивающих последовательностей $S = \{S_1, ..., S_m\}$, с помощью которых полный список неисправностей F разбивается на неразличимые классы $F_1, ..., F_p$, каждый из которых может содержать более одной неисправности: $\exists j : \left|F_j\right| \neq 1$. Последнее означает, что алгоритму не удалось построить разбивающую последовательность для неисправностей, которые входят в данный класс F_j.

Структурно описываемый метод представляет собой итеративный процесс генерации диагностических последовательностей для некоторого текущего выбранного множества неразличимых неисправностей $F_j \in F'$. Таким образом, структура алгоритма соответствует двухуровневой схеме применения алгоритмов ГА (рис.2.2). При этом целевое множество F_j определяет объект *Цель* для шаблона А3.1. Всякий раз, когда удаётся построить различающую последова-

тельность для F_j, она добавляется в финальный тест.

Каждая итерация алгоритма состоит из трёх фаз.

1) Выбор целевого класса F_j из текущего множества классов неразличимости F' – фаза 1 алгоритма, представляющая верхний уровень метода.
2) Построение с помощью ГА, по возможности, различающей последовательности для выбранного класса F_j - фаза 2 алгоритма соответствует нижнему уровню метода.
3) Дополнительная фаза 3 алгоритма, реализуемая также на верхнем уровне, и которая заключается в следующем. Если в фазе 2 нижнего уровня метода построена различающая последовательность S для целевого класса F_j, то выполняется моделирование для других классов $F_l \in F', l \neq j$ на данной последовательности S, чтобы проверить возможность того, что она также является разбивающей и для них.

Видно, что такая структура метода в общем виде повторяет структуру алгоритма генерации тестов для одиночных константных неисправностей (раздел 3.2), в которой также выделялись 3 фазы работы. Хотя в данном случае нагрузка семантика понятия *Цель* двухуровневого метода и, следовательно, строящихся последовательностей существенно иная.

Укрупнённый псевдокод верхнего уровня метода приведён ниже.

Алгоритм А3.6.

```
ГенерацияДиагностических
Тестов_ВерхнийУровень(A0, Параметры)
{
  F =ПостроениеНачальногоСпискаНеисправностей(A0);
  F'= F ;
  while( не_достигнут_критерий_остановки() )
  {
    //фаза 1-выбор целевого множества-случайная генерация
    Fj=ВыбратьЦелевоеМножество(A0, F',параметры);
    if(Fj==NULL )
      return;//цель не выбрана-конец алгоритма
    else
    {
```

```
      // фаза 2 – генетический алгоритм
      Si=ГА_построения_ДП(A0,Популяция,Fj);
      if( Si диагностическая для Fj )
      {
        F'=ОбновитьМножество(Fj,F');
        ДобавитьВТест(Si);
        // фаза 3 – дополнительная проверка
        ДополнительнаяПроверкаДругихМножеств(Si,F');
      }// конец if – вызов Фазы 3 алгоритма
      else
        ОтметитьКакНеразличимое(Fj);
    }  // конец if – выбрана целевое множество
  }    // конец while – не достигнут критерий остановки
}      // конец алгоритма
```

Дадим пояснения к псевдокоду верхнего уровня метода.

Фаза 1 метода представлена процедурой *ВыбратьЦелевоеМножество()*. В ней происходит выбор целевого множества $F_j \in F'$, для которого будет строиться различающая последовательность. Одновременно с этим строится начальная популяция особей для фазы 2 метода. Работа функции будет описана ниже.

Если выбрать множество $F_j \in F'$ не удалось, то алгоритм заканчивает работу.

Функция *ГА_построения_ДП()* реализует вызов ГА-метода построения диагностической последовательности для множества неисправностей F'. Работа ГА построения такой последовательности будет описана ниже.

Если такая последовательность построена, то по результатам диагностического моделирования следует обновить множество F' (функция *ОбновитьМножество()*). Если диагностическое моделирование показывает, что F_j распалось на множества, в которых есть такие подмножества, мощность которых больше единицы (содержат более чем одну неисправность), то они должны быть включены в F'. Множества с мощностью 1 содержат одну неисправность и далее не рассматриваются.

После этого вызывается функция *ДобавитьВТест()* и далее *ДополнительнаяПроверкаДругихМножеств()*, которая реализует фазу 3

алгоритма, являющуюся дополнительной.

Если же для данного множества F_j не удалось построить различающую последовательность, то предполагается, что ГА не смог разбить его на более мелкие подмножества. Для того, чтобы ГА не вызывался далее для множества F_j, для него в функции *ОтметитьКакНеразличимое()* устанавливается соответствующий признак.

Фазы 1-3 метода повторяются итеративно, но не более чем заранее заданное число раз, которое задаётся в качестве параметра.

Опишем работу метода в фазе 1. Выбор некоторого множества $F_j \in F'$ в качестве целевого является нетривиальной задачей. К настоящему времени не предложено эффективных (ни детерминированных, ни эвристических) методов такого выбора. Поэтому выбрана стратегия с использованием функции оценки (см. далее) фазы 2 метода. Она заключается в следующем.

В фазе 1 производится генерация набора случайных входных последовательностей $Pop = \{S_i\}$, которое также можно рассматривать как популяцию. Далее производится диагностическое моделирование на каждой из данных последовательностей для всех классов неразличимости, входящих в F_j на текущий момент. Для каждого из них происходит вычисление оценочной функции $O(A_0, F_j, S_i)$ (3.7). Если в данном месте ни для одной из последовательностей не получено значение, которое превышает некоторое пороговое: $O(A_0, F_j, S_i) < O_{пор}$, то происходит генерация нового множества последовательностей $\{S_i\}$. Такая попытка производится не более чем предопределённое число раз. Если по окончании заданного числа итераций такой последовательности не найдено, то весь алгоритм завершает работу.

Если же в некоторый момент времени найдена последовательность S_i и множество неразличимых неисправностей $F_j \in F'$, для которых выполняется $O(A_0, F_j, S_i) > O_{пор}$, то такая неисправность выбирается в качестве *Цели* для нижнего уровня метода (фазы 2), а множество последовательностей $\{S_i\}$ передаётся туда в качестве начальной популяции $Pop_{нач}$.

Псевдокод процедуры выбора целевого множества F_j, реали-

зующей описанную стратегию, приведён ниже.

Алгоритм А3.7

```
ВыбратьЦелевоеМножество(A0, F', Параметры)
{
  L = Lнач;
  while( НеДостиглиМаксимальногоЧислаИтераций() )
  {
    // заполнить случайными последовательностями длины L
    Pop = СлучайнаяГенерация(L);
    // цикл по всем элементам множества F'
    for( int j=0 ; j<|F'| ; j++)
    {
      Fj = ВыбратьНерассмотренноеМножество(F');
      for( int k=0 ; k< Nособ ; k++ )
      {
        O = ДиагностическоеМоделирование(A0, Fj, Pop[k]);
        if( O > Oпор )
        {
          Popнач = Pop;
          return Fj;
        } // конец if – множество выбрано
      }   // конец цикла по особям в популяции
    }     // конец цикла по множеству F'
    L = УвеличитьДлину(L);
  }       // конец цикла по итерациям
  return NULL; //итерации закончились, а выбор не сделан
}         // конец процедуры фазы 1
```

В данной процедуре (как и на верхнем уровне метода раздела 3.2) мы не используем генетические операции, а термин *Популяция* используется только для обозначения множества последовательностей.

Непосредственно поиск решения с помощью ГА реализуется в фазе 2, которая формирует нижний уровень метода.

Входными данными для метода являются описание ЦУ A_0, множество F_j, для которого строится ДП, а также начальная популяция $Pop_{нач}$, построенная в фазе 1 метода.

Для построения ГА зададим его компоненты. Кодирование осо-

бей и популяций, а также генетические операции соответствуют предыдущим ГА-методам построения ИдП.

Наиболее сложным в данном методе является построение оценочной функции. Очевидно, что для некоторой входной последовательности S и класса неразличимых неисправностей F_j достаточно легко определить является ли S диагностической (определения 3.6, 3.9), т.е. разбивает ли она данный класс как минимум на два класса неразличимости. Для этого достаточно выполнить диагностическое моделирование поведения заданного ЦУ A_0 для всех неисправностей, которые входят в F_j. Однако оценить, насколько близко данная последовательность S подошла к решению такой задачи, гораздо труднее. Именно в этом и состоит трудность построения оценочной функции данного алгоритма.

В методах, которые рассмотрены в данной и предыдущих главах, оценочная функция заданной последовательности S строилась на основании параметров различающей активности (2.27)-(2.29) устройств: чем больше различных значений на контрольных множествах в двух сравниваемых ЦУ, тем выше оценка данной входной последовательности. Выберем аналогичный подход и при решении данной задачи.

Общая оценка последовательности S строится как сумма оценок каждого входящего в неё вектора X_i:

$$O(A_0, F_j, S) = \sum_{i=1}^{длина(S)} O(A_0, F_j, X_i) =$$
$$= \sum_{i=1}^{длина(S)} (c_1 \cdot N_1(X_i, A_0, F_j) + c_2 \cdot N_2(X_i, A_0, F_j) + \tag{3.7}$$
$$+ c_3 \cdot N_3(X_i, A_0, F_j)),$$

где:

- X_i - i-й набор последовательности S;
- c_1-c_3 - нормализующие константы;
- N_1-N_3 - параметры различающей активности.

Видно, что такое построение оценки в целом соответствует ГА-

методу верификации эквивалентности заданных ЦУ (2.26) и ГА-методу построения теста заданной неисправности (3.3). Однако, поскольку здесь происходит сравнение поведения не двух заданных ЦУ, а фактически целого множества ЦУ, задаваемого множеством неисправностей F_j, то параметры N_1, N_2 и N_3, несут другую смысловую нагрузку.

N_1 - число внешних выходов с различными значениями в неисправных ЦУ, соответствующих неисправностям, входящим в текущий класс неразличимости F_j:

$$N_1(X_i, A_0, F_j) = \sum_{g \in Y} R(g, X_i, A_0, F_j), \tag{3.8}$$

где Y - множество внешних выходов ЦУ.

N_2 - число элементов состояний с различными значениями в неисправных ЦУ, которые соответствуют неисправностям, входящим в текущий класс неразличимости F_j:

$$N_2(X_i, A_0, F_j) = \sum_{g \in Z} R(g, X_i, A_0, F_j), \tag{3.9}$$

где Z - множество линий состояний ЦУ.

N_3 - число выходов вентилей (комбинационных блоков, контрольных точек) с различными значениями в неисправных ЦУ, которые соответствуют неисправностям, входящим в текущий класс неразличимости F_j:

$$N_3(X_i, A_0, F_j) = \sum_{g \in G} R(g, X_i, A_0, F_j), \tag{3.10}$$

где G - множество линий комбинационных блоков ЦУ.

Соответственно, функции различия $R()$ в (3.8)-(3.10) должны быть переопределены для множества сравниваемых ЦУ. Функции различия элементов для множества ЦУ определяются следующим образом:

$$R(g, X_i, A_0, F_j) = \begin{cases} 0, \textit{ если выходы элементов } g \textit{ множества ЦУ}, \\ \textit{которые принадлежат класса } F_j, \textit{ одинаковы} \\ \textit{после подачи набора } X_i; \\ 1, \textit{ иначе.} \end{cases} \quad (3.11)$$

Легко видеть, что функции различия $r()$ двух устройств (2.14)-(2.16) являются частными случаями функции различия $R()$ для множества устройств, когда верифицируется поведение пары ЦУ. Легко можно показать, что функции различия $R()$ выражаются через функции достижения значений U^0 и U^1.

Можно видеть, что функции оценки (2.26) в ГА-методе верификации эквивалентности и (3.1) в методе построения проверяющих тестов являются частными случаями оценки в виде (3.7).

Также (3.7) отличается от (3.3) следующим моментом. В функции оценке (3.3) отсутствует член, показывающий различие на внешних выходах двух ЦУ. Это связано с тем, что такая информация показывает, что задача решена: последовательность порождает различные выходные реакции в двух верифицируемых ЦУ. Такие данные всегда доступны после моделирования. С другой стороны такая информация является бесполезной в том смысле, что она не показывает насколько последовательность приблизилась к решению задачи.

Однако для функции оценки (3.7) информация о различии выходных реакций множества устройств N_1 является существенной: чем больше различных устройств – тем качество последовательности выше.

По аналогии с (3.2) оценку последовательности S можно построить, выбирая максимальное значение оценки, которое достигнуто для некоторого её вектора:

$$O(A_0, F_j, S) = \max_i O(A_0, F_j, X_i). \quad (3.12)$$

Поскольку ГА построения диагностического теста заданного множества F_j строится по одноуровневой схеме, то используемые генетические операции (селекция, скрещивание, мутация) соответст-

вуют описанным в разделе 2.2.

При алгоритмической реализации метода используется схема с промежуточной популяцией. Для её построения из основной популяции итеративно выбираются пары родителей, над которыми с заданными вероятностями $P_{скр}$ (для операции скрещивания) и $P_{мут}$ (для мутации) выполняются соответствующие операции. Результирующие особи-потомки помещаются в промежуточную популяцию. Также используется стратегия элитизма, причём в популяции обновляется 80% особей.

В виде псевдокода описанный ГА-метод построения ДП заданного множества неразличимых неисправностей приведён ниже.

Алгоритм А3.8.

```
ГА_построения_ДП( A0 , Pop_нач , F_j )
{
  i = 0 ; //счётчик числа поколений
  Pop_i = Pop_нач ;
  ОценитьПопуляцию( A0 , F_j , Pop_i );
  while( не_достигнут_критерий_остановки() )
  {
    Pop_пром =∅;
    Позиция=0;
    for( int j=0 ; j<ЧислоНовыхОсобей ; o++ )
    {
      S_A =ВыбратьРодителя( Pop_i );
      S_B =ВыбратьРодителя( Pop_i );
      if( rand() > P_скр )
        S_пот =ВыполнитьСкрещивание( S_A , S_B );
      else
        S_пот = S_A ;
      if( rand() > P_мут )
        S_пот =ВыполнитьМутацию( S_пот );
      ДобавитьОсобьВПопуляцию( Pop_пром , S_пот , Позиция);
      Позиция++;
    } // конец for – построение промежуточной популяции
    for( каждой особи S_k в Промежуточная )
      Вычислить E(S_k , F_j );
```

```
    ПостроитьНовуюПопуляцию( Pop_i ,  Pop_пром );
    for( каждой особи S_k в Популяции )
    {
      ВыполнитьДиагностическоеМоделирование( S_k , F_j );
      if( S_k диагностическая для F_j )
        return S_k;
    };
    i++;
  } // конец while – достигнут критерий остановки
  return NULL;
```

В данном псевдокоде переменные несут следующий смысл:

- A_0- заданное ЦУ;
- F_j - целевое множество неразличимых неисправностей, для которого ГА строит ДП;
- S_A, S_B и $S_{пот}$ - последовательности-родители и последовательность-потомок соответственно;
- $P_{скр}$ и $P_{мут}$ - вероятности применения операторов скрещивания и мутации соответственно;
- $Pop_{нач}$, Pop_i и $Pop_{пром}$ - начальная, текущая и промежуточная популяции.

Функции в псевдокоде (*ПостроитьНовуюПопуляцию()*, *ВыполнитьСкрещивание()*, *ВыполнитьМутацию()* и т.д.) имеют стандартное наполнение.

Результатом работы разрабатываемого ГА-метода являются два списка:

- список списков множеств с мощностью 1; здесь хранятся полностью отличимые неисправности;
- список F' списков F_j мощностью >1; каждый такой список содержит неразличимые неисправности.

Для такой структуры в [149] предложена мера, которая определяет множества эквивалентных неисправностей относительно заданной тестовой последовательности S, т.е. показывает число классов неразличимых неисправностей мощности 1, 2 и т.д. В соответствии с данной мерой диагностические качества последовательности S относительно множества неисправностей F для заданного ЦУ A_0 определя-

ет последовательность чисел $\{N^i_{нер}\}$, в которой $i-й$ элемент показывает число классов неразличимости неисправностей мощности i. В такой интерпретации для диагностической последовательности S в идеальном случае (определение 1) длина последовательности числе $\{N^i_{нер}\}$ будет равна 1, а элемент последовательности N^1 будет равен общему числу рассматриваемых неисправностей $N^1 = |F|$, показывая, что поведение каждого неисправного ЦУ отличается от поведения всех остальных при моделировании на последовательности S.

В [150] эта мера ДП обобщена таким образом, чтобы учитывать значение неопределённого сигнала u из алфавита E_3 при моделировании поведения последовательностных ЦУ, которые начинают работу из полностью неопределённого состояния $Z_u = (u,u,...,u)$. В том случае, если в присутствии одной неисправности f_i ЦУ порождает на выходе значение 0 из алфавита моделирования E_3, а в присутствии другой неисправности f_j на том же выходном контакте наблюдается значение $u \in E_3$, то в этом случае нельзя определённо говорить о различимости неисправностей f_i и f_j. На самом деле данное свойство различимости будет зависеть от начального состояния ЦУ: для некоторых начальных состояний такие неисправности будут различимы, а для некоторых – нет. Однако в общем случае гарантировать различимости неисправностей f_i и f_j нельзя. Поэтому делается предположение, что выходные значения 0 и $1 \in E_3$ неотличимы от значения $u \in E_3$. В дальнейшем мера качества ДП вычисляется с учётом данного предположения. Видно, что такое расширение функции различия по множеству устройств $R()$ (3.11) на трёхзначную логику является аналогичным расширению функции различия $r()$ на E_3 в соответствии с (2.33).

Важной особенностью является используемый метод моделирования с неисправностями. Для определения диагностических свойств заданной последовательности используется диагностическое моделирование. В контексте разрабатываемого ГА-метода такое моделирование используется в двух случаях:

- для определения, является ли данная последовательность S_i диаг-

ностической для заданного множества F_j;

- для вычисления оценки последовательности S_i: $O(A_0, F_j, S_i)$, в том числе и в фазе 1 алгоритма.

Обычно для анализа проверяющих свойств заданной последовательности используются адаптированные версии методов моделирования типа PROOFS [17] либо ему подобных [8, 19]. Однако в ГА-методе построения ДП прямое использование таких методов невозможно. Это связано с тем, что указанные методы моделирования ориентированы на быструю проверку полноты заданной входной последовательности. В них происходит сравнение поведения всех неисправных устройств с исправным устройством. При этом, если для неисправности f_l в некоторый момент времени установлено, что она уже проверена данной последовательностью S_i, то она удаляется из списка неисправностей и моделирование поведения неисправного ЦУ для неё далее не производится. Такая проверка является некорректной для диагностического моделирования.

Покажем как метод диагностического моделирования может быть надстроен над методом моделирования с неисправностями. В качестве базового выберем метод, описанный в [8].

В данном методе необходимо внести следующие изменения.

1. Изменения в структуре данных. Разрабатываемый ГА-метод генерации ДП тестов обрабатывает список F', содержащий списки F_j неразличимых неисправностей.
2. Неисправность f_l удаляется из множества F_j только тогда, когда она будет отличена от всех остальных, т.е. образует класс неразличимости с размерностью 1: $F_j = \{f_l\}$;
3. Моделирование поведения ЦУ A выполняется для всех входных векторов X_k последовательности S_j, $k = \overline{1, длина(S_j)}$ и для всех неисправностей из текущего списка: $f_l \in F_j$;
4. После моделирования $k-го$ входного вектора X_k последовательности S_j необходимо сравнить значение на внешних выходах ЦУ A_0 со значениями выходов для всех неисправностей, которые принадлежат одному классу неразличимости: $f_l \in F_j$.

Отметим, что такой подход имеет преимущество над методами, использующими матрицы выходных условий [151, 40, 65] и матрицы различимости [152]. Оно выражается в том, что нет необходимости заполнять соответствующие матрицы, которые для современных разрабатываемых ЦУ имеют очень большую размерность. Фактически, описанное построение списков в диагностическом методе моделирования и исключение из рассмотрения множеств мощности 1, а также множеств, для которых была показана неразличимость, позволяет в дальнейшем не производить для них моделирование и операции сравнения выходных реакций. Это соответствует уменьшению размерности данных матриц, что автоматически ведёт к существенному снижению емкостной и временной сложности метода.

В системе ASMID-Evolution (глава 5) метод диагностического моделирования используется не только для вычисления оценок последовательностей-особей в ГА генерации ДП, но также имеет самостоятельную ценность для определении диагностических свойств заданной входной последовательности. Для этого в разработанный метод диагностического моделирования встроены процедуры вычисления распространённых диагностических метрик.

Стандартным также является замечание о результатах работы ГА-метода. Если для некоторого множества F_j на нижнем уровне не удастся построить различающую последовательность, то, вообще говоря, это может оказаться не так. Результаты работы существенно зависят от глубины поиска в ГА-методе, которая может регулироваться разработчиком путём задания входных параметров метода.

Отметим также связь описанного ГА-метода с обходом по дереву решений. Видно, что описанный подход, фактически, строит обход по такому дереву (рис.3.3), однако особенным образом. Верхний уровень абстракции на основании псевдослучайной генерации выбирает ветвь в таком дереве, по которой будет произведена попытка движения. При этом выбор не обязательно происходит в продолжение движения от предыдущей ветви, а в том направлении, которое кажется наиболее приемлемым с точки зрения функции оценки, вычисленной на верхнем уровне. На нижнем уровне строится различающая последовательность, разбивающая класс на подклассы. В зависимости от успешности построения последовательности может произойти движение по ветви как на один уровень вниз, так и на несколько. Классы

мощности единица являются концом определённой ветви поиска (листьями дерева).

Как и для двухуровневого ГА-метода построения тестов следует сделать замечание об расширении применимости ГА-метода построения ДП на другие типы неисправностей. Из описания выше видно, что используемый тип неисправностей явно входит только в процедуры моделирования, которые вызываются для оценки качества особей-последовательностей и проверки обнаружимости неисправности. Поэтому, если разработчик ЦУ работает с другими типами неисправностей, и в его арсенале присутствуют процедуры их моделирования, то алгоритм может быть легко расширен для работы с ними. Семантика построения оценочной функции останется при этом прежней, тогда как вид функции оценки (3.7),(3.8)-(3.10) и функций разности (3.11) потребует адаптации.

Глава 4

ПАРАЛЛЕЛЬНЫЕ ВЕРСИИ ГЕНЕТИЧЕСКИХ АЛГОРИТМОВ ПОСТРОЕНИЯ ВХОДНЫХ ИДЕНТИФИЦИРУЮЩИХ ПОСЛЕДОВАТЕЛЬНОСТЕЙ

В ГА-методах построения входных ИдП основным моментом является тот факт, что для оценки качества генерируемых решений выполняется моделирование работы ЦУ на заданной последовательности-особи. В зависимости от требуемой точности/скорости работы может использоваться моделирование исправных ЦУ, а также моделирование ЦУ с неисправностями. Процедуры моделирования поведения ЦУ сами по себе требуют значительных вычислительных ресурсов, в частности времени. Итеративный многократный вызов таких процедур приводит к тому, что ГА построения идентифицирующих последовательностей являются достаточно медленными методами, что и является основным их недостатком.

Таким образом, воссоздание поведения ЦУ с заданной точностью путём моделирования его работы является существенным свойством ГА, позволяющим искать решения требуемого качества. С другой стороны, это же свойство существенно замедляет скорость работы таких ГА. Поэтому задача ускорения работы таких ГА в общем, а также непосредственно процедур вычисления оценочных функций, основанных на моделировании, в частности, является одной из центральных при построении алгоритмов данного рода.

Выделим следующие подходы, направленные на ускорение работы ГА построения идентифицирующих последовательностей:

1. Построение параллельных версий ГА без изменения структуры алгоритма, но с организацией параллельного вычисления в процедуре оценки популяции: схема «хозяин-рабочий» ПГА.
2. Построение параллельных версий ГА с изменением его глобальной

структуры: схема «островов» ПГА.

3. Построение параллельных процедур моделирования внутри ГА, при этом структура алгоритма также не изменяется.

В данной главе разрабатывается ряд методов ускорения работы ГА, которые относятся к первому и второму из указанных направлений.

В разделе 4.1 разрабатываются ПГА построенные по схеме хозяин-рабочий для слабопараллельных ВС.

В разделе 4.2 данная схема распространяется на сильнопараллельные ВС с общей памятью и большим числом процессоров.

В разделе 4.3 разрабатывается обобщённая структура РГА, построенного по схеме островов.

Третье направление разработано авторами в [91, 93, 153], однако выходит за рамки настоящего исследования.

Основным отличием методов, разрабатываемых в разделах 4.1 и 4.2, является то, что здесь параллельно работают процедуры оценки отдельных особей. При этом в некоторых алгоритмах построения ИдП процедура оценки одной особи может состоять как из одной процедуры моделирования ЦУ, так и из нескольких.

Исходным при построении параллельных методов, относящихся к каждому из направлений, является доступная аппаратная платформа, на которой работает САПР разработчика. Фактически, различные методы проецируются на доступную аппаратную часть и получают развитие вместе с ней, формируя набор подходов построения параллельных версий ГА. Диапазон аппаратуры ВС, для которой разрабатывались соответствующие методы, включает все современные доступные платформы: от двухядерных рабочих станций до многопроцессорных ВС с общей и распределённой памятью. Исходя из этого, в каждом из указанных направления мы выделяем методы, предназначенные для слабопараллельных систем и многопроцессорных систем. Очевидно, также, что методы, хорошо проявляющие себя для одного типа аппаратуры параллельных ВС, могут оказаться неэффективными для другого, и наоборот.

Далее мы будем выделять следующие типы параллельных систем:

- слабопараллельные с общей памятью, включают рабочие станции с одним центральным процессором, содержащим 2-4 вычислительных ядра;

- сильнопараллельные системы с общей памятью, включают рабочие станции с несколькими центральными процессорами с общим числом вычислительных ядер 8-12;
- вычислительный кластер: независимые рабочие станции, каждая из которых содержит свою локальную оперативную память, связанные высокоскоростными линиями связи.

Другие типы вычислительных сред могут быть построены путём комбинации указанных выше. Накладывание друг на друга различных методов, предназначенных для различных платформ в указанном смысле, позволяет строить новые параллельные методы, которые предназначены для работы на конкретной аппаратной платформе. Такое совмещение может быть эффективным в целом ряде практических ситуаций.

Примером может служить случай, когда параллельная ВС состоит из множества узлов с распределённой памятью, причём каждый узел состоит из двух- или четырёхядерного процессора.

Необходимость построения параллельных версий ГА-методов генерации ИдП, например, для систем с небольшим числом вычислительных ядер иллюстрируется следующим простым экспериментом. На ВС с одноядерным и многоядерным процессорами запускались характерные задачи САПР по обработке ЦУ, в частности, ГА-метод генерации входных тестовых последовательностей. С помощью стандартной утилиты ОС Windows Task Manager наблюдалась вычислительная загрузка ядер процессоров. Для ВС с одноядерным процессором данная загрузка была равна 100%. Для ВС, содержащей процессор с двумя ядрами, большая часть вычислительной работы пришлась на одно ядро (загрузка 100%), тогда как второе ядро процессора практически простаивало, обслуживая в основном запросы ОС и других приложений. Для ВС с четырёхядерным процессором ситуация оказалась ещё хуже (рис.4.1). Ни одно из ядер процессора не имело полной загрузки. На диаграмме видны два ядра с переменной средней загрузкой около 35-40%. Причиной данной ситуации является технология многоядерных процессоров, которая пытается один вычислительный процесс распределить на несколько ядер с целью унификации их загрузки. Таким образом видно, что добавление вычислительных ядер не только не улучшает выполнение существующих алгоритмов, а в некоторых случаях производительность даже несколько падает. Аналогичные результаты авторы наблюдали и при выполнении других

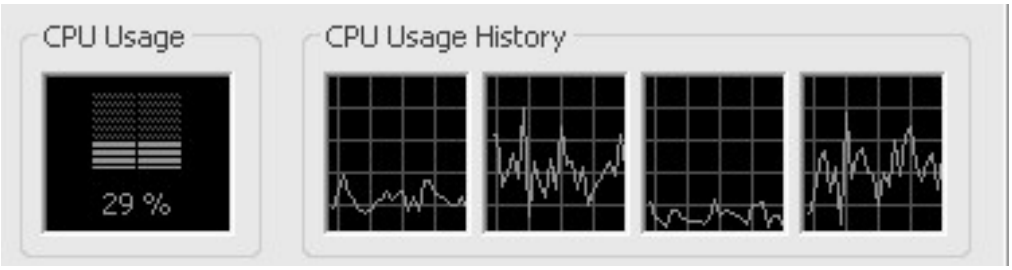


Рис.4.1. Загрузка ядер процессора приложением с одним вычислительным потоком.

программных компонент системы АСМИД, а также иных САПР.

4.1. Схема «хозяин-рабочий» параллельных ГА для слабопараллельных ВС с общей памятью

Схема «хозяин-рабочий» предполагает, что в параллельной версии алгоритма существует только один основной цикл ГА построения популяций, который соответствует процессору «хозяину». Все остальные процессоры называются «рабочими» и они реализуют процедуры вычисления оценочных функций особей. Таким образом, распараллеливанию подвергается процедура оценки популяции, представленная, например, алгоритмом А2.4.

Такая схема построения ПГА является наиболее простой и очевидной. Основное её назначение - повысить скорость работы ГА-методов за счёт организации параллельных вычислений в процедурах оценок особей в популяции на доступных вычислительных ядрах/процессорах системы.

В качестве предпосылок построения такой схемы можно отметить следующие. Из описания структуры ГА в разделе 2 (например, алгоритм А2.1) видно, что данный алгоритм носит итеративный характер. При этом работа на текущей итерации i существенно зависит от результатов, которые получены в результате работы предыдущей итерации $i-1$. Такая последовательная схема работы ГА не позволяет напрямую построить параллельную версию алгоритма. Однако в ГА-методах построения ИдП всё-таки можно выделить достаточно круп-

ные фрагменты, которые являются независимыми и позволяют параллельную реализацию. Это процедуры вычисления оценочных функций. В рассматриваемых ГА-методах эти процедуры являются процедурами моделирования работы ЦУ, что предполагает достаточно высокую вычислительную нагрузку. Лишь небольшая часть времени приходится на процедуры построения новых популяций, которые в принципе не распараллеливаются. Основываясь на этом можно предположить, что распараллеливание вычислений оценок особей в популяции позволит достичь приемлемого масштабирования алгоритма.

Однако эффект повышения быстродействия алгоритма будет проявляться далеко не для каждого алгоритма ГА, в котором организовано параллельное вычисление внутри процедуры оценки популяции. Это связано с тем, что организация такого параллельного вычисления также имеет накладные расходы. Если вычисление оценок особей происходит достаточно быстро (по относительно простым формулам), то накладные расходы на организацию параллельного вычисления будут сопоставимы с временными ресурсами на вычисление оценок особей. В этом случае выигрыша в быстродействии может не быть вообще, либо он будет очень незначителен.

Исходя их этого следует ожидать, что данная схема будет показывать высокую эффективность в алгоритмах построения ИдП.

Далее в данном подразделе описываются методы параллельного вычисления оценок особей в популяции на рабочих станциях с двух- и четырёхядерными процессорами. Системы данного класса имеют достаточно ограниченные возможности по построению структур параллельных версий ГА.

Для распараллеливания процедур вычисления оценочных функций выделим следующие три подхода.

1. Параллельное моделирование поведения ЦУ внутри процедуры оценки одной особи.
2. Параллельное вычисление оценок различных особей внутри процедуры оценки популяции.
3. Смешанный подход, который объединяет два предыдущих.

При реализации на рассматриваемом классе систем все данные подходы основаны на многопоточном программировании. Вычислительный поток содержит исполняемый код одной или нескольких процедур. Каждый вычислительный поток выполняется в системе не-

зависимо от других Для их взаимодействия и синхронизации существуют специальные механизмы. Основными действиями, которые можно производить с потоками являются их создание, запуск на выполнение, приостановка, ожидание окончания и удаление.

Рассмотрим подробнее каждый из подходов.

1. Первый подход [154-156] заключается *в параллельном моделировании поведения нескольких ЦУ внутри процедуры оценки одной особи*.

Такая ситуация, в частности, возникает в методах, где для оценки заданной последовательности-особи необходимо выполнить моделирования поведения двух ЦУ A_1 и A_2. Примерами таких методов являются алгоритм А2.8 и алгоритм А3.4.

Опишем подробнее данный подход на примере алгоритма ГА верификации эквивалентности. Последовательный псевдокод реализации функции *ОценитьОсобь()* представлен алгоритмом А2.9. В нём дважды происходит вызов функции моделирования поведения исправного ЦУ *ИсправноеМоделирование()*.

Видно, что моделирование поведения второго ЦУ A_2 в последовательной версии не начинается до завершения моделирования первого ЦУ A_1. Поскольку для операционной системы данный код принадлежит одному потоку исполнения, то, например, для двухядерного ЦП общая загрузка ядер составит не более 50%.

Сопоставим одному вычислительному потоку функцию *ИсправноеМоделирование()*. Параметрами, которые передаются в поток являются ссылка на описание моделируемого ЦУ и ссылка на входную последовательность-особь. Результатом работы потока моделирования является массив значений сигналов SV на всех линиях моделируемого ЦУ. Тогда в терминах потоков последовательный код А2.9 может быть записан следующим образом.

Алгоритм А4.1

```
ОценитьОсобь(A1,A2,Особь)
{
  ПотокМоделирования1->СоздатьПоток();
  ПотокМоделирования1->Данные(A1,Особь);
  ПотокМоделирования1->Выполнить();
  ПотокМоделирования1->ЖдатьЗавершения();
```

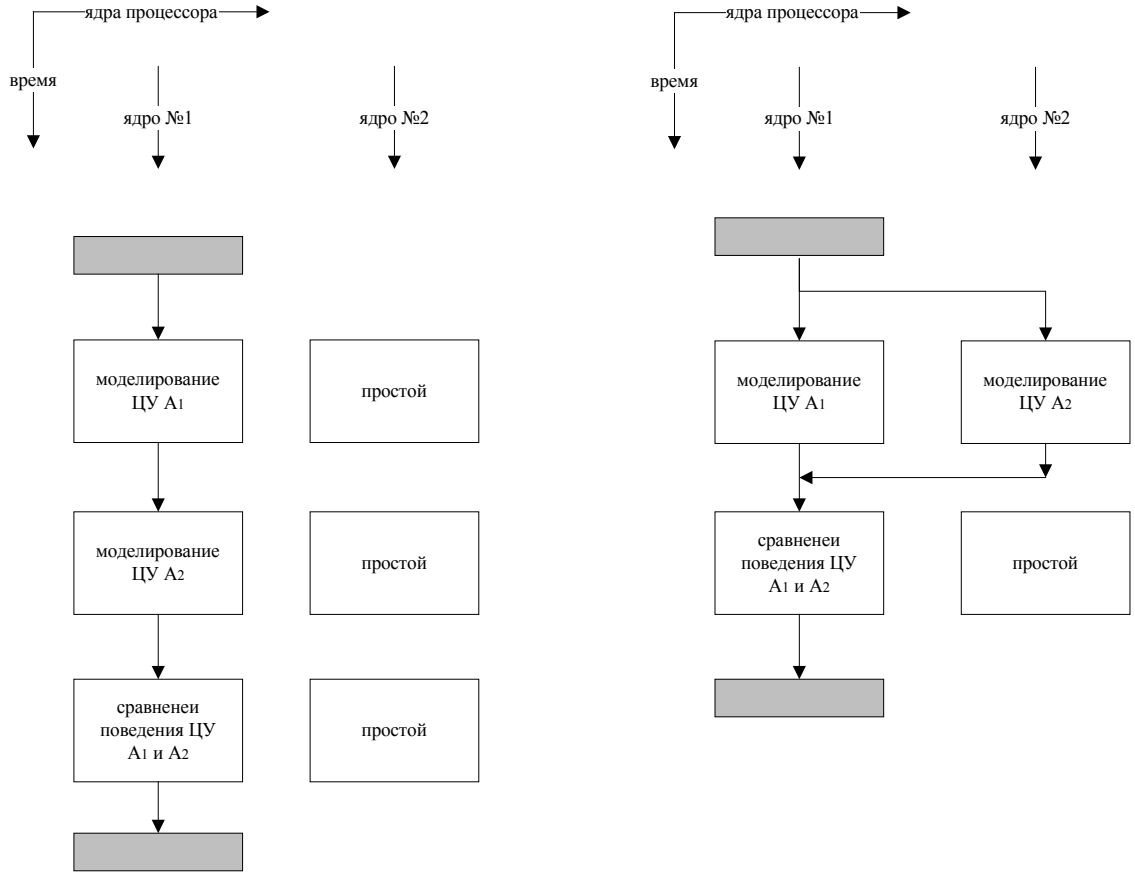
```
  ПотокМоделирования2->СоздатьПоток();
  ПотокМоделирования2->Данные(A2,Особь);
  ПотокМоделирования2->Выполнить();
  ПотокМоделирования2->ЖдатьЗавершения();
  n1=СравнениеПоведения(SV1,SV2);
  n2=СравнениеПоведения(SV1,SV2);
  n3=СравнениеПоведения(SV1,SV2);
  // непосредственно вычисление оценочной функции
  Оценка=ВычислитьОценку(n1,n2,n3);
  ПотокМоделирования1->УдалитьПоток();
  ПотокМоделирования2->УдалитьПоток();
  return Оценка;
} // конец – вычисление оценки особи
```

В такой реализации последовательно создаются два вычислительных потока моделирования поведения исправного ЦУ. При этом второй поток создаётся и запускается только после окончания работы первого. После завершения второго потока происходит сравнение результатов моделирования двух ЦУ A_1 и A_2, на основании которого вычисляется оценка в соответствии с построенной оценочной функцией. Диаграмму выполнения потоков моделирования устройств A_1 и A_2 в последовательном варианте функции оценки особи иллюстрирует рис.4.2.а. Именно такая последовательная организация не позволяет полностью задействовать оба ядра в двухядерном процессоре.

Для устранения указанного недостатка необходимо организовать параллельное выполнение двух потоков моделирования поведения исправных ЦУ A_1 и A_2. Для этого достаточно запустить на выполнение второй вычислительный поток сразу после запуска первого. Псевдокод такой реализации вычисления функции оценки приведён ниже.

Алгоритм А4.2

```
ОценитьОсобь(Особь, A1, A2)
{
  ПотокМоделирования1->СоздатьПоток();
  ПотокМоделирования1->Данные(A1,Особь);
```



а) последовательная б) параллельная

Рис.4.2. Диаграмма выполнения процедур моделирования ЦУ в последовательной и параллельной версиях функции оценки особи.

```
ПотокМоделирования1->Выполнить();
ПотокМоделирования2->СоздатьПоток();
ПотокМоделирования2->Данные(A2,Особь);
ПотокМоделирования2->Выполнить();
ПотокМоделирования1->ЖдатьЗавершения();
ПотокМоделирования2->ЖдатьЗавершения();
n1=СравнениеПоведения(SV1,SV2);
n2=СравнениеПоведения(SV1,SV2);
n3=СравнениеПоведения(SV1,SV2);
// непосредственно вычисление оценочной функции
Оценка=ВычислитьОценку(n1,n2,n3);
ПотокМоделирования1->УдалитьПоток();
```

```
  ПотокМоделирования2->УдалитьПоток();
  return Оценка;
} // конец – вычисление оценки особи
```

Данную модификацию для двухядерной ВС показывет рис.4.2.б. Из рисунка также видно, что нераспараллеленным остаётся фрагмент вычисления оценочной функции последовательности путём сравнения поведений двух ЦУ A_1 и A_2. Однако длительность данной процедуры в сравнении с процедурами моделирования является небольшой и, следовательно, не должна ухудшать качество параллельной версии.

Поскольку предполагается, что два верифицируемых ЦУ A_1 и A_2 имеют одинаковую последовательностную структуру, а отличаются в реализации комбинационных блоков, то сложность процедур их моделирования примерно одинакова и для разных ядер время моделирования будет также примерно одинаковым. Это должно обеспечить хорошую балансировку нагрузки между вычислительными ядрами. Поскольку в нашем примере реализуется два параллельных потока, то наибольшее ускорение должно быть получено для двухядерных систем.

Для проверки эффективности предложенной модификации алгоритма были проведены эксперименты на контрольных схемах из международного каталога ISCAS-89 [5]. Для экспериментов из каталога были взяты схемы средней и большой размерности – от 1500 логических вентилей. Программная реализация алгоритмов проводилась в среде CodeGear с использованием потоковых классов TThread.

Стратегия экспериментов (как и в двух дальнейших случаях) заключалась в фиксации некоторых параметров в ГА:

- число особей в популяции = 100;
- число поколений = 50.

Это, с одной стороны, позволяло исследовать зависимость эффективности параллельной версии алгоритма от организации параллельного моделирования устройств, а с другой стороны, учитывает, что в ГА помимо моделирования (оценки особей) есть «накладные расходы», которые выражаются в процедурах построения популяций. Для параллельной версии алгоритма данный фрагмент кода является последовательным и служит препятствием для ускорения работы.

Таблица 4.1

Время работы модификаций алгоритма на аппаратных платформах с различным числом вычислительных ядер.

	время работы, сек.							
	1 ядро[1]		2 ядра[2]		3 ядра[3]		4 ядра[4]	
	посл.[5]	пар.[6]	посл.	пар.	посл.	пар.	посл.	пар.
s3271	102	102	100	53	103	55	106	58
s3330	73	74	73	39	79	44	80	48
s3384	89	90	88	47	91	52	95	54
s4863	232	232	226	119	229	122	232	122
s5378	63	63	62	34	72	44	76	48
s6669	276	280	278	146	270	149	280	154
s9234	60	59	54	29	73	59	77	69
s13207	111	114	98	48	99	84	103	95
s15850	187	181	165	75	155	116	159	128
s35932	827	812	780	402	689	420	702	436
s38417	517	512	468	289	414	325	412	391
s38584	599	587	561	345	487	350	498	363

[1] - 2-ядерный процессор Intel Core 2 Duo E-4500 2,2Ghz, выключено 1 ядро;
[2] - 2-ядерный процессор Intel Core 2 Duo E-4500 2,2Ghz;
[3] - 4-ядерный процессор Intel Core Quad Q6600 2,4Ghz, выключено 1 ядро;
[4] - 4-ядерный процессор Intel Core Quad Q6600 2,4Ghz, время работы;
[5] – последовательная версия алгоритма;
[6] – параллельная версия алгоритма.

Эксперименты проводились на 2 различных аппаратных платформах с числом ядер от одного до четырёх, что достигалось отключением одного из ядер в каждой платформе. Поскольку аппаратные платформы не являлись эквивалентными, то для каждой из них измерялось время работы как последовательной, так и параллельной версий алгоритма. Результаты экспериментов приведены в табл.4.1.

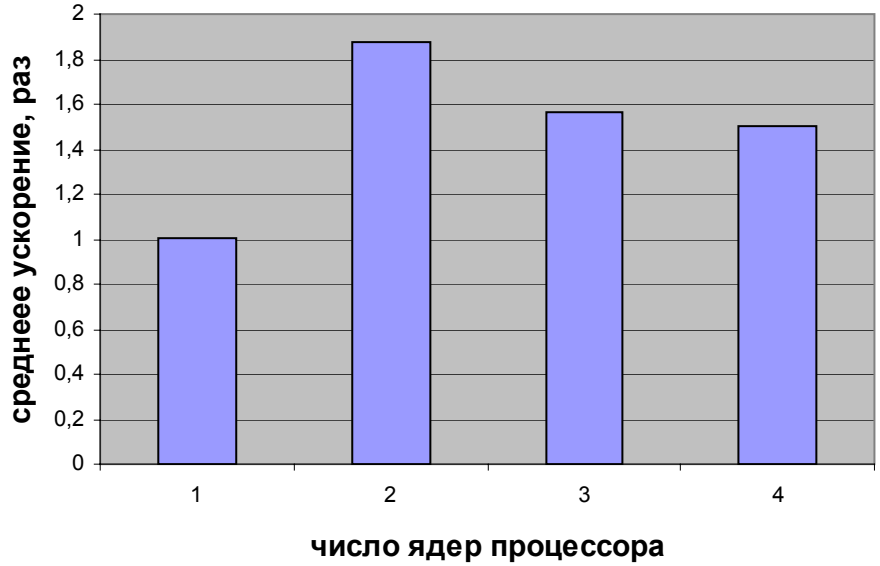


Рис.4.3. Среднее увеличение быстродействия для различных экспериментальных платформ.

На основании этих данных были вычислены средние для всех рассмотренных контрольных схем значения ускорений для каждой платформы. Диаграмма на рис.4.3 показывает сравнение этих значений. Видно, что наилучшее ускорение 1.88 раза получено для платформы с двухядерным процессором. Дальнейшее увеличение числа ядер не улучшает ситуации. Этот факт является очевидным, поскольку описанная модификация по сути является двухпоточной. Для трёх- и четырёхядерной платформы двухпоточная версия алгоритма даёт даже меньшее ускорение в сравнении с двухядерной. Видно, что распределение двух вычислительных потоков на три или четыре ядра ведёт к деградации производительности.

Эффективность параллельных алгоритмов принято измерять следующими параметрами: ускорение работы, эффективность загрузки ядер и доля последовательного кода [157].

Ускорение, рассчитываемое для параллельной реализации алгоритма, определяется формулой:

$$S_p(n) = \frac{T_1(n)}{T_p(n)}, \qquad (4.1)$$

где p – число процессоров в параллельной реализации алгоритма, n – параметр вычислительной сложности алгоритма, $T_i(n)$ - время выполнения параллельного алгоритма на системе с i процессорами.

Здесь и далее в работе под числом процессоров будем понимать:

- для вычислительной системы с общей памятью - суммарное число вычислительных ядер в системе без учёта их физического отношения к процессорам;
- для вычислительной системы с распределённой памятью – число однопроцессорных рабочих станций, которое входит в него.

Эффективность использования процессоров при параллельной реализации алгоритма рассчитывается по формуле:

$$E_p(n) = \frac{T_1(n)}{p \cdot T_p(n)} = \frac{S_p(n)}{p}. \qquad (4.2)$$

Поскольку здесь под числом процессоров понимается число ядер, то параметр $E_p(n)$ будет выражать эффективность использования ядер процессора.

Доля последовательного кода (доля последовательных вычислений) f служит для оценки свойств алгоритма к распараллеливанию и определяется из закона Амдаля:

$$f = \frac{p/S_p - 1}{p - 1}. \qquad (4.3)$$

Для системы с лучшим значением среднего ускорения были вычислены данные характеристики в соответствии с формулами 4.1-4.3, которые приведены в табл.4.2.

Заметим, что средние значения ускорения 1.88 раза, эффективности загрузки ядер 0.94, и доли последовательного кода 0.06 являются очень высокими показателями, хотя в предлагаемой модификации достигаются только на двухядерной ВС.

Также из табл.4.2 видно, что для двух контрольных схем s13207 и

Таблица 4.2

Значения параметров ускорения, эффективности использования ядер и доли последовательных вычислений для параллельной версии алгоритма.

	$S_2(n)$	$E_2(n)$	f
s3271	1,87	0,94	0,07
s3330	1,87	0,94	0,07
s3384	1,87	0,94	0.07
s4863	1,9	0,95	0,05
s5378	1,82	0,91	0,1
s6669	1,9	0,95	0,05
s9234	1,86	0,93	0,08
s13207	2	1	0
s15850	2,2	1,1	0
s35932	1,94	0,97	0,03
s38417	1,62	0,81	0,23
s38584	1,63	0,82	0,23
средн.	1,88	0,94	0,06

s15850 достигнуто линейное ускорение с коэффициентами 2.0 и 2.2. Это говорит о том, что последовательная версия алгоритма не оптимальна при запуске на двухядерной системе.

2. Второй подход [154-156] заключается *в параллельном вычислении оценок нескольких особей в одной популяции*.

Именно такое построение рассматривается рядом авторами как наиболее простой способ построения параллельных версий ГА [132].

В данном подходе процедура оценки особи оформляется в виде отдельного исполняемого потока. Привязав потоки к ядрам моделирования, мы получим ситуацию, когда на первом ядре вычисляется

оценка первой особи, на втором ядре – для второй и т.д. Число особей в генетическом алгоритме обычно велико (100 и более) в сравнении с числом ядер инструментальной ЭВМ, и поэтому необходимо дополнительное исследование, которое должно показать количество одновременно выполняемых потоков с наилучшей загрузкой вычислительных ядер.

Легко видеть, что предлагаемая схема также не изменяет структуру самого ГА, а именно его главный цикл построения новых популяций.

Для понимания сути задачи и разработки параллельной версии алгоритма по данной схеме рассмотрим подробно построение процедуры вычисления оценок особей. Вначале рассмотрим последовательную реализацию данной процедуры. Псевдокод её реализации представлен алгоритмом А2.4.

Процедура *ОценитьОсобь()*, которая вложена во внешний цикл, в непараллельной модели вычислений для особи i будет выполняться только после того, как завершена оценка предыдущей особи $i-1$. Если условно принять за один модельный такт время, которое необходимо для вычисления оценки одной особи в популяции, то схематично порядок вычисления оценок при такой организации процесса можно изобразить как на рис.4.4. Очевидно, что для последовательной модели с одним процессором/ядром данный порядок вычислений является естественным.

Построенные таким образом вычисления, как и в первом подходе, будут давать загрузку только одного ядра, а остальные ядра процессора будут простаивать. На самом же деле ситуация выглядит ещё хуже. Процессор не направит всю вычислительную нагрузку в одно ядро, а распределит её между всеми имеющимися ядрами.

Для построения параллельной версии сначала ассоциируем функцию *ОценитьОсобь()* с исполняемым потоком. В этом случае последовательный код в терминах потоков будет выглядеть следующим образом.

Алгоритм А4.3

```
ОценитьПопуляцию(A0, Популяция, ЧислоОсобей)
{
  for( i=0 ; i<ЧислоОсобей ; i++ )
```

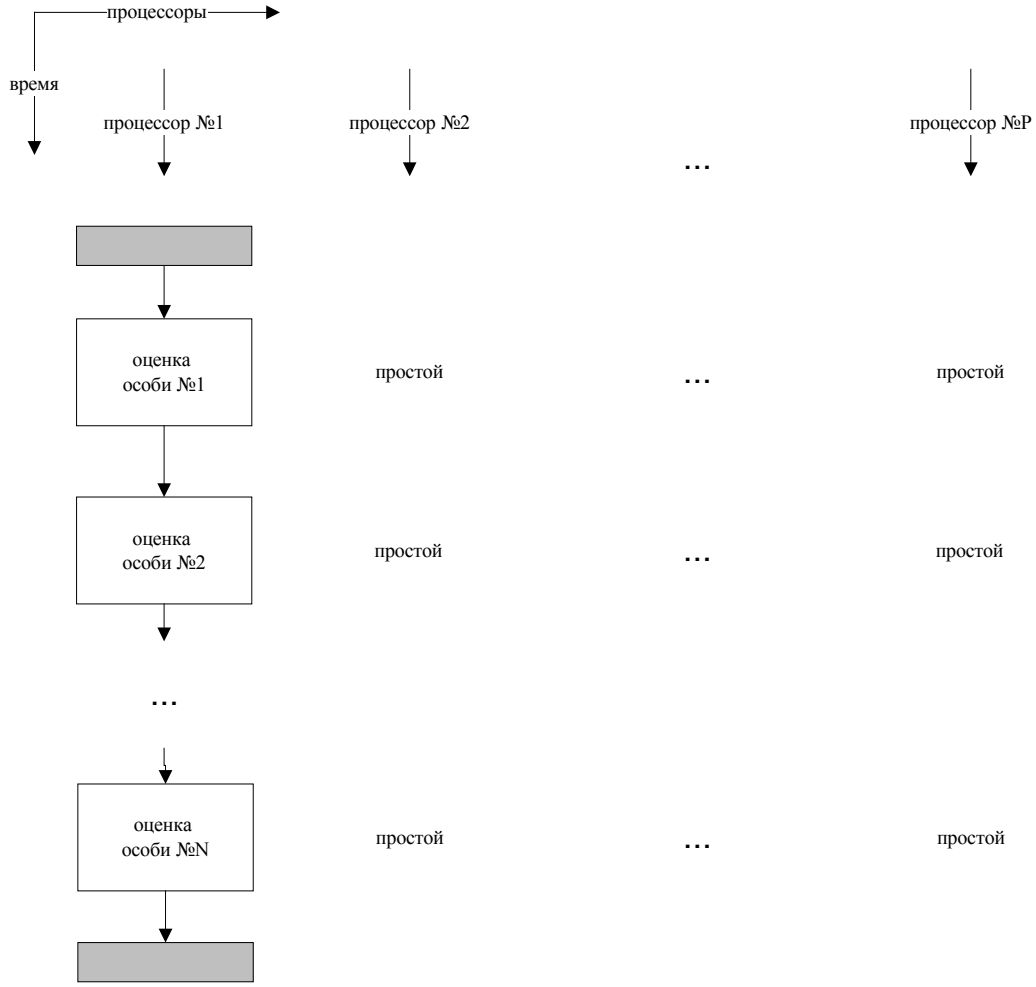


Рис.4.4. Диаграмма вычисления оценок особей в последовательной версии процедуры оценки популяции.

```
  {
    ПотокОценкиОсоби->СоздатьПоток();
    ПотокОценкиОсоби->Данные(A0,Популяция[i]);
    ПотокОценкиОсоби->Выполнить();
    ПотокОценкиОсоби->ЖдатьЗавершения();
    ПотокОценкиОсоби->УдалитьПоток();
  } // конец цикла по особям
}   // конец процедуры оценки популяции
```

Диаграмма исполнения данного алгоритма будет в точности соответствовать приведённой на рис.4.4.

Чтобы избежать простоя вычислительных ядер необходимо изменить структуру данной процедуры, позволив вычислительным потокам работать параллельно на ядрах процессора. Поскольку число работающих одновременно потоков оценки особей необходимо определять экспериментально, к аргументам функции *ОценитьПопуляцию()*добавиться параметр *ЧислоПотоков*.

Реализация изменённой процедуры оценки особей популяции для произвольного заранее заданного числа потоков приведена ниже.

Алгоритм А4.4

```
ОценитьПопуляцию(A0,Популяция,ЧислоОсобей,ЧислоПотоков)
{
  for( int i=0 ; i<ЧислоОсобей/ЧислоПотоков ; i++ )
  {
    k=i*ЧислоПотоков;
    // запуск заданного числа потоков
    for( int j=0 ; j<ЧислоПотоков ; j++ )
    {
      ПотокОценкиОсоби[j]->Данные(A0,Популяция[k+j]);
      ПотокОценкиОсоби[j]->Выполнить();
    } // конец for – подготовка и запуск потоков
    // ожидание окончания работы всех потоков
    for( int j=0 ; j<ЧислоПотоков ; j++ )
    {
      ПотокОценкиОсоби[j]->ЖдатьЗавершения();
    } // конец for – ждать окончания работы потоков
  }   // конец for – по пулу потоков
}     // конец процедуры оценки популяции
```

В данном коде переменная *ПотокОценкиОсоби* представляет собой массив указателей на вычислительные потоки, каждый из которых вычисляет оценку одной особи. Размерность такого пула потоков соответствует значению переменной *ЧислоПотоков*.

Пусть L - число одновременно запускаемых потоков (размер пула потоков, переменная *ЧислоПотоков* в псевдокоде), p - число вычислительных ядер в системе. Диаграмма работы процедуры оценки популяции (алгоритм А4.4) показана на рис.4.5. Здесь изображена первая и $j-я$ итерации запуска пула потоков.

В данном подходе после получения соответствующих данных

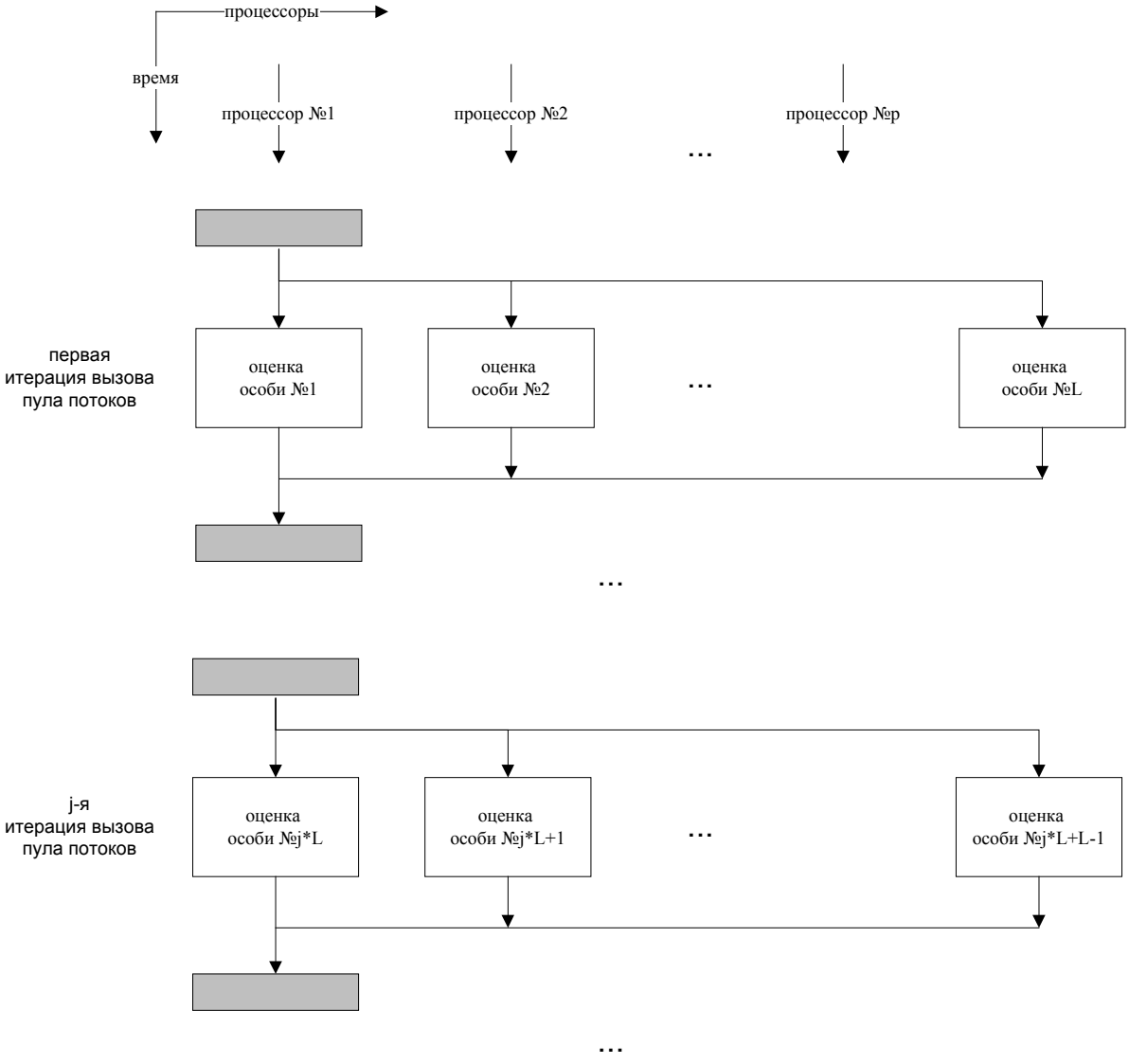


Рис.4.5. Диаграмма вычисления оценок особей в параллельной версии процедуры оценки популяции.

запускаются и работают параллельно потоки оценки разных особей в популяции. Момент времени окончания работы всех потоков в пуле является точкой синхронизации в ПГА. Поскольку число особей в популяции обычно велико и существенно больше числа процессоров p, а число запускаемых потоков L может не соответствовать числу процессоров в системе, то необходимы повторные запуски пула потоков. Число таких запусков определяется значением

$\left\lceil \frac{ЧислоОсобей}{L} \right\rceil$. При этом в последней итерации вызова пула потоков не все потоки будут запущены в работу и вычислять оценку особей, поскольку отношение выше может не быть целым числом.

Частое создание и удаление потоков существенно влияет на производительность ГА в целом. Размещение вызова соответствующих процедур в коде функции *ОценитьПопуляцию()* будет приводить к их итеративному вызову и создавать излишнюю вычислительную нагрузку, что будет вести к снижению скорости работы ГА в целом. Чтобы избежать негативного влияния данного фактора здесь и далее будет использоваться технология предварительного создания пула потоков. Для этого процедуры создания потоков выполняются предварительно на этапе инициализации работы ГА в соответствии с кодом ниже.

Алгоритм А4.5

```
ИнициализацияГА(…,ЧислоПотоков,…)
{
  …
  // создание массива потоков
  for( int i=0 ; i< ЧислоПотоков ; i++ )
  {
    ПотокОценкиОсоби[i]->СоздатьПоток();
  }
  …
}
```

Переменная *ЧислоПотоков* будет, таким образом, являться дополнительным параметром данной процедуры. Все потоки создаются в приостановленном виде и для начала работы требуют запуска процедуры *Выполнить()*.

Аналогично, код удаления пула потоков выполняется после окончания основного цикла ГА развития популяций в процедуре освобождения памяти.

Алгоритм А4.6

```
ОсвобождениеПамяти (…,ЧислоПотоков,…)
```

```
{
  …
  // удаление массива потоков
  for( int i=0 ; i< ЧислоПотоков ; i++ )
  {
    ПотокОценкиОсоби[i]->УдалитьПоток();
  }
  …
}
```

Таким образом, из кода процедуры оценки популяции (алгоритм А4.4) удалены функции создания/удаления потоков оценки особи, создающие излишнюю вычислительную нагрузку.

Реальное повышение производительности в предложенном подходе будет существенно зависеть от механизма переключения потоков операционной системы и механизма доступа ядер процессора к памяти, который должен обеспечивать вычислительные ядра необходимыми для работы данными.

Для проверки эффективности предложенной модификации проводились эксперименты с программной реализацией на системе с 4-ядерным процессором Intel Core2Quad Q6600.

В ходе экспериментов число одновременных потоков L изменялось от одного до четырёх (число ядер в инструментальной ЭВМ). Также для сравнения были проведены эксперименты с числом потоков, которое заведомо (и существенно) больше числа ядер в системе: $L = 100$. Параметры генетического алгоритма соответствовали экспериментам, проведённым с первым подходом параллелизации.

Численные результаты проведённых экспериментов приведены в табл.4.3. Столбец, в котором для выполнения указан один поток вычисления, соответствует последовательному коду и берётся в качестве эталона.

В данной серии экспериментов изменяется число вычислительных потоков. Его нельзя напрямую подставлять в качестве параметра p в формулы (4.1)-(4.3), поскольку число вычислительных ядер в системе фиксировано. Для того, чтобы различать характеристики по параметру числа потоков введём следующие обозначения с дополнительным индексом:

Таблица 4.3

Время работы параллельной версии алгоритма в зависимости от числа вычислительных потоков.

имя схемы	время работы, сек.				
	число параллельных функций оценки особей				
	1	2	3	4	100
s3271	106	63	47	39	33
s3330	78	47	38	33	30
s3384	94	56	44	37	32
s4863	225	131	97	79	68
s5378	74	49	43	38	41
s6669	300	173	127	103	91
s9234	72	69	73	75	74
s13207	101	108	111	119	127
s15850	158	135	132	129	151
s35932	711	446	343	302	359
s38417	416	345	319	304	358
s38584	491	370	319	389	367

- $S_p^i(n)$ - ускорение, которое достигается параллельной версией алгоритма с числом потоков i на системе с p процессорами относительно последовательной версии;
- $E_p^i(n)$ - эффективность использования процессоров при параллельной реализации алгоритма с i потоками на системе с p процессорами;
- f_n^i - доля последовательного кода в алгоритме при параллельной реализации алгоритма с i потоками на системе с p процессорами.

Таблица 4.4

Значения параметров ускорения, эффективности использования ядер и доли последовательных вычислений для параллельной версии алгоритма с различным числом исполняемых потоков.

имя схемы	число параллельных функций оценки особей											
	2			3			4			100		
	S_4^2	E_4^2	f_4^2	S_4^3	E_4^3	f_4^3	S_4^4	E_4^4	f_4^4	S_4^{100}	E_4^{100}	f_4^{100}
s3271	1.68	0.42	0.46	2.26	0.56	0.26	2.72	0.68	0.16	3.21	0.80	0.082
s3330	1.66	0.41	0.47	2.05	0.51	0.32	2.36	0.59	0.23	2.6	0.65	0.18
s3384	1.68	0.42	0.46	2.14	0.53	0.29	2.54	0.63	0.19	2.94	0.73	0.12
s4863	1.72	0.43	0.44	2.34	0.58	0.24	2.85	0.71	0.13	3.31	0.83	0.07
s5378	1.51	0.38	0.55	1.72	0.43	0.44	1.95	0.49	0.35	1.80	0.45	0.405
s6669	1.73	0.43	0.44	2.36	0.59	0.23	2.91	0.73	0.12	3.30	0.82	0.07
s9234	1.04	0.26	0.94	0.99	0.25	1.02	0.96	0.24	1.06	0.97	0.24	1.04
s13207	0.94	0.23	1.09	0.91	0.23	1.13	0.85	0.21	1.25	0.80	0.20	1.43
s15850	1.17	0.29	0.81	1.20	0.30	0.78	1.22	0.31	0.76	1.05	0.26	0.94
s35932	1.59	0.40	0.50	2.07	0.52	0.31	2.35	0.59	0.23	1.98	0.49	0.34
s38417	1.21	0.30	0.77	1.30	0.33	0.69	1.37	0.34	0.64	1.16	0.29	0.81
s38584	1.33	0.33	0.67	1.54	0.38	0.53	1.26	0.32	0.72	1.34	0.33	0.66
средн.	1.44	0.36	0.63	1.74	0.43	0.52	1.95	0.49	0.49	2.04	0.51	0.51

Видно, что введённый дополнительный индекс i числа вычислительных потоков непосредственно не влияет на вычисление данных характеристик по формулам (4.1)-(4.3), однако его использование необходимо для понимания ситуации и анализа результатов.

В табл.4.4 приведены параметры, характеризующие параллельную версию алгоритма для данной серии экспериментов. Из таблицы видно, что среднее ускорение для набора контрольных схем растёт с ростом числа параллельных функций оценки особей. При значении

$i = 100$ данное ускорение даже выше, чем при i равном числу ядер инструментальной системы: $i = 4$. Возможно, что для такого рода алгоритмов оптимальное значение числа параллельных запусков функций оценки особей i следует выбирать большим, чем число ядер системы. Этот вопрос подробно будет изучен в разделе 4.2.

В данной серии экспериментов также присутствует одна схема s13207, для которой предложенный подход не дал ускорения работы ни для одного из рассмотренных значений i. Для схемы s9234 ускорение работы алгоритма получено только для $i = 2$. Соответственно, в этих случаях для указанных схем получалось значение ускорения работы меньше единицы: $S < 1$, а доля последовательного кода выше единицы: $f > 1$.

3. Третий подход является комбинацией первых двух и заключается в том, что *при параллельном вычислении оценок особей в одной популяции внутри каждой процедуры оценки особи также есть параллельные процедуры моделирования ЦУ*.

В данном подходе в качестве параллельных потоков оформляется процедура вычисления оценочной функции, которая сама внутри является дополнительно распараллеленной. В такой реализации в псевдокоде алгоритма А4.4 каждая процедура *ОценитьОсобь()* также будет содержать параллельный код, который соответствует псевдокоду алгоритма А4.2.

Диаграмма работы вычислительных потоков для одной итерации цикла по пулу потоков (алгоритм А4.4) приведена на рис.4.6.

В данной реализации также необходим выбор оптимального числа одновременных потоков моделирования. Для экспериментов также использовался ГА верификации эквивалентности цифровых схем, в котором фиксировалось число особей и максимальное число итераций построения популяций.

Инструментальная платформа и характер проводимых экспериментов соответствует предыдущему случаю.

Время работы программной реализации алгоритма в зависимости от числа параллельных функций оценки особей приведено в табл.4.5. Заметим, что реальное число вычислительных потоков L в экспериментах было в два раза больше, чем число i параллельных функций оценки особи. Это объясняется тем, что каждая такая функция внутри

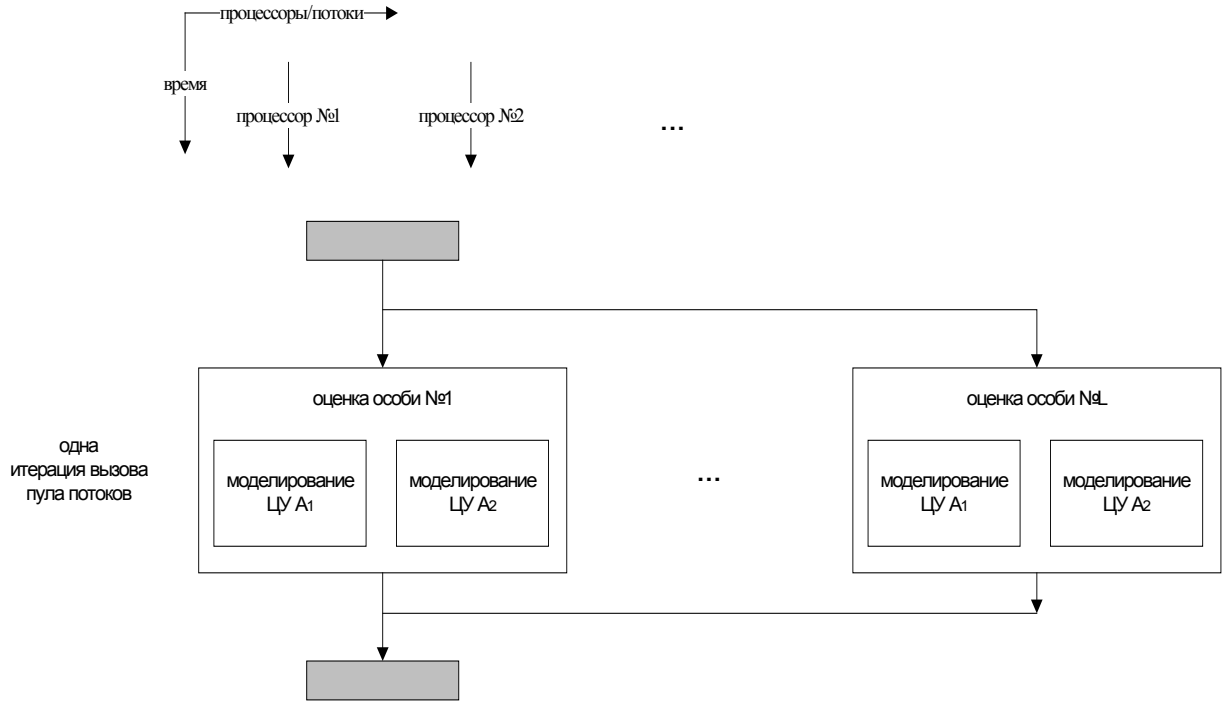


Рис.4.6. Диаграмма параллельного вычисления оценок особей, содержащих параллельные процедуры моделирования ЦУ.

содержит код параллельного моделирования двух верифицируемых ЦУ.

Характеристики параллелизации алгоритма при третьем подходе приведены в табл.4.6. Здесь следует отметить, что даже при минимальном числе параллельных оценочных функций $=1$ код реализации содержит два вычислительных потока. Поэтому брать его время исполнения в качестве последовательной версии нельзя. Для этой цели в расчётах выбрано время работы модификации алгоритма в соответствии со вторым подходом с числом параллельных функций оценки особи $=1$; при этом число параллельных вычислительных потоков также $=1$.

Таблица 4.5

Время работы параллельной версии алгоритма в зависимости от числа вычислительных потоков.

имя схемы	время работы, сек.				
	число параллельных функций оценки особей				
	1	2	3	4	100
	число параллельных потоков моделирования ЦУ				
	2	4	6	8	200
s3271	57	38	38	35	36
s3330	45	33	32	30	35
s3384	53	35	36	33	36
s4863	121	72	77	71	70
s5378	47	39	38	36	46
s6669	148	95	103	98	97
s9234	71	79	77	77	80
s13207	105	119	122	124	133
s15850	137	132	132	191	154
s35932	477	287	292	288	362
s38417	374	304	308	322	359
s38584	400	285	297	301	368

Наилучшее среднее ускорение времени работы $S_4 = 3.21$ раза (схема s4863) достигнуто при числе параллельных функции оценки особей $= 100$ (число потоков исполнения $= 200$). Это ещё раз подчёркивает необходимость дополнительного изучения случаев, когда число вычислительных потоков больше числа ядер процессора. Для большинства контрольных схем (s35932, s38417, s38581 и др.) максимальное ускорение достигнуто при числе параллельных функций

Таблица 4.6

Значения параметров ускорения, эффективности использования ядер и доли последовательных вычислений для параллельной версии алгоритма с различным числом исполняемых потоков.

имя схемы	число параллельных функций оценки особей (потоков)														
	1(2)			2 (4)			3 (6)			4 (8)			100 (200)		
s3271	1,86	0,46	0,38	2,79	0,70	0,14	2,79	0,70	0,14	3,03	0,76	0,11	2,94	0,74	0,12
s3330	1,73	0,43	0,44	2,36	0,59	0,23	2,44	0,61	0,21	2,60	0,65	0,18	2,23	0,56	0,26
s3384	1,77	0,44	0,42	2,69	0,67	0,16	2,61	0,65	0,18	2,85	0,71	0,13	2,61	0,65	0,18
s4863	1,86	0,46	0,38	3,13	0,78	0,09	2,92	0,73	0,12	3,17	0,79	0,09	3,21	0,80	0,08
s5378	1,57	0,39	0,51	1,90	0,47	0,37	1,95	0,49	0,35	2,06	0,51	0,32	1,61	0,40	0,50
s6669	2,03	0,51	0,32	3,16	0,79	0,09	2,91	0,73	0,12	3,06	0,77	0,10	3,09	0,77	0,10
s9234	1,01	0,25	0,98	0,91	0,23	1,13	0,94	0,23	1,09	0,94	0,23	1,09	0,90	0,23	1,15
s13207	0,96	0,24	1,05	0,85	0,21	1,24	0,83	0,21	1,28	0,81	0,20	1,30	0,76	0,19	1,42
s15850	1,15	0,29	0,82	1,20	0,30	0,78	1,20	0,30	0,78	0,83	0,21	1,28	1,03	0,26	0,97
s35932	1,49	0,37	0,56	2,48	0,62	0,20	2,43	0,61	0,21	2,47	0,62	0,21	1,96	0,49	0,35
s38417	1,11	0,28	0,87	1,37	0,34	0,64	1,35	0,34	0,65	1,29	0,32	0,70	1,16	0,29	0,82
s38584	1,23	0,31	0,75	1,72	0,43	0,44	1,65	0,41	0,47	1,63	0,41	0,48	1,33	0,33	0,67
средн.	1,48	0,37	0,62	2,05	0,51	0,46	2,00	0,50	0,47	2,06	0,52	0,50	1,90	0,48	0,55

оценки особи $=4$ (число вычислительных потоков $=8$). При этих же значениях достигнуто наибольшее среднее ускорение по набору контрольных схем: $S=2.06$.

Как и в подходе №2, для двух контрольных схем (s9234 и s13207) параллельные версии алгоритма замедляют работу. При этом худший результат (наибольшее замедление) $S=0.76$ показано в подходе №3 при числе параллельных функций $=100$.

Диаграмма, показывающая сравнение средних значений ускорения для второго и третьего подходов, приведена на рис.4.7.

При сравнении двух последних подходов видно, что при числе параллельных функций оценки i от 1 до 4 лучшие результаты показывает подход №3. При этом число параллельных вычислительных потоков в два раза больше в сравнении с подходом №2, что позволяет

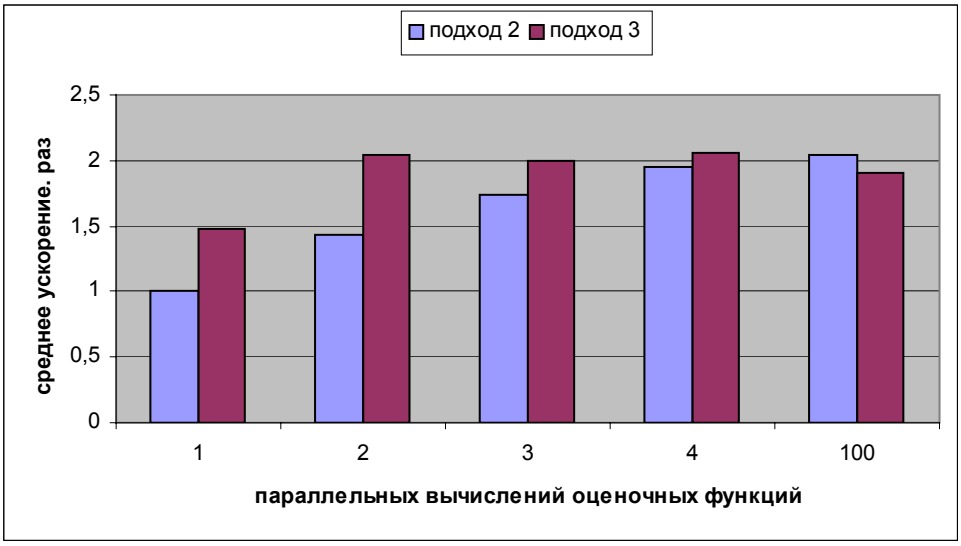


Рис.4.7. Среднее ускорение работы модификаций алгоритма в зависимости от числа параллельных функций оценки особей.

более эффективно загружать вычислительные ядра работой. Это также является ещё одним подтверждением тому, что в данном подходе, возможно, следует выбирать число параллельных потоков большим, чем число ядер процессора.

При числе параллельных функций оценки большем числа ядер выигрывает подход №2. Именно он будет выбран далее при изучении работы алгоритма на сильнопараллельной ВС.

Наибольшее среднее ускорение $S = 2.06$ достигнуто в подходе 3 для случая, когда параллельно вычисляются четыре процедуры *ОценитьОсобь()*. Однако сравнение таблиц 4.4 и 4.6 показывает, что наибольшие абсолютные значения ускорения ($S = 3.31$ для схемы s4863 и $S = 3.30$ для схемы s6669) достигнуты именно во втором подходе.

Общим свойством предложенных подходов является то, что разброс значений достигнутых ускорений работы является существенным: от замедления работы $S = 0.76$ до существенного роста скорости $S = 3.31$.

Однако достигнутое среднее ускорение $S = 2.04 - 2.06$ показывает, что загрузку ядер удалось увеличить более чем в два раза в сравнении с последовательной версией алгоритма.

4.2. Схема «хозяин-рабочий» параллельных ГА для сильно-параллельных ВС с общей памятью

В настоящее время стандартом «де-факто» для рабочих станций становится применение многоядерных процессоров [83], отражая успехи производителей в разработке таких процессоров. В коммерческих образцах процессоров уже сегодня число ядер достигает 4-6. При этом в компании Intel ведутся разработки коммерческих чипов с числом вычислительных ядер до 80-и. Это подхлестнуло интерес к разработке параллельных алгоритмов для таких рабочих станций. Для разработчиков инструментальных средств такие рабочие станции представляются как многопроцессорные ВС с общей памятью.

В предыдущем разделе 4.1 предложены параллельные версии ГА для работы на двух- и четырёхядерных рабочих станциях, построенных по схеме «хозяин-рабочий». В данном разделе изучаются свойства масштабируемости рассмотренных параллельных версий ГА-методов в отображении на рабочие станции с большим числом вычислительных ядер.

Под масштабируемостью программного обеспечения понимают способность к пропорциональному увеличению производительности при соответствующем увеличении аппаратных средств. В нашем случае под увеличением аппаратных средств будем понимать увеличение числа вычислительных ядер в инструментальной системе. Основной вопрос проводимого исследования: являются ли предлагаемые параллельные версии ГА-методов масштабируемыми, т.е. будет ли дальнейшее увеличение числа ядер в инструментальной системе подтверждено адекватным ростом скорости работы ПГА?

Предложенные в предыдущем разделе три подхода имеют различную практическую применимость.

Первый подход отлично работает на двухядерных вычислительных системах, где показывает наилучшие показатели параллелизации. Однако самостоятельное использование таких систем ограничено. Чаще подход может быть применим в параллельных системах, где один вычислительный узел построен на двухядерном процессоре. Это же замечание справедливо и для третьего (комбинированного) подхода.

С другой стороны, наиболее общим является подход, в котором параллелизации подвергаются процедуры оценки отдельных особей.

Именно он имеет наибольшие перспективы при дальнейшем росте числа процессоров рабочих станций, поскольку метод не накладывает ограничений на структуру процессоров в таких системах.

Вторым его преимуществом является то, что он позволяет производить адаптацию последовательных ГА для работы на параллельных ВС с минимальными модификациями. В частности, изменению подвергается единственная процедура оценки особей в популяции.

Именно этот подход был выбран для адаптации на сильнопараллельные ВС с общей памятью [158-160]. Таким образом, базовой реализацией параллельной функции оценки особей в популяции является представленная в алгоритме А4.4. Ограничения в работе ГА соответствуют таковым в разделе 4.1.

Эксперименты по апробации подхода производились на 12-ядерной рабочей станции со следующими техническими характеристиками:

- многоядерная ВС содержала два процессора Intel(R) Xeon CPU X5650 с частотой 2.67Ггц (каждый по 6 вычислительных ядер); функция GetSystemInfo() показывала наличие 12 вычислительных ядер в системе;
- технология HyperThreading – выключена;
- объем оперативной памяти 16 Гб;
- операционная система MS Windows Server 2008 R2.

В таких условиях были проведены машинные эксперименты, которые должны дать ответ на следующие вопросы:

- является ли предложенная структура параллельного алгоритма масштабируемой?
- каков предел масштабируемости данной версии алгоритма?
- сколько вычислительных потоков необходимо запускать для наилучшей загрузки системы, и как это число связано с числом вычислительных ядер?

При проведении экспериментов также использовались контрольные схемы ISCAS-89, из которых сразу были исключены схемы малой размерности ввиду того, что для них время моделирования составляет порядка нескольких секунд и построение параллельных версий алгоритмов не является актуальным.

Первые же эксперименты показали, что нет никакой необходимости привязывать вычислительные потоки к ядрам. Это в несколько раз уменьшает производительность параллельной версии алгоритма.

Данный факт согласуется с выводами [161]: в параллельной вычислительной среде наибольшее ускорение получается в случае такой привязки потоков, которая допускает их миграцию между ядрами внутри одного сокета либо вычислительного узла. Поскольку наш вычислительный сервер фактически представляет собой один вычислительный узел с 12 ядрами, то наибольшее ускорение должно быть получено без привязки потоков к ядрам.

При проведении машинных экспериментов для выбранного множества схем выполнялись многократные запуски параллельной версии ГА с различным числом одновременных вычислительных потоков (переменная *ЧислоПотоков* в коде алгоритма А.4).

Все проведённые эксперименты были разделены на две серии.

В первой серии число потоков изменялось от 1 до 12, где верхняя граница определялась числом физических вычислительных ядер.

Во второй серии экспериментов число потоков изменялось от 1 до 128. Данная серия экспериментов проводилась с целью выяснить максимальное возможное ускорение ПГА. Данные о возможности достигнуть наибольшее ускорение параллельной версии алгоритма при числе вычислительных потоков большем, чем число вычислительных ядер, получены в разделе 4.1.

Следует сделать замечание о базе сравнения параллельных версий алгоритма. В качестве базовой последовательной версии алгоритма был выбран ПГА, в котором на инструментальной ЭВМ с 12 вычислительными ядрами запускался один вычислительный поток оценки особи. Скорость работы такой реализации, вообще говоря, отличается от истинно последовательной версии алгоритма, которая должна соответствовать системе с одним вычислительным ядром, и уменьшается на 1-4%, что связано с накладными расходами на организацию параллельных вычислений. Однако такую погрешность вполне можно считать приемлемой, поскольку не имеется никакой возможности в инструментальной системе отключить от работы 11 ядер для получения абсолютно точных цифровых данных.

Числовые результаты времени работы для контрольных схем в первой серии экспериментов приведены в табл.4.7.

Вычисленные на их основании характеристики ускорения работы алгоритма S_p^i приведены в табл.4.8.

Приведённые в табл.4.7-4.8 численные данные свидетельствуют,

Таблица 4.7

Время работы параллельной версии алгоритма в зависимости от числа параллельных вычислительных потоков оценки особей в популяции.

имя схемы	время работы в зависимости от числа параллельных потоков i, сек.											
	1	2	3	4	5	6	7	8	9	10	11	12
s3271	162	91	65	45	39	29	29	22	26	19	24	18
s3330	124	69	50	38	33	28	28	26	25	24	24	22
s3384	143	83	59	46	39	34	32	30	28	24	26	23
s4863	342	193	132	78	72	51	52	45	49	38	45	36
s5378	107	65	49	38	33	31	30	29	29	28	28	27
s6669	406	229	160	118	99	78	75	63	63	53	57	52
s9234	84	57	47	42	38	37	37	37	37	37	38	37
s13207	124	87	71	64	60	59	59	58	59	58	58	57
s15850	197	126	95	78	67	63	63	62	63	61	63	62
s35932	914	511	291	200	174	151	144	137	133	115	121	109
s38584	633	372	228	148	124	118	112	107	111	98	104	97

что предлагаемый подход показывает очень хорошую масштабируемость для большей части контрольных схем:

- максимальное достигнутое ускорение $S_{12}^{i} = 9.5$;
- среднее по набору схем достигнутое ускорение $S_{12}^{i} = 5.88$.

Следует отметить, что в отличие от экспериментов на рабочей станции с четырьмя вычислительными ядрами, в данных экспериментах ускорение работы получено для всех схем. Для схем, у которых ускорение на слабопараллельной ВС было меньше единицы (s9234 и s13207), в данной серии получено ускорение больше единицы: 2.27 и 2.18 раза соответственно. Также при числе потоков 2-4 (что соответствует экспериментам раздела 4.1) было получено небольшое ускорение работы.

Таблица 4.8

Ускорение работы параллельной версии алгоритма в зависимости от числа параллельных вычислительных потоков оценки особей в популяции.

имя схемы	ускорение s_{12}^{i} работы в зависимости от числа i параллельных потоков, сек.											
	1	2	3	4	5	6	7	8	9	10	11	12
s3271	1,00	1,78	2,49	3,60	4,15	5,59	5,59	7,36	6,23	8,53	6,75	9,00
s3330	1,00	1,80	2,48	3,26	3,76	4,43	4,43	4,77	4,96	5,17	5,17	5,64
s3384	1,00	1,72	2,42	3,11	3,67	4,21	4,47	4,77	5,11	5,96	5,50	6,22
s4863	1,00	1,77	2,59	4,38	4,75	6,71	6,58	7,60	6,98	9,00	7,60	9,50
s5378	1,00	1,65	2,18	2,82	3,24	3,45	3,57	3,69	3,69	3,82	3,82	3,96
s6669	1,00	1,77	2,54	3,44	4,10	5,21	5,41	6,44	6,44	7,66	7,12	7,81
s9234	1,00	1,47	1,79	2,00	2,21	2,27	2,27	2,27	2,27	2,27	2,21	2,27
s13207	1,00	1,43	1,75	1,94	2,07	2,10	2,10	2,14	2,10	2,14	2,14	2,18
s15850	1,00	1,56	2,07	2,53	2,94	3,13	3,13	3,18	3,13	3,23	3,13	3,18
s35932	1,00	1,79	3,14	4,57	5,25	6,05	6,35	6,67	6,87	7,95	7,55	8,39
s38584	1,00	1,70	2,78	4,28	5,10	5,36	5,65	5,92	5,70	6,46	6,09	6,53
средн.	1,00	1,68	2,38	3,27	3,75	4,41	4,50	4,98	4,86	5,65	5,19	5,88

В общем, приведенные числовые характеристики являются очень высокими. Авторам в настоящее время не известны аналогичные результаты для параллельных генетических алгоритмов, использующих для вычисления оценки особей исправное моделирование поведения ЦУ. Поэтому произвести прямого корректного сравнения с аналогами не возможно. Подобные результаты приведены в [162], однако относятся к моделированию с неисправностями. Полученные здесь параметры ускорения работы параллельной версии алгоритма превосходят, опубликованные в [162].

Графики ускорения работы ПГА в лучшем, худшем и среднем случаях для первой серии экспериментов приведены на рис.4.8.

Интересной особенностью является нелинейный рост ускорения в зависимости от числа потоков: для чётного числа потоков уско-

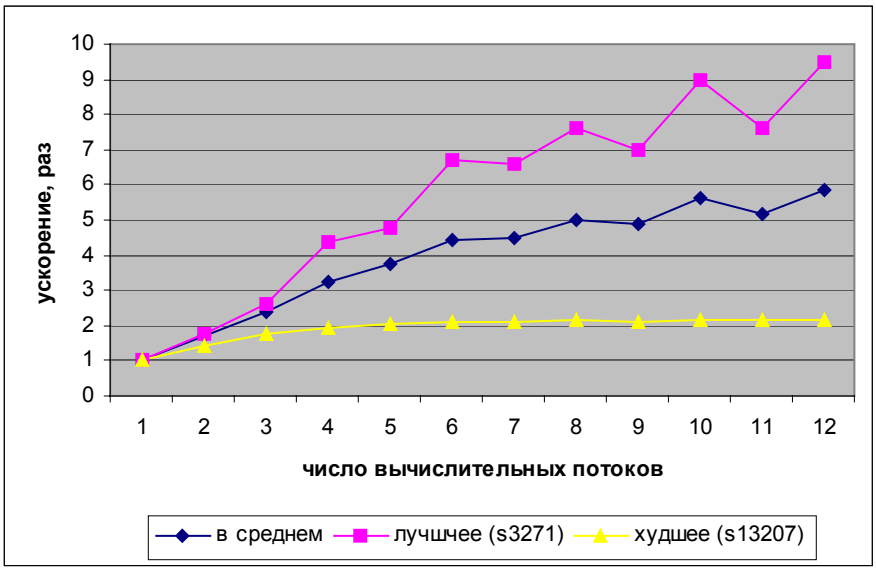


Рис.4.8. Графики ускорения работы ПГА в зависимости от числа вычислительных потоков в среднем, лучшем и худшем случаях. Серия 1 экспериментов.

рение выше, чем для нечётного. Очевидно, причиной данного явления является структура многоядерных процессоров: шестиядерный процессор состоит из трёх блоков вычислительных ядер, каждый из которых включает два вычислительных ядра с общей кэш-памятью. В том случае, когда число вычислительных потоков не является кратным двум, данные в кэш-память одного из таких блоков поступают только для одного вычислительного ядра, что и ведёт к снижению производительности. Таким образом, число вычислительных потоков i следует выбирать кратным числу вычислительных ядер p в узле ВС.

В табл.4.9, колонка «серия 1» приведены результаты по максимально достигнутому ускорению S_{12}^{i} и соответствующему числу вычислительных потоков i для каждой из контрольных схем. Анализ данных результатов показывает, что почти для всех схем максимальное ускорение работы ПГА достигнуто в правом конце диапазона изменений числа потоков при $i = 12$. Это факт, а также результаты экспериментов в разделе 4.1 заставляют сделать предположение, что дальнейшее наращивание числа потоков i позволит и далее расти па-

Таблица 4.9

Максимальное достигнутое ускорение ПГА для контрольных схем.

схема	серия 1, 12 потоков максимум		серия 2, 128 потоков максимум	
	макс. достигнутое ускорение, S_{12}^{i}	минимальное число потоков i, при котором достигнуто ускорение	макс. достигнутое ускорение, S_{12}^{i}	минимальное число потоков i, при котором достигнуто ускорение
s3271	9.00	12	13.50	105
s3330	5.64	12	9.54	115
s3384	6.22	12	11.00	100
s4863	9.50	12	13.15	119
s5378	3.96	12	4.12	18
s6669	7.81	12	12.30	128
s9234	2.27	7	2.33	17
s13207	2.18	12	2.18	12
s15850	3.23	10	3.23	12
s35932	8.39	12	10.51	125
s38584	6.53	12	7.11	25
средн.	5.88	11.36	8.09	80

раметру ускорения работы S.

С этой целью была проведена вторая серия экспериментов. Здесь параметр числа потоков i изменялся от 1 до 128.

Поскольку объём числовых результатов, полученных в результате проведения экспериментов очень велик, мы будем, в основном, приводить значения для времени работы и ускорения работы ПГА. На основе данной информации и известных формул можно вычислить остальные характеристики параллельной версии алгоритма: эффек-

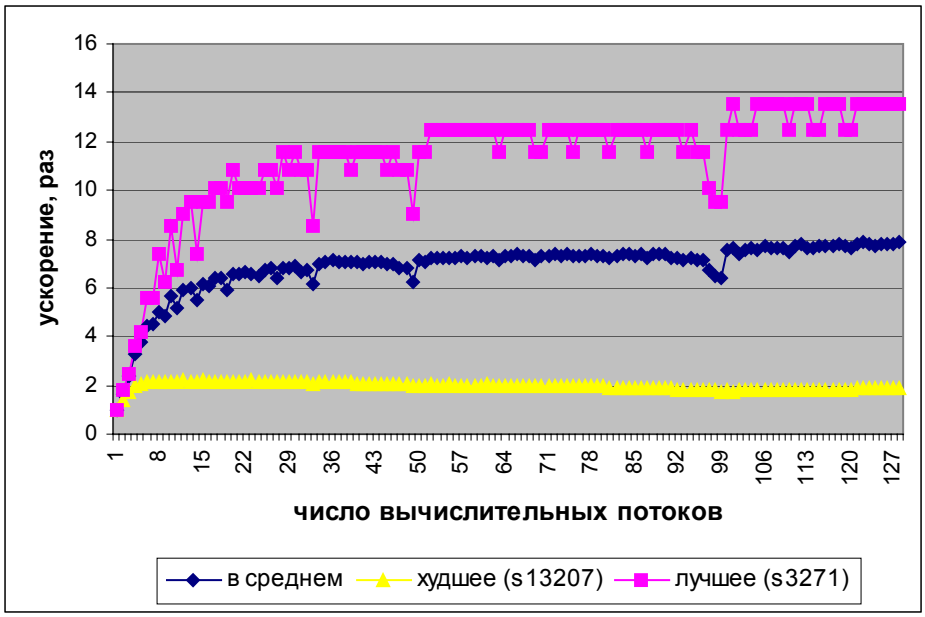


Рис.4.9. Графики ускорения работы ПГА в зависимости от числа вычислительных потоков в среднем, лучшем и худшем случаях. Серия 2 экспериментов.

тивность использования ядер и долю последовательного кода [157].

На рис.4.9 приведёны графики ускорения работы ПГА в лучшем, худшем и среднем случаях в данной серии экспериментов.

В табл.4.9 (колонка «серия 2») показаны максимальные достигнутые ускорения S_{12}^{i} для каждой из контрольных схем и набора в целом и параметр числа потоков i, при котором они достигаются.

В целом табл.4.9 позволяет сравнить аналогичные данные в первой и второй сериях экспериментов.

Сравнение приведённых данных с данными экспериментов серии 1 и графиком на рис.4.8 показывает, что максимальное достигнутое ускорение существенно выросло: с $S_{12}^{i}=9.5$ до $S_{12}^{i}=13.5$ (схема s3271). Среднее ускорение работы ПГА по набору схем выросло с $S_{12}^{i}=5.88$ до $S_{12}^{i}=8.09$.

Для трёх схем из одиннадцати (s3271, s4863, s6669) максимальное ускорение S_{12}^{i} было выше, чем число вычислительных ядер $p=12$ в системе: 13.50, 13.15 и 12.30 раз соответственно.

Для одной из схем (s3384) дальнейшее увеличение числа потоков позволило увеличить полученное в серии 1 значение ускорения ещё в 1.77 раза: с 6.22 до 11 раз.

Только для одной из схем (s6669) во второй серии экспериментов максимальное ускорение достигнуто при максимальном числе потоков 128. Это говорит о том, что для набора схем в целом нет необходимости далее увеличивать число параллельных вычислительных потоков i.

Для схем с низким значением параметра ускорения в первой серии экспериментов дальнейший рост числа потоков практически не улучшает ситуацию. А для двух схем (s13207, s15850) лучшие значения ускорения S_{12}^{i} относятся к первой серии экспериментов: $i \le 12$.

Таким образом, приведённые числовые результаты показывают, что для максимальной утилизации вычислительных ядер в системе, число одновременно запущенных потоков i должно быть больше, чем число ядер. Среднее значение по набору контрольных схем составило $i = 80$, что существенно больше числа ядер вычислительной системы $p = 12$. Однако сильное варьирование данного параметра говорит, что точное его определение всё ещё не представляется возможным.

Такой результат опровергает [163], где утверждается, что число вычислительных потоков в многоядерной системе должно равняться числу вычислительных ядер.

4.3. Схема «островов» параллельных ГА

Первоначально попытки построения распределённых ГА строились на модели «хозяин-рабочий». Это связано с простотой подхода, в частности, с тем, что структура алгоритма ГА при этом не изменяется. В такой модели существует только одна копия ГА, которая выполняется на главном процессоре, называемом «хозяином». Остальные процессоры в узле осуществляют второстепенные функции: на них лежит задача вычисления оценочных функций особей в популяции, заключающаяся в моделировании поведения ЦУ. Таким образом, непосредственный поиск осуществляется только на головном компь-

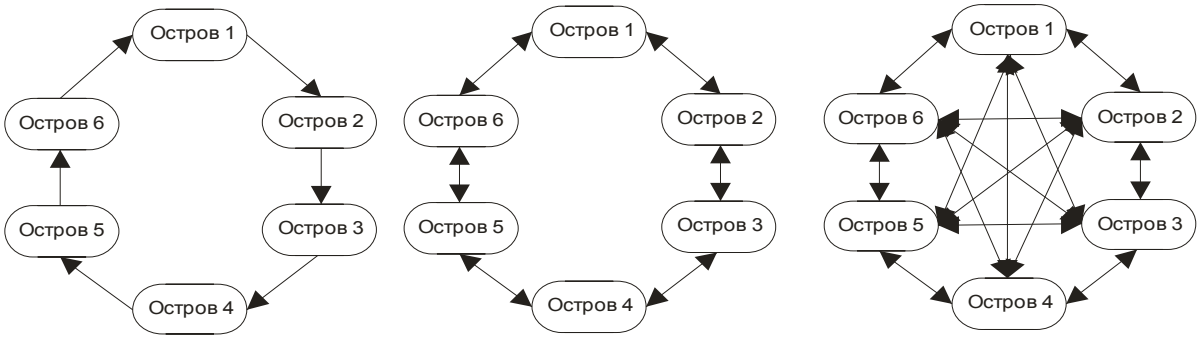
ютере, а параллелизация служит только для ускорения работы алгоритма.

В данном разделе подробно описываются методы работы компонент распределённой системы вычислений, реализующих модель островов распределённых генетических алгоритмов (РГА) построения тестов [164]. На основе обобщения известных методов построения ПГА предлагается подход, который позволяет легко реализовывать различные топологии обмена данными между компонентами и различные стратегии адаптации их параметров без изменения своей структуры.

Первые попытки построения параллельных версий ГА генерации тестов по схеме островов относятся к середине 90-х годов прошлого столетия [63, 70]. Однако рассмотренные подходы либо являются поверхностно описанными, либо имеют узкую направленность (например, рассчитаны на определённый тип оборудования в кластере). В основном, описание таких алгоритмов сводится к тому, что происходит параллельная эволюция нескольких популяций со слегка изменёнными параметрами.

«Модель островов» ПГА разрабатывается в первую очередь не для повышения быстродействия работы ГА, а для улучшения качества поиска [127]. Модель островов отличается от модели «хозяин-рабочий» тем, что включает два главных структурных компонента: острова и сервер. Острова представляют собой подпопуляции решений. На каждом из таких островов выполняется своя копия ГА-метода построения ИдП: цикл построение новых популяций из текущей (глава 2). При этом генетические параметры работы этих алгоритмов, обычно, несколько отличаются друг от друга. Это позволяет направить поиск на каждом таком «острове» в отличном от других направлении. Организующим элементом выступает головной процессор-сервер (в некоторых реализациях он не выделяется явно), который выполняет управляющие функции: построение начальной популяции, инициализация ГА островов, определение параметров ГА островов, завершение их работы.

Важным элементом такой схемы является операция обмена между островами: через заданное количество итераций (достаточно редко) острова производят обмен лучшими особями. Данный обмен называется миграцией. Она позволяет корректировать направление поиска для менее удачных островов.



а) обмен по кольцу б) двунаправленный обмен по кругу в) обмен каждый с каждым

Рис.4.10. Различные топологи взаимодействия компонент в ПГА схемы островов.

Выделим основные характеристики миграции:

- топология, которая определяет: какие подпопуляции будут считаться соседними и, соответственно, между какими островами будет возможен обмен особями (рис.4.10);
- время изоляции, которое определяет, сколько эпох ГА не будет производиться миграция;
- степень миграции, которая показывает количество особей, участвующих в миграции;
- стратегия отбора особей для миграции;
- стратегия удаления особей из подпопуляции при проведении миграции;
- стратегия репликации мигрирующих особей.

Классифицировать распределенные генетические алгоритмы можно по методу миграции, по однородности подпопуляций и по параметрам схемы обмена.

По методу миграции выделяются:

- изолированные РГА, где миграция между подпопуляциями не происходит;
- синхронные РГА, где миграция происходит в одно и то же время;
- асинхронные ГА, где миграции происходят в разное время, как правило, при готовности 2 соседей.

По однородности популяций выделяют:

- однородные РГА, в которых параметры ГА (кодирование, кроссинговер, мутация, редукция и их вероятности) одинаковы для всех островов;
- неоднородные ГА, в которых каждый остров может иметь собственные значения параметров.

По виду обмена между популяциями существуют статическая и динамическая схема соединений. В статической схеме такие характеристики обмена особями, как, например, степень миграции и время изоляции, неизменны на протяжении всей работы алгоритма. В динамической же схеме характеристики могут изменяться в зависимости от сходимости в той или иной подпопуляции.

В разрабатываемом подходе структурный элемент «сервер» является основным компонентом, через который реализуется инициализация и политика взаимодействия остальных компонент. Поскольку топология взаимодействия островов реализуется на сервере, то каждый из островов не знает о существовании других. Отдельно взятый клиент точечно взаимодействует только с сервером. Это существенно упрощает реализацию метода, поскольку избавляет от механизма широковещательной связи между клиентами и необходимости разработки соответствующего протокола.

В качестве базового выберем двухуровневый ГА-метод построения проверяющих тестов ЦУ (раздел 3.2) с теми же условиями применимости. При дальнейшем описании также будем пользоваться введёнными ранее обозначениями:

- A_0 - заданное ЦУ;
- F - полный список неисправностей;
- F' - текущий список непроверенных неисправностей;
- $f_{цел}$ - текущая целевая неисправность;
- $S_{тест}$ - тест для целевой неисправности;
- S - финальный тест.

Верхний уровень данного метода (алгоритм А3.2) сопоставим серверу разрабатываемого подхода. Клиенты реализуют основной цикл ГА поиска теста для выбранной неисправности – алгоритм А3.4. Тогда схема взаимодействия компонент может быть представлена следующим образом – рис.4.11.

При таком разделении сервер выполняет следующие основные функции:

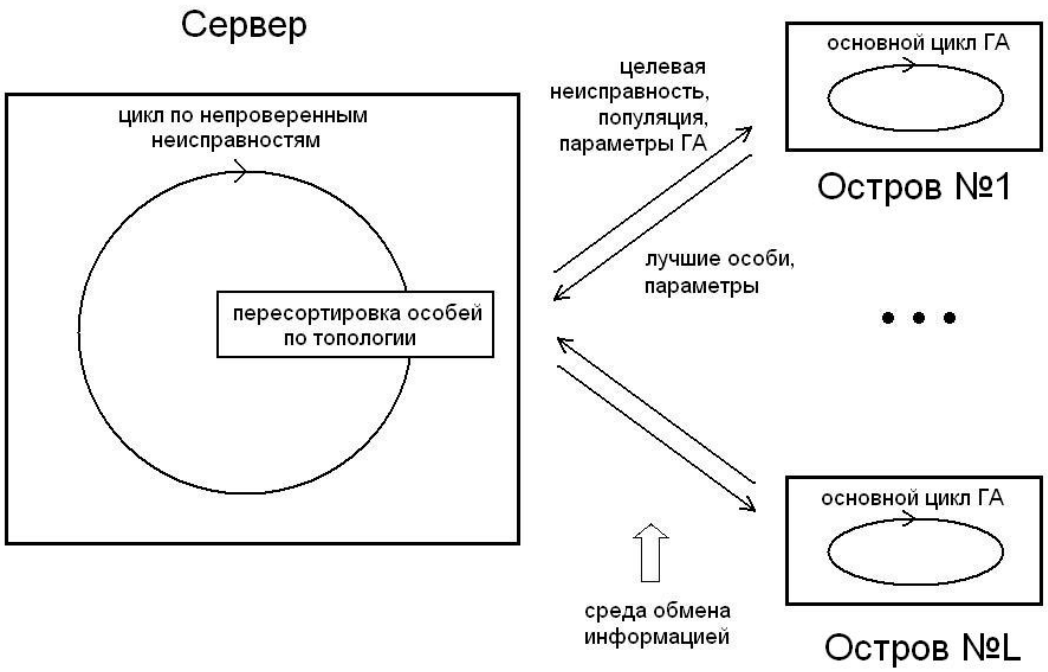


Рис.4.11. Структура ПГА модели «островов» с реализацией топологии связи на сервере.

1) ввод описания ЦУ A_0;
2) поиск доступных процессоров-«клиентов»;
3) передача описания ЦУ A на процессоры-«клиенты»;
4) итеративный цикл выбора целевой неисправности $f_{цел}$;
5) передача целевой неисправности $f_{цел}$ и начальной популяции $Pop_{нач}$ клиентам;
6) реализация стратегии взаимодействия клиентов при поиске тестовой последовательности для неисправности $f_{цел}$, которая включает топологию взаимодействия, степень миграции и время изоляции клиентов;
7) формирование отчёта.

Аналогично, при реализации нижнего уровня алгоритма построения проверяющих тестов, выделяются следующие функции клиентов (островов):

1) поиск сервера;
2) получение описания ЦУ A_0;

3) получение целевой неисправности $f_{цел}$ и начальной популяции $Pop_{нач}$;
4) итеративная реализация основного цикла ГА построения новых популяций при поиске теста;
5) реализация взаимодействия с сервером при поиске теста, которое включает стратегию отбора и удаления особей, реализацию степени миграции особей.

При использовании данной схемы для построения распределённого ГА поиска тестов для ВС с распределённой памятью в качестве среды обмена информацией выступают высокоскоростные линии связи между узлами, которые наиболее часто реализуются с помощью стека протоколов TCP/IP. При реализации алгоритма в этом случае процедуры взаимодействия реализуются с помощью технологии блокирующих сокетов.

В том случае, когда строится алгоритм ПГА для ВС с общей памятью, нет необходимости организовывать процедуры взаимодействия. Необходимые параметры в этом случаем могут передаваться как параметры потоков исполнения, либо как внешние переменные.

Основной проблемой при построении параллельных версий алгоритмов является точная и корректная реализация взаимодействия его компонент. Опишем подробно механизм такого взаимодействия в разрабатываемом подходе. Для этого выделим в структурных компонентах (сервер, клиент) состояния взаимодействия.

Для сервера основные состояния имеют следующую семантическую нагрузку.
1) Подготовка к очередной итерации работы ГА на клиентах: выбор очередной целевой функции; построение начальной популяции и передача этих данных на острова.
2) Опрос состояний результатов поиска на островах.
3) Реализация обмена характеристиками процесса поиска и лучшими особями между островами, основываясь на реализуемой топологии.
4) Получение теста и лучших особей с острова, который нашёл решение.
5) Моделирование с неисправностями на новом тесте с целью определения дополнительно проверяемых неисправностей.

После этого строится граф переходов между состояниями серве-

ра. Он изображён на рис.4.12. Видно, что в работе сервера выделяются два вложенных цикла:

- верхний цикл по списку непроверенных неисправностей;
- вложенный цикл по итерациям обмена с клиентами.

Также легко видеть, что вершинами обмена для сервера в такой интерпретации являются вершины 1, 2, 3, 4. При реализации обмена данными между сервером и клиентами данные вершины будут являться точками синхронизации.

Аналогичный граф возможных состояний клиента показан на рис.4.13. Смысл попадания процесса в вершины графа следующий.

1) Начало работы над новой целевой неисправностью: получение неисправности и начальной популяции.
2) Основной цикл ГА построения новых популяций.
3) Поиск не завершён, но наступило время обмена данными с другими островами. Передача лучших особей и характеристик ГА на сервер для обмена.
4) Тест найден на данном острове. Передача теста, лучших особей и харак-

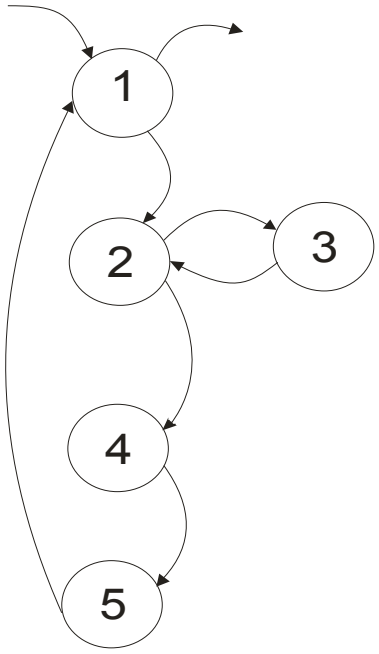


Рис.4.12. Граф состояний сервера ПГА построения тестов.

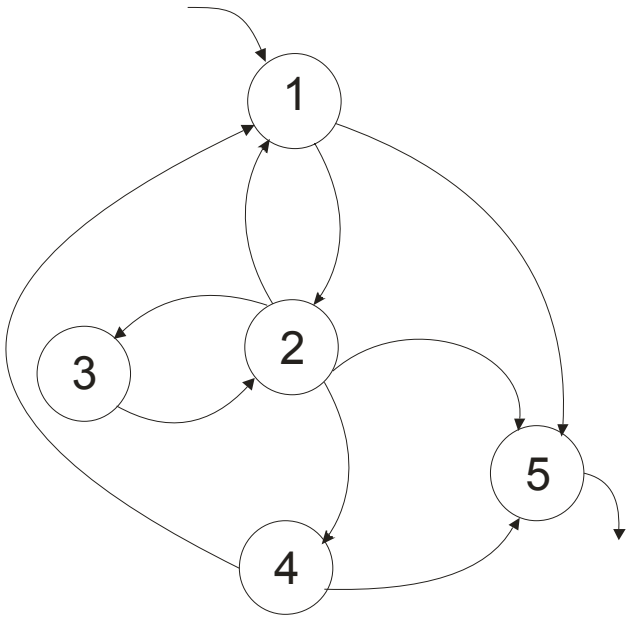


Рис.4.13. Граф состояний клиента ПГА построения тестов.

теристик ГА на сервер.

5) Все возможные неисправности-цели рассмотрены, сервер передал сигнал на остановку.

Таким образом видно, что вершинами обмена информацией с сервером являются вершины 1, 3, 4, которые и являются вершинами синхронизации.

Исходя из развития процесса поиска сервер управляет переходами между состояниями клиентов. Это осуществляется путём передачи информационного тега, следом за которым, возможно, происходит обмен необходимой информацией.

Множество возможных тегов сервера $Tg_{серв} = \{1,2,3,4,5\}$. Семантика тегов заключается в следующем:

«1» - начало работы; клиент должен получить описание ЦУ A_0 и параметры ГА: размер популяции, максимальное число итераций построения популяций, число итераций между обменом информацией, вероятности скрещивания и мутации и т.д.

«2» – начало работы над новой неисправностью; клиент должен получить целевую неисправность $f_{цел}$ и начальную популяцию $Pop_{нач}$;

«3» - поиск не завершен, обмен лучшими популяциями с сервером; клиент должен переслать установленное число лучших особей на сервер и получить лучших особей топологически связанного другого клиента с сервера;

«4» - остановка поиска, тест найден/не найден, ожидание новой неисправности;

«5» - завершение работы клиентов.

Построение модели обмена с синхронизацией сервера и клиентов во всех точках взаимодействия позволяет обеспечить его непротиворечивость, т.е. исключаются случаи, когда и сервер и клиент находятся в одинаковых состояниях ожидания (либо передачи) данных.

Топология связи клиентов реализуется с помощью матрицы смежности. Если в работе участвует p клиентов, то матрица имеет вид:

$$D = \begin{pmatrix} d_{11} & \dots & d_{1p} \\ & \dots & \\ d_{p1} & \dots & d_{pp} \end{pmatrix}, \qquad (4.4)$$

где элемент

$$d_{ij} = \begin{cases} 1, \textit{если клиент с номером } i \textit{ передаёт особей клиенту с номером } j; \\ 0, \textit{в противном случае.} \end{cases}$$

Пример матриц смежности для топологий с рис.4.10 и размерности $p = 6$ представлен на рис.4.14.

Остановимся теперь подробно на работе сервера.

Сервер начинает работу с чтения описания устройства A_0 и построения списка неисправностей F. При этом сразу строится сжатый по эквивалентности список [1].

После поиска клиентов сервер отсылает им описание ЦУ. Также на все клиенты отсылаются параметры ГА: число особей в популяции, вероятности операций скрещивания и мутации, число итераций ГА, которые должны быть выполнены на острове перед обменом информации. Здесь реализуется стратегия однородности или неоднородности параметров субпопуляций.

Далее в соответствии с верхним уровнем стратегии построения тестов на сервере начинается цикл выбора целевых неисправностей.

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

а) б) в)

Рис. 4.14. Пример матриц смежности для топологий обмена особями с рис.4.10.

Из списка выбирается очередная непроверенная неисправность, которая называется целевой $f_{цел}$. Для неё строится начальная популяция $Pop_{нач}$. Стратегия построения начальной популяции полностью соответствует таковой в разделе 3.2.

Далее всем островам передаются данные для начала процедуры поиска: целевая неисправность $f_{цел}$ и начальная популяция $Pop_{нач}$.

После этого сервер переходит в режим ожидания данных от клиентов. По прошествии некоторого числа итераций ГА на островах они передают на сервер информацию о результатах поиска решения.

Если один из островов передал информационный тег клиента, показывающий, что тест найден, то с данного клиента принимается построенная тестовая последовательность $S_{тест}$, неисправность в списке F' отмечается как проверенная и прерывается цикл по текущей неисправности. Далее выполняются следующие действия:

- последовательность $S_{тест}$ добавляется в финальный тест S;
- выполняется дополнительное моделирование ЦУ A_0 на последовательности $S_{тест}$ и, в случае обнаружения новых неисправностей, обновляется список непроверенных неисправностей F'.

Если же ни один из клиентов не сообщил о построении теста для $f_{цел}$, то сервер будет ждать от островов лучших полученных особей. После их получения в соответствии с топологией обмена для каждой их таких особей определяется клиент, на который они будут отправлены. После передачи особей клиентам сервер увеличивает счётчик итераций для текущей неисправности и переходит в режим приёма тегов от клиентов.

Если за заданное число итераций тест для неисправности $f_{цел}$ не найден, то:

- все клиенты останавливаются путём посылки тега 4;
- неисправность отмечается как непроверенная и удаляется из списка F';
- сервер переходит к выбору очередной неисправности.

После исчерпания всего списка неисправностей, либо достижения максимального числа итераций поиск прекращается, клиентам отправляется тег «5» полного завершения работы, а сервер записывает найденный итоговый тест S в файл и формирует отчёт.

Описанный словесно метод работы сервера изображен ниже в виде укрупнённого псевдокода алгоритмической реализации.

Алгоритм А4.7

```
Сервер_распределённый_ГА(имя_ЦУ, Параметры)
{
  // если найдены клиенты, то начинается работа
  if( (p=ПоискКлиентов()) >0 )
  {
    A=ВводОписанияЦУ(имя_ЦУ);
    F=ПостроениеСпискаТестируемыхНеисправностей(A);
    F'=F;
    ИнициализацияПараметровГА(p);
    do_in_threads_in_parallel
    {
      ПередатьТег(1);
      ПередатьОписаниеКлиенту(A);
      ПередатьПараметрыГАКлиенту(ПАРАМЕТРЫ_ГА);
    } // конец выполнения потоков – точка синхронизации
    while( ЕстьНепроверенныеНеисправности() &&
           НеДостигнутоУсловиеОстановки() )
   {
      //также строится Pop_нач
      f_цел=ВыборЦелевойНеисправности(A, F, Pop_фин);
      do_in_threads_in_parallel
      {
        ПередатьТег(2);
        ПередатьЦелевуюНеисправность(f_цел);
        ПередатьНачальнуюПопуляцию(Pop_нач);
      } // конец выполнения потоков – точка синхронизации
      // сервер переходит в режим ожидания результатов
      ИТЕРАЦИЙ=0;
      while( f_цел->detected != detected && ИТЕРАЦИЙ < )
      {
        do_in_threads_in_parallel
        {
          ТегКлиента=ПолучитьТег();
          // если неисправность обнаружена
          if( ТегКлиента == 1 )
          {
            S_тест=ПолучитьТест();// получить тест
```

```
        Pop_фин=ПолучитьФинальнуюПопуляцию();
        //отметить неисправность как
        // проверенную
        f_цел->detected=detected;
       break;// прерываем цикл – тест найден
     } // конец if – клиент нашёл тест
   }   // конец do – получение тега с клиента
   // неисправность не обнаружена, но не достигнуто
   //максимальное количество итераций
   // – обмен лучшими особями
   do_in_threads_in_parallel
   {
     ПередатьТег(3); //сообщение об обмене особями
     ПолучитьОсобей ();
   }
   // определить какие особи будут
   // переданы каким клиентам
   СортироватьОсобейПоТопологии();
   do_in_threads_in_parallel
   {
     ПередатьОсобей ();
   }
   ИТЕРАЦИЙ++;
} // конец while – итерации для одной неисправности
do_in_threads_in_parallel
{
  ПередатьТег(4); // клиент ожидает неисправность
}
if( f_цел->detected == detected )// цикл закончился
                                 // построением теста
{
  S=ДобавитьВТест(S_тест);
  // Фаза 3 – дополнительное моделирование
  // с неисправностями
  МоделированиеСНеисправностями(A, F', s_тест);
  F'=ОбновитьМножество(F');
}
else // цикл закончился, но тест не построен
{
  ОтметитьКакНепроверяемую(f_цел);
  F'=УдалитьИзСписка(F', f_цел);
}
```

```
    }   // конец while – есть непроверенные неисправности
    // завершаем работу всех клиентов
    do_in_threads_in_parallel
    {
      ПередатьТег(5);// полное завершение работы клиентов
    }
  }     // конец if – есть клиенты-острова
}       // конец алгоритма-сервера
```

В данном коде переменная *Итераций* показывает количество итераций обмена лучшими особями между клиентами, переменная *p* показывает число найденных клиентов, т.е. число островов, которые будут производить поиск.

Проверка окончания цикла по непроверенным неисправностям в списке выполняется в функции *НеДостигнутоУсловиеОстановки ()*.

Отметим, что обмен данными сервера с островами не является требовательным к ресурсам процессора и среды передачи, хотя требует определённого времени. С целью ускорения данных процедур, такой обмен можно организовать параллельно с использование вычислительных потоков. Каждый поток работает со своим клиентом. В приведённом выше псевдокоде этот момент отмечен блоком *do_in_threads_in_parallel{}*. Будем подразумевать, что каждый такой блок включает в себя:

- запуск пула потоков;
- выполнение в каждом потоке функций, которые указаны в теле блока;
- ожидание завершения всех потоков;
- приостановку работы потоков из пула.

Перейдём теперь к описанию алгоритма работы клиентов в распределённой модели ГА построения тестов.

В рассматриваемой модели каждый узел вычислительного кластера за исключением сервера выступает в качестве острова-клиента. Именно на островах происходит поиск решения с помощью генетического алгоритма.

В отличие от непараллельной реализации в данном подходе необходимо реализовать механизм взаимодействия с сервером вычислений, что порождает ряд дополнительных состояний алгоритма.

Клиент начинает работу с поиска сервера, после чего получает от

него тег начала работы «1», а следом описание ЦУ A_0 и параметры ГА.

Далее клиент работает в цикле выбора действий, которые определяются множеством информационных тегов сервера $Tg_{серв}$. В зависимости от полученного тега клиент выполняет следующие действия:

- «2» - начало работы над поиском теста для новой выбранной сервером неисправности: клиент получает неисправность $f_{цел}$, начальную популяцию $Pop_{нач}$ и запускается основной цикл поиска ГА;
- «3» - обмен промежуточными результатами поиска с сервером: на сервер передаются лучшие особи из текущей популяции, а следом с сервера принимаются лучшие особи с других островов;
- «4» - остановка поиска: текущий цикл ГА завершается, переход к ожиданию новой целевой неисправности;
- «5» - завершение работы клиента.

Основной цикл работы ГА реализуется традиционно для данных типов алгоритмов. В описываемом подходе он реализован на основании алгоритма А3.4. Отличие заключается в том, что через заданное количество итераций происходит его остановка и обмен через сервер лучшими особями, полученными к текущему моменту. Завершив заданное число итераций ГА, формируется тег состояния клиента, который далее передается на сервер. Множество возможных тегов состояния клиента $Tg_{кл} = \{1,2\}$. Смысл числовых значений тегов заключается в следующем:

«1» - тест для неисправности $f_{цел}$ найден, передача на сервер теста, завершение обработки текущей неисправности и ожидание следующей;

«2» - завершено заданное число итераций ГА, тест не найден, необходим обмен особями и дальнейший поиск работы;

Алгоритмическая реализации описанного метода в виде псевдокода приведена ниже.

Алгоритм А4.8

```
Клиент_Распределённый_ГА()
{
  // начинаем работу, если найден сервер
  if( поиск_сервера() == найден )
  {
    ПолучитьТегСостояния();//тег всегда =1
    // - принять описание ЦУ и параметры ГА
    A=ПолучитьОписаниеЦУ();
    ПАРАМЕТРЫ_ГА=ПолучитьПараметрыГА();
    // бесконечный цикл ожидания состояний с сервера
    while(1)
    {
      ТЕГ_СОСТОЯНИЯ=ПолучитьТегСостояния();
      switch( ТЕГ_СОСТОЯНИЯ )
      {
        //начинаем работу с новой неисправностью
        case 2:
        { //начинаем работу с новой неисправностью
          f_цел=Получить_с_сервера_целевую_неисправность();
          Pop_нач=ПолучитьНачальнуюПопуляцию();
          Pop_тек=Pop_нач;
          break;
        }
        case 3:
        { // итерации не исчерпаны,
          // и другой остров не нашёл тест
          Pop_нов=ПринятьНовыхОсобейССервера();
          Pop_тек=ЗаменитьОсобейВПопуляцииНаНовые(Pop_нов);
          break;
        }
        case 4:
        {  // другой клиент нашёл тест - завершаем работу
          break;
        }
        case 5:
        {  // тег=5 - полное завершение работы
          exit();
        }
        // несуществующие теги состояний
        default: {};
```

```
        }
      }
      // основной цикл построения популяций
      // – алгоритм А2.9
      ГА_ПостроениеТеста(A, f_цел, Pop_тек, ЧислоИтераций)
      if( f_цел->detected == detected )
      {
        ПередатьТег(2);
        ПередатьТест(S_тест);  // получить тест
        ПередатьПопуляцию(Pop_тек);
      }
      else
      {
        ПередатьТег(1);
        ПередатьЛучшихОсобейНаСервер();
      } // конец цикла по if – неисправность обнаружена
    }   // конец while – цикл по состояниям
  }     // конец if – сервер найден
}       // конец алгоритма
```

В данном коде переменные имеют следующий смысл:

$Pop_{тек}$ - текущая популяцию;

$Pop_{нач}$ - стартовая популяция ГА, получаемая с сервера;

$Pop_{нов}$ - подпопуляция, которая принимается с сервера и замещает худших особей;

ЧислоИтераций - показывает число итераций, которое необходимо выполнить ГА перед обменом с сервером;

Структурно модель РГА для ВС с распределённой памятью реализуется как запуск на узлах кластера одной копии сервера и необходимого числа копий клиентов. Поскольку фаза поиска на клиентах соответствует фазе ожидания на сервере (и наоборот), то на узле ВС, который соответствует серверу также можно выполнять клиентский код. В этом случае РГА будет содержать $p+1$ компонент, где p - число узлов в кластере. Для ВС с общей памятью компоненты оформляются не как отдельные приложения, а как потоки исполнения.

Поскольку в качестве базового подхода был выбран двухуровневый метод построения тестов (раздел 3.2), то и условия применимости данного РГА это синхронные последовательностные ЦУ и оди-

ночные константные неисправности. Из описания видно, что предложенный метод не зависит от типа моделируемых неисправностей и может быть легко распространен на любой тип неисправностей и модель ЦУ, для которых разработан соответствующий алгоритм моделирования с неисправностями.

Глава 5

АВТОМАТИЗИРОВАННАЯ СИСТЕМА МОДЕЛИРОВАНИЯ И ИДЕНТИФИКАЦИИ АСМИД-EVOLUTION

Жизненный цикл современных ЦУ основан на использовании средств автоматизированного проектирования (САПР). Их применение позволяет достичь основную задачу – повышение эффективности производства ЦУ на всех этапах: проектировании, предпродажного тестирования, сопровождения эксплуатации, а также ремонта.

Для повышения эффективности использования САПР они должны содержать современные алгоритмы моделирования и диагностики. Новые алгоритмы, с одной стороны, позволяют сократить время разработки новых ЦУ. Другая их задача – позволить обработку схемы большей размерности, что является актуальным при непрерывном росте сложности дизайнов современных ЦУ.

В данном разделе описывается новая система автоматизированного моделирования и диагностики АСМИД-Evolution, которая содержит реализацию новейших эволюционных алгоритмов идентификации, предложенных в данной работе.

5.1. Назначение, функции, структура системы и взаимодействие компонент

Рассматриваемая система, с одной стороны, является продолжением предыдущих версий системы АСМИД, АСМИД-П, АСМИД-Е [165-172], наследуя, в частности, базовую внутреннюю структуру данных. С другой стороны, она содержит новую идеологическую составляющую, а именно – широкое использование эволюционных алгоритмов и параллельных вычислений в методах идентификации ЦУ. При этом с помощью эволюционных алгоритмов реализована как

традиционная функциональность системы (генерация проверяющих тестов), так и совершенно новая: построение широкого класса ИдП, алгоритмы проектирования энергоэффективных ЦУ. Применение эволюционных алгоритмов и параллельных вычислений позволяет системе проводить обработку больших проектов на логическом уровне представления.

Системы моделирования и идентификации линейки АСМИД разрабатываются в Институте прикладной математики и механики Национальной академии наук Украины. Последней является описываемая в настоящем разделе система АСМИД-Evolution. Основными функциями, которые реализуются системой, являются следующие:

- ввод описания ЦУ на логическом уровне;
- трансляция текстового описания ЦУ во внутреннюю структуру данных, представленную системой таблиц описания;
- логическое моделирование поведения исправного ЦУ на заданной входной последовательности; для моделирования используются 3-х и 16-и значный алфавиты;
- логическое моделирование поведения ЦУ в присутствии неисправностей на заданной входной последовательности с целью изучения её диагностических свойств; для моделирования используются 3-х и 16-и значный алфавиты;
- генерация входных ИдП следующих классов: тестовые для множества одиночных константных неисправностей, тестовые для неисправностей «задержка распространения сигнала», инициализирующие последовательности, последовательности для верификации эквивалентности двух заданных схем, диагностические последовательности;
- генерация избыточных тестов;
- оценка пикового одно-, n-тактного и устойчивого рассеивания тепла заданного ЦУ;
- просмотр результатов работы всех программ в текстовом виде;
- просмотр временных диаграмм поведения исправного ЦУ в графическом виде.

Функционально система разбита на несколько модулей, выполняющих законченные функции. Укрупнённая структура системы АСМИД-Evolution приведена на рис.5.1. В системе выделяются следующие компоненты (подсистемы).

1. Блок предварительной обработки.

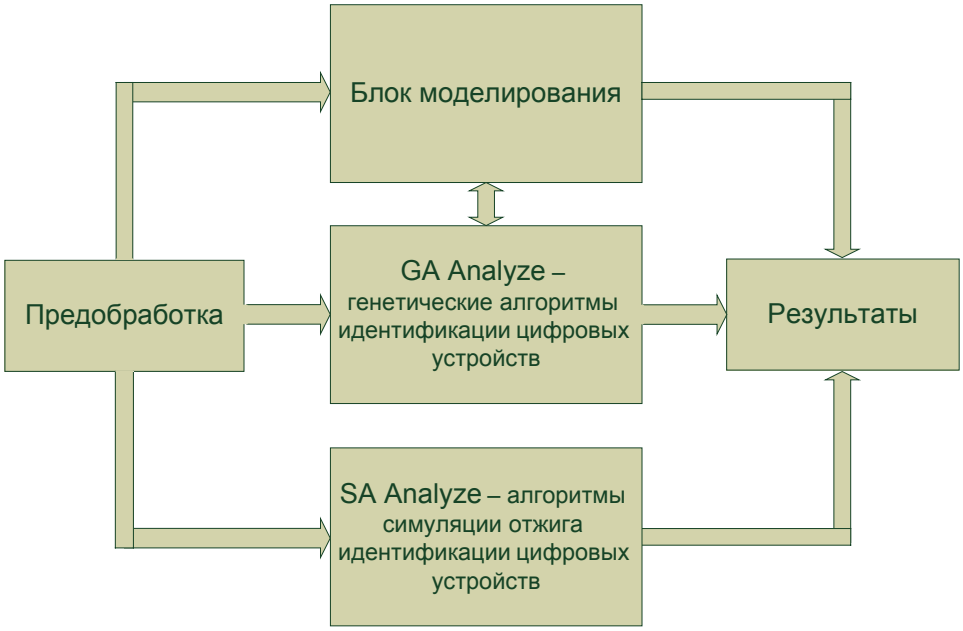


Рис.5.1. Общая структура системы АСМИД-Evolution.

2. Блок моделирования.
3. Подсистема GA-Analyze.
4. Подсистема SA-Analyze.
5. Блок постобработки и просмотра результатов.

Остановимся подробнее на описании функций, выполняемых каждой из подсистем. Подсистема предпроцессора (предварительной обработки) (рис.5.2) предназначена для подготовки описания схемы к дальнейшей работе. Она содержит следующие элементы:

- программа транслятор входного описания ЦУ во внутреннюю структуру данных;
- программа вычисления характеристик наблюдаемости и управляемости элементов схемы;
- программа построения полного списка неисправностей;
- программа сжатия списка неисправностей по эквивалентности.

Система ориентирована на текстовый ввод описания, поскольку именно такое описание генерируется автоматизированными системами из описания на функциональном уровне (VHDL, Verilog и т.п.). Пример автоматически сгенерированного текстового описания цифровой схемы s27 из набора ISCAS-89 приведён в главе 1.

Входными данными для подсистемы является текстовое описа-

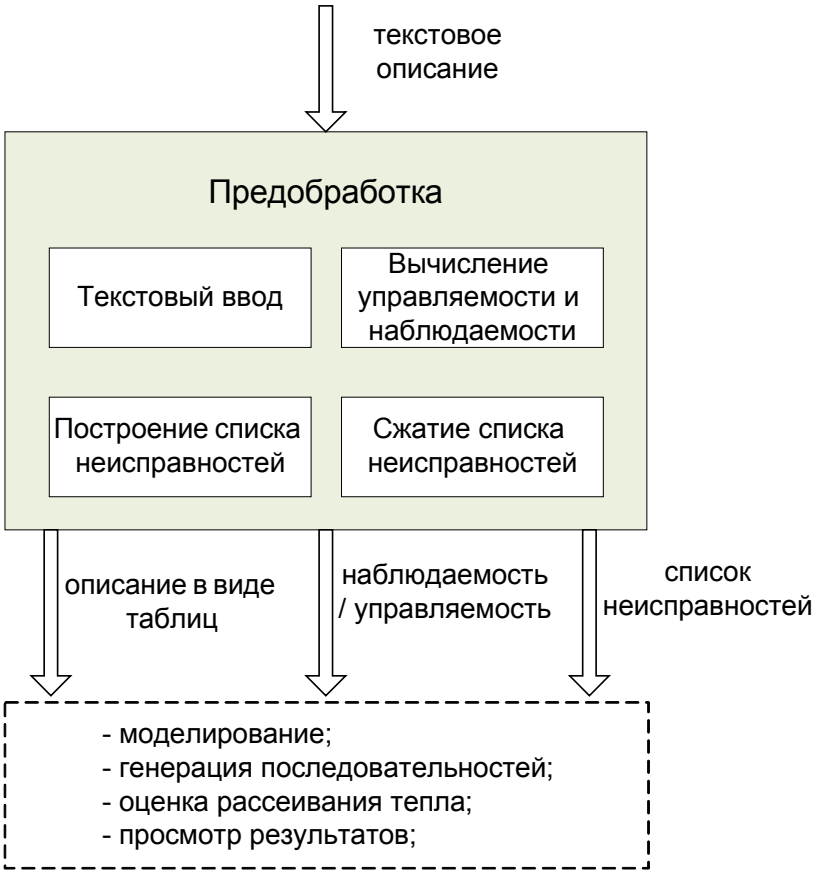


Рис.5.2. Подсистема предобработки и её входные/выходные потоки данных.

ние ЦУ в формате ISCAS-89 (файл с расширением *.ben).

Выходными данными подсистемы являются:

- файл описания ЦУ во внутреннем формате системы (расширение *.out);
- файл с числовыми данными о параметрах наблюдаемости и управляемости всех линий схемы (расширение *.cop);
- файл с полным/сжатым списком одиночных константных неисправностей (расширение *.f).

Результаты работы подсистемы используются во всех других подсистемах: блок моделирования, GA-Analyze, SA-Analyze, просмотр результатов работы.

Блок моделирования (рис.5.3) содержит следующие функциональные элементы:

- программа моделирования работы исправного ЦУ в 3-х и 16-и значном алфавитах;

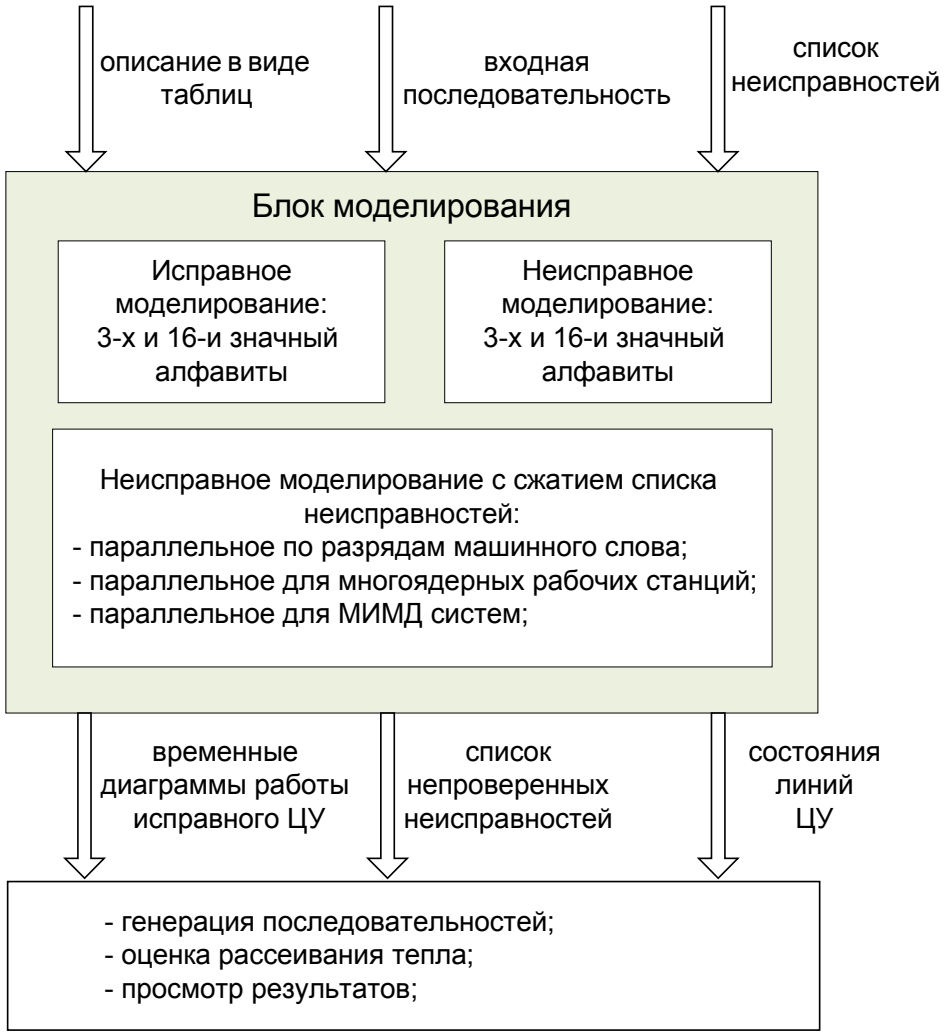


Рис.5.3. Подсистема моделирования и её входные/выходные потоки данных.

- программа моделирования работы ЦУ с неисправностями в 3-х и 16-и значном алфавитах; при этом каждая неисправность моделируется для каждого входного набора;
- программа моделирования работы ЦУ с неисправностями с сжатием списка неисправностей; используется параллельное по разрядам машинного слова одиночное распространение неисправностей; программа предназначена для одноядерных рабочих станций и служит для быстрого определения диагностических свойств заданной входной последовательности;
- программа параллельного моделирования поведения ЦУ с неисправностями для многоядерных (многопроцессорных) рабочих станций с общей памятью;

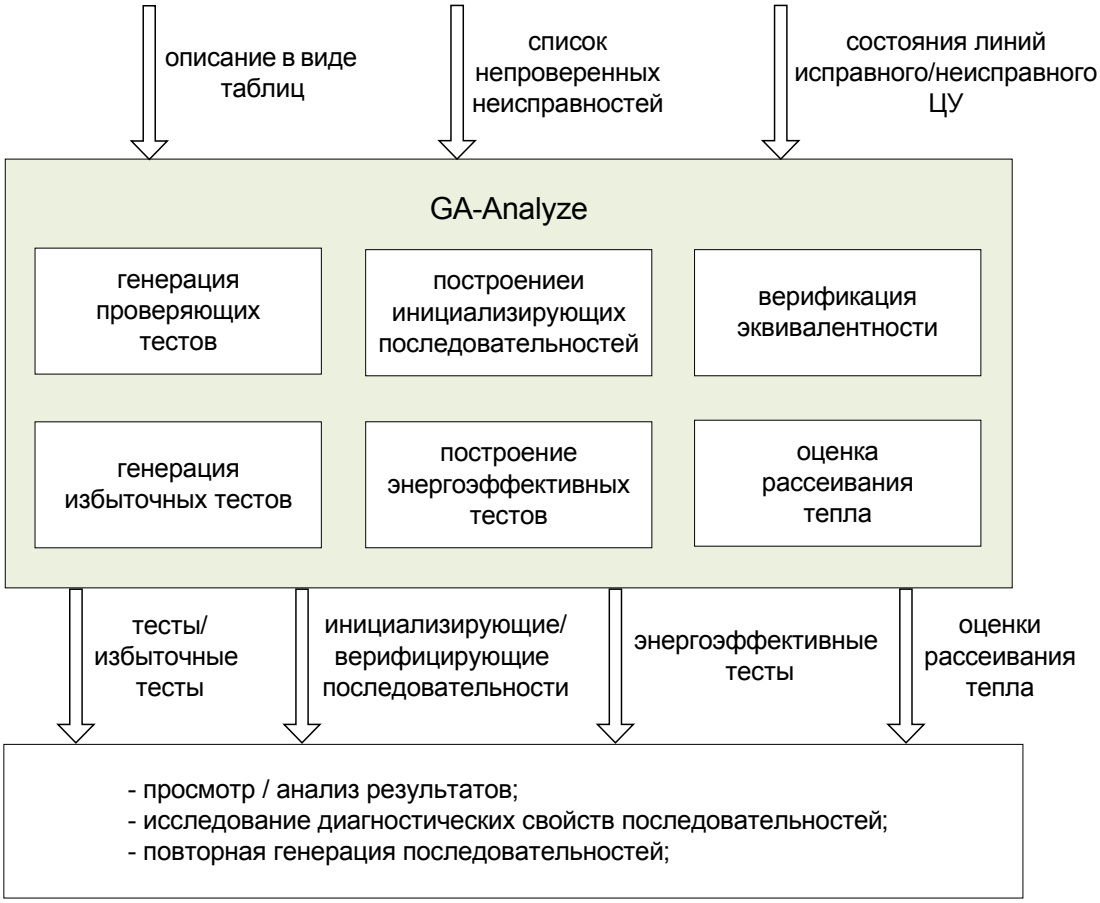


Рис.5.4. Подсистема GA-Analyze и её входные/выходные потоки данных.

- программа параллельного моделирования поведения ЦУ с неисправностями для вычислительной системы с распределённой памятью (локальной сети).

Выходными данными подсистемы являются:

- файл с состояниями линий схемы при работе на заданной входной последовательности (расширение *.rea);
- списки неисправностей, которые не проверяются заданной входной последовательностью.

Дополнительно все процедуры моделирования позволяют получать состояния внутренних линий моделируемого ЦУ в произвольный момент времени. Эти данные используются в эволюционных процедурах построения ИдП и оценки рассеивания тепла для вычисления оценочных функций.

В подсистеме GA-Analyze (рис.5.4) в качестве основного средства оптимизации выбран генетический алгоритм. Она содержит следую-

щие функциональные компоненты:
- ГА-метод генерации проверяющих тестов ЦУ;
- ГА-метод генерации диагностических тестов ЦУ;
- ГА-ориентированный метод построения последовательностей для логической инициализации ЦУ;
- ГА-метод верификации эквивалентности поведения двух заданных ЦУ;
- ГА-метод генерации избыточных тестов;
- ГА-ориентированный метод оценки пиковых показателей рассеивания тепла ЦУ;
- ГА-метод выбора субоптимального подмножества тестов с минимальным рассеиванием тепла (энергоэффективных тестов).

Результатом работы подсистемы являются:
- файл с проверяющим тестом (расширение *.tst) и списком непроверенных неисправностей (расширение *.f);
- файл с диагностическим тестом (расширение *.dts);
- файл с инициализирующей последовательностью (расширение *.ini);
- файл с верифицирующей последовательностью (расширение *.ver);
- файл с избыточными тестами (расширение *.rts);
- файл с оценкой рассеивания тепла для заданного множества последовательностей (расширение *.rpw);
- файл с энергоэффективным тестом (расширение *.ets);
- числовые оценки параметров рассеивания тепла (*.pwr).

Подсистема SA-Analyze имеет такую же структуру, как и подсистема GA-Analyze, а также работает с аналогичными входными и выходными данными. Отличием является то, что алгоритмы, реализующие её функции, основаны на оптимизационной стратегии симуляции отжига. Включение в систему подсистем с аналогичными функциями, но с различным алгоритмическим наполнением повышает гибкость разработчика в выборе компонент для работы. В связи с тем, что структура подсистемы SA-Analyze аналогична подсистеме GA-Analyze, мы не приводим её описание.

Подсистема постобработки и просмотра результатов включает следующие компоненты (рис.5.5):
- программа просмотра таблиц внутреннего формата описания ЦУ;
- программа просмотра временных диаграмм работы исправного

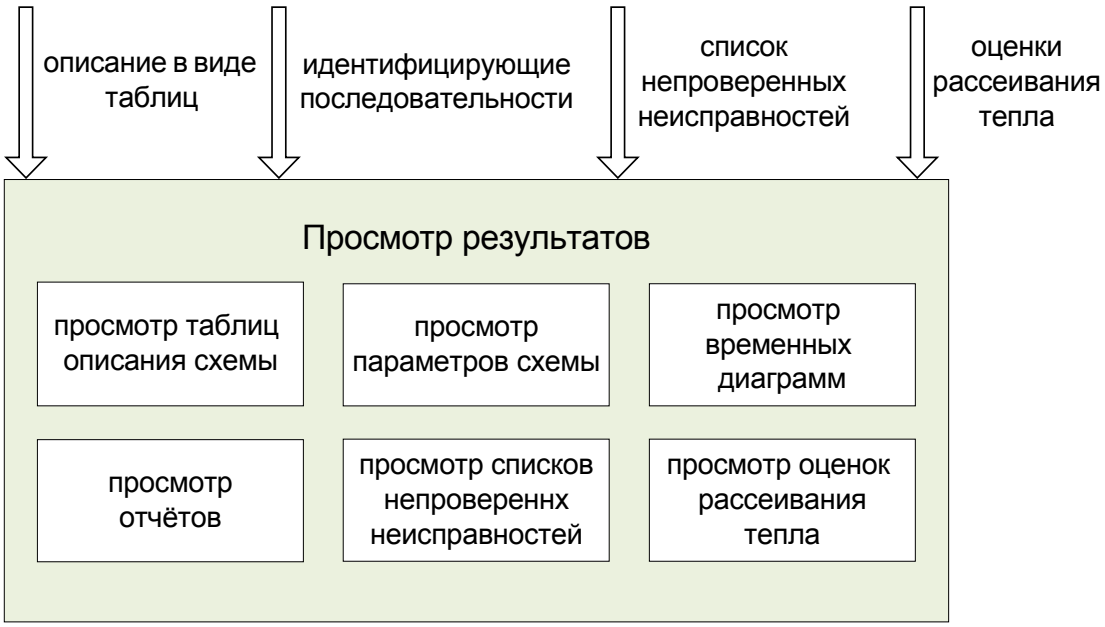


Рис.5.5. Подсистема просмотра результатов.

ЦУ;

- программа просмотра параметров наблюдаемости и управляемости ЦУ (*.cop);
- программа универсальная программа просмотра файлов всех форматов, использующихся в системе: тесты (*.tst), диагностические тесты (*.dts), списки неисправностей (*.f), инициализирующие последовательности (*.ini), верифицирующие эквивалентность последовательности (*.ver), избыточные тесты (*.rts), данные о рассеивании тепла набора последовательностей (*.rpw), оценки рассеивания тепла (*.pwr)

Подсистема не генерирует новые файлы, а только предоставляет пользователю результаты работы для анализа.

5.2. Структуры данных системы

Структуры данных системы включают:

- структуру данных описания ЦУ в виде системы связных таблиц;
- структуры динамических данных, которые создаются во время работы программных модулей;

- структуры представления данных в файлах.

Входным для системы является текстовое описание обрабатываемого ЦУ. В качестве исходного принят формат описания синхронных последовательностных ЦУ ISCAS-89 (глава 1).

Программой транслятором trans89.exe текстовое описание преобразуется во внутреннюю структуру данных описания ЦУ – систему связанных таблиц. Именно с данным представлением работают все программные модули системы. Описание ЦУ в системе хранится в двоичном виде в трёх следующих таблицах.

1. TYPES – содержит описание всех типов элементов, которые содержатся в обрабатываемом ЦУ. Размерность таблицы определяется переменной NUM_TYPES. Таблица содержит следующие поля описания типа элемента:
   - N_IN – число входов;
   - N_OUT – число выходов;
   - N_SOST – число элементов состояний;
   - ID – уникальный идентификатор.

   Базовыми типами логическими элементов согласно стандарта описания являются: AND, NAND, OR, NOR, XOR, NXOR, NOT, DFF, BUFF. Для первых шести из перечисленных базовых типов стандарт допускает варианты с различным числом входов.
2. EL – содержит описание всех элементов схемы. Размерность таблицы – NUM_EL. Таблица содержит следующие поля описания элемента:
   - NAME_EL – имя элемента;
   - R_TYPES – ссылка на таблицу TYPES, показывающая тип элемента;
   - R_LINKS – ссылка на таблицу LINKS, показывающая начало зоны контактов данного элемента в таблице.
3. LINKS – содержит списки контактов элементов в схеме и их связи друг с другом. Размерность таблицы определяется переменной NUM_LINKS. Таблица содержит следующие поля:
   - R_EL – ссылка строку таблицы элементов, которая его описывает;
   - CONT – ссылка на контакт-последователь в схеме; контакты, которые в схеме образуют один физический узел связаны данным полем в кольцо, что позволяет эффективно проводить пересылку значений сигналов по контактам, а также продвигаться

вперёд и назад по описанию схемы;

- SV – значения сигналов на контактах ЦУ; при использовании 3-значного алфавита данное поле содержит две компоненты кодирования, при использовании 16-значного – четыре;
- FLAG – битовые признаки состояний контактов; реализуемые значения и их интерпретация зависят от текущей программы, которая обрабатывает ЦУ

Каждому элементу в ЦУ в данной таблице отведено несколько строк описаний. Начало зоны контактов определяется ссылкой R_LINKS таблицы описания элементов EL. В начале зоны идут выходные контакты элемента схемы, после них – входные контакты.

В каждой таблице описания первые для удобства три строки описывают фиктивные элементы с фиксированными уровнями сигналов: 0, 1, u.

Пример заполненных таблиц описания ЦУ для указанной выше схемы S27 приведён ниже.

LINKS

N	R_EL	CONT	FLAG
0	0	-1	1
1	1	-1	1
2	2	-1	1
3	3	15	1
4	4	37	1
5	5	40	1
6	6	25	1
7	7	16	1
8	8	34	1
9	8	30	2
10	9	20	1
11	9	17	2
12	10	38	1
13	10	39	2
14	11	19	1
15	11	3	2
16	12	7	1
17	12	32	2
18	13	23	1

EL

NAME_EL	R_TYPES	R_LINKS
c0	0	0
c1	0	1
cu	0	2
G0	1	3
G1	1	4
G2	1	5
G3	1	6
YG17	2	7
G5	3	8
G6	3	10
G7	3	12
G14	4	14
G17	4	16
G8	5	18
G15	6	21
G16	6	24
G9	7	27
G10	8	30
G11	8	33

19	13	31	2	G12	8	36
20	13	10	2	G13	8	39
21	14	29	1			
22	14	41	2			
23	14	26	2			
24	15	28	1			
25	15	6	2			
26	15	18	2			
27	16	35	1			
28	16	24	2			
29	16	21	2			
30	17	9	1			
31	17	14	2	**TYPES**		
32	17	33	2	N	N_IN	N_OUT	ID
33	18	11	1	0	0	1	0
34	18	8	2	1	0	1	0
35	18	27	2	2	1	0	0
36	19	22	1	3	1	1	29
37	19	4	2	4	1	1	1
38	19	12	2	5	2	1	4
39	20	13	1	6	2	1	5
40	20	5	2	7	2	1	2
41	20	36	2	8	2	1	3

В процессе работы системы используется целый ряд структур представления данных. Одной из центральных является представление списка неисправностей, которое используется в программах моделирования с неисправностями и построения тестов. Список неисправностей представляет собой двунаправленный список (рис.5.6), который содержит следующие поля:

- «Контакт» - ссылка на номер контакта в таблице LINKS, к которому относится неисправность;
- «Тип» - определяет тип неисправности; принимаемые значения: «0» - для неисправности const0, «1» - для const1;
- «Проверена» - булево поле, которое показывает, проверена ли данная неисправность текущей входной последовательностью; принимаемые значения: «0» - не проверена, «1» - проверена;
- «Прервана» - булево поле которое показывает, прервано ли рас-

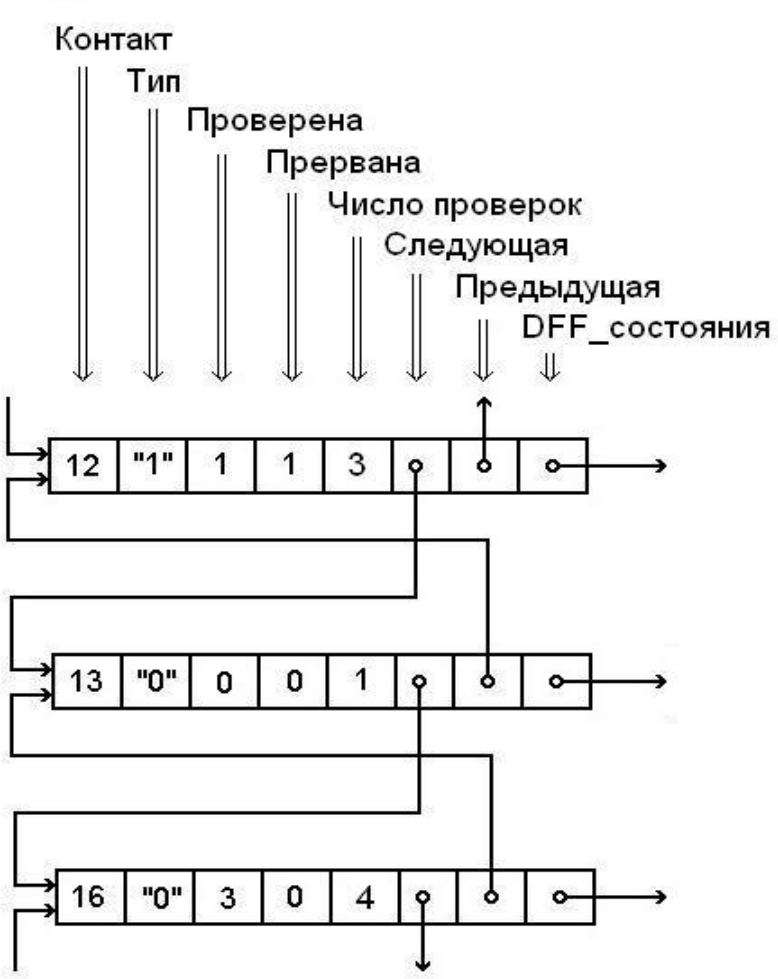


Рис.5.6. Структура списка неисправностей.

смотрение данной неисправности для текущей входной последовательности; принимаемые значения: «0» - не прервана, «1» - прервана; в алгоритмах моделирования с неисправностями прерывание происходит, когда неисправность классифицируется как высокоактивная [18]; в алгоритмах генерации тестов прерывание рассмотрения неисправности происходит в том случае, когда ГА не смог для неё построить тест;

- «Число проверок» - в алгоритмах избыточного тестирования показывает число подпоследовательностей, которые обнаружили данную неисправность;
- «Следующая» - указатель на следующую неисправность в списке;
- «Предыдущая» - указатель на предыдущую неисправность в списке;

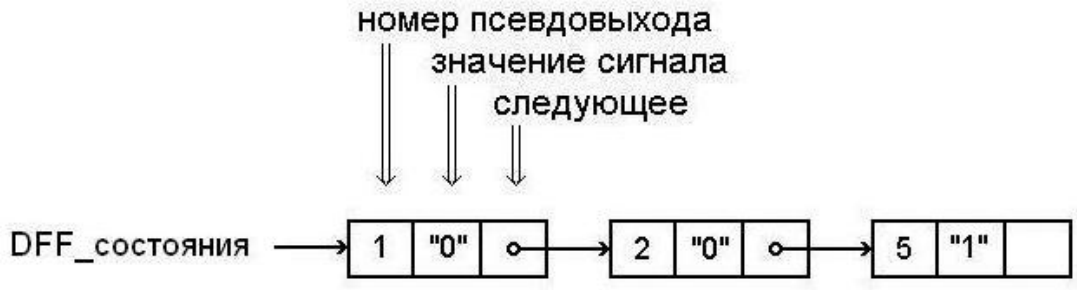


Рис.5.7. Список значений сигналов элементов состояний.

- «DFF_состояний» – значения сигналов на линиях, соответствующим элементам состояний (см. ниже).

Представление множества неисправностей в виде списка (в предыдущих версиях системы – в виде массива) придаёт гибкость при его обработке: возможно добавление/удаление неисправностей в произвольном месте списка. Двунаправленность списка позволяет эффективное продвижение по нему в обоих направлениях, а также позволяет полностью удалять запись о неисправности из него.

В ряде методов, реализованных в системе, возникает необходимость запоминать значения сигналов на линиях, которые соответствуют элементам состояний (триггерам). Это необходимо, в частности, в алгоритмах генерации тестов и алгоритмах параллельного моделирования с одиночным продвижением влияния неисправностей [8, 91-92, 173]. Для этого используется список значений сигналов элементов состояний (рис.5.7). Заметим, что в данном списке хранятся значения сигналов не для всех триггеров, а только для тех, чьи значения отличаются от значений соответствующих линий исправного ЦУ, полученных к текущему такту модельного времени. Такое решение позволяет существенно экономить память для больших списков, когда элементы состояний должны быть сохранены для десятков сотен тысяч неисправностей. Каждый элемент списка состояний содержит следующие поля:

- «Номер псевдовыхода» - номер элемента состояния, для которого сохранено значение;
- «Значение сигнала» - поле, показывающее запомненное значение;

принимаемые значения: 0, 1, *и*;
- «Следующее» - указатель на следующий элемент списка.

В системе для описания ЦУ и для хранения результатов работы используются файлы различных форматов.

Описание ЦУ хранится в файле с расширением *.out. Файл содержит три таблицы описания схемы, описанные выше, однако данные представлены в двоичном виде.

Идентифицирующие последовательности хранятся в файлах следующих типов:
- проверяющие тесты (*.tst);
- диагностические тесты (*.dts);
- инициализирующие последовательности (*.ini);
- верифицирующие эквивалентность последовательности (*.ver);
- энергоэффективные тесты (*.ets).

Все файлы указанных типов имеют одинаковый формат. Данные представлены в текстовом виде. Покажем формат описания на примере.

```
5 6 00100 01000 11011 10011 01000 00001
```

В данном описании:
- первое число соответствует числу входов ЦУ (ширина последовательности);
- второе число соответствует числу хранящихся входных двоичных наборов (длина последовательности);
- далее следует описание непосредственно входных наборов, которые разделены пробелом.

Видно, что такой формат описания последовательностей не фиксирует моменты модельного времени, в которые начинаются новые подпоследовательности. Однако в задаче построения энергоэффективных тестов такие данные являются существенными. Для этого в конец файла добавляется информация о длине каждой из подпоследовательностей. Файлы с описанием избыточных тестов имеют расширение *.rts. Пример такого файла приведён ниже.

```
5 7 00100 01000 11011 10011 01000 00001 10001 2 2
3
```

В данном примере общая длина последовательности – 7, длины подпоследовательностей 2, 2 и 3 соответственно. таким образом, формат описания данного файла обратно совместим с форматом *.tst.

Ещё одним новым форматом в системе является файл с описанием переключательной активности последовательностей - *.rpw. В файле в текстовом виде хранится оценки рассеивания тепла для каждой подпоследовательности, а также информация о проверяемых ими неисправностях. Пример данного файла приведён ниже.

```
   10
   32
   10 2.680000e+02 12 0 3 4 5 10 13 15 18 19 20 24
28
   10 2.780000e+02 21 0 1 2 3 5 7 10 11 12 13 14
15 16 17 18 19 20 23 25 28 29
   10 2.890000e+02 12 0 3 4 5 10 13 14 15 18 19 23
24
   10 2.840000e+02 12 0 3 4 5 6 10 13 15 18 20 24
28
   10 2.960000e+02 24 0 1 2 3 4 5 6 7 8 9 10 12 13
16 17 21 22 24 25 26 27 29 30 31
   10 2.790000e+02 11 0 3 5 10 14 15 18 19 20 23
28
   10 2.850000e+02 5 0 3 5 6 10
   10 2.760000e+02 12 0 3 4 5 10 13 14 15 18 19 23
24
   10 2.930000e+02 10 0 3 5 6 10 15 18 19 20 28
   10 2.850000e+02 25 0 1 2 3 5 6 7 8 9 10 11 12
13 15 16 17 18 19 20 21 22 25 27 28 29
```

В файле содержится следующая информация (в порядке появления):

- число рассматриваемых подпоследовательностей;
- число неисправностей в полном списке;
- далее для каждой подпоследовательности с новой строки: длина подпоследовательности, оценка рассеиваемой мощности в условных единицах, число проверяемых неисправностей, список номеров проверяемых неисправностей из полного списка.

Списки неисправностей хранятся в фалах с расширением *.f. Это может быть либо полный список, либо только те неисправности, которые не проверились к текущему моменту и будут рассматриваться на следующей итерации вызова программы генерации тестов. Пример файла со списком неисправностей приведён ниже.

```
Не проверяются неисправности:0
G1/1=0-?
G2/1=0
G3/1=0
YG17/1=1
XG5/1=0
ZG5/1=0-?
ZG5/1=1
XG6/1=1
ZG6/1=0-?
```

В первой строке указывается число непроверяемых неисправностей, которые перечисляются далее списком. Каждая строка списка содержит:

- имя логического элемента;
- разделитель «/»;
- номер контакта элемента; отчёт ведётся от выходного контакта, который имеет номер 0;
- разделитель «=»;
- тип рассматриваемой неисправности: «0» или «1»;
- символ «-?» добавляется при необходимости в том случае, если неисправность проверилась условно.

5.3. Программная реализация и работа с системой

Все алгоритмы, которые предложены в данном исследовании, реализованы программно. В качестве инструментальной платформы реализации использовалась среда CodeGear RAD 2007.

Объём программной реализации алгоритмов варьируется от 1000 строк кода (алгоритмы моделирования) до 4000 строк кода (двух-

уровневые алгоритмы генерации тестов).

Реализация включает 24 программных модуля, из которых 3 относятся к предварительной обработке, 2 – к просмотру результатов, остальные 21 непосредственно реализуют основную функциональность системы.

При программной реализации был выбран объектно-ориентированный подход. При этом подходе алгоритм построения ИдП представляется виде объекта. Его методами и свойствами являются элементы алгоритма. Поскольку в разных алгоритмах методы и свойства часто совпадают, такой подход даёт высокий уровень повторного использования кода.

Покажем, как это выглядит на практике. Рассмотрим ГА выбора подмножества последовательностей [174]. При объектно-ориентированном подходе данный ГА реализуется в виде отдельного потока исполнения и имеет следующие методы:

- *GAThread()* – процедура инициализации вычислительного потока;
- *GenerateStartPopulation()* – построение начальной популяции особей;
- *ValuePopulation()* – оценка особей в популяции;
- *ValueIndividual()* – оценка особи;
- *StopCriterionReached()* – проверка условия окончания работы;
- *ConstructNewPopulation()* – построение новой популяции из текущей.

Генетические операции над особями-двоичными последовательностями (рис.2.5-2.8) реализуются методами:

- *DoVerticalCrossingover()* – вертикальное скрещивание;
- *DoGorizontalCrossingover()* – горизонтальное скрещивание;
- *DoCutLineMutation()* – удаление набора с последовательности;
- *DoAddLineMutation()* – добавление набора в последовательность;
- *DoChangeLineMutation()* – изменение набора в последовательности.

Покажем, какие из приведённых методов могут быть использованы в других алгоритмах:

- *GenerateStartPopulation()* - во всех алгоритмах ГА построения ИдП;
- *ValuePopulation()* – во всех алгоритмах ГА построения ИдП;
- *ValueIndividual()* – в соответствующем алгоритме СО построения ИдП, в котором совпадает вид оценочной функции;

- *DoVerticalCrossingover()*, *DoGorizontalCrossingover()* – во всех ГА построения ИдП;
- *DoCutLineMutation()*, *DoAddLineMutation()*, *DoChangeLineMutation()* – во всех ГА построения ИдП, а также во всех алгоритма СО построения ИдП в качестве возмущающих операций.

Аналогичные замечания могут быть сделаны относительно процедур моделирования (для одноядерных систем, многоядерных систем и систем с распределённой памятью), которые используются для вычисления оценок особей и конфигураций в соответствующих алгоритмах.

В целом, такой подход позволяет существенно повысить скорость разработки новых программных модулей, реализующих эволюционные алгоритмы идентификации.

Для пользователя система представляется в виде головного модуля и ряда вызываемых программ.

Головной модуль содержит закладки, которые соответствуют основным подсистемам: «Хранилище схем», «Моделирование», «Генетический алгоритм», «Алгоритм симуляции отжига», «Отчёты и диаграммы».

Вкладка «Хранилище схем» (рис.5.8) предоставляет следующие возможности.

1. Вызов программы транслятора описания ЦУ во внутреннюю структуру данных, программа Trans89.exe.
   Входные данные: файл описания *.ben, результатом работы является двоичный файл с заполненными таблицами описания ЦУ *.out;
2. Вызов программы вычисления параметров наблюдаемости и управляемости ЦУ, программа Observer.exe, реализуется энтропийный подход вычисления данных параметров [1].
   Входными данными является файл описания ЦУ *.out, результатом работы двоичный файл *.cop с данными о параметрах.
3. Вызов программы построения полного списка неисправностей рассматриваемого ЦУ, программа FaultList.exe.
   Входными данными является файл описания ЦУ *.out, результатом работы файл с полным списком неисправностей *.f.

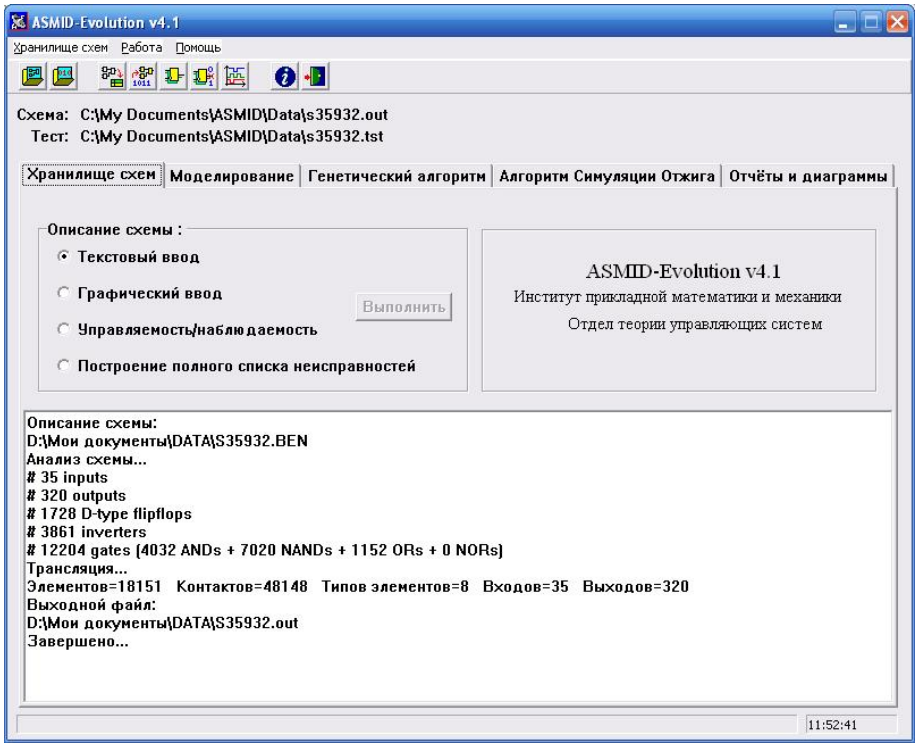


Рис.5.8. Вкладка «Хранилище схем» системы ASMID-Evolution.

Вкладка «Моделирование» (рис.5.9) соответствует «блоку моделирования» структурной схемы (рис.5.1) и содержит вызов следующих программных модулей, реализующих её функциональность.

1. Программа Mg16vn.exe моделирования поведения исправного ЦУ на заданной входной последовательности в 3-х значном и 16-и значном алфавитах.
   Входные данные: описание ЦУ - файл *.out, входная последовательность – файл *.tst. Выходные данные: файл со значениями сигналов на линиях ЦУ для каждого такта модельного времени *.rea.
2. Программа Mg16vf.exe моделирования с неисправностями в 3-х и 16-м значном алфавитах без динамического сжатия списка неисправностей.
   Входные данные: описание ЦУ - файл *.out, входная последовательность – файл *.tst. Выходные данные: файл со значениями сигналов на линиях ЦУ для каждого такта модельного времени *.rea, файл со списком непроверенных неисправностей *.f.
3. Программа Proofsb.exe моделирования с неисправностями для одноядерной рабочей станции с динамическим сжатием списка неис-

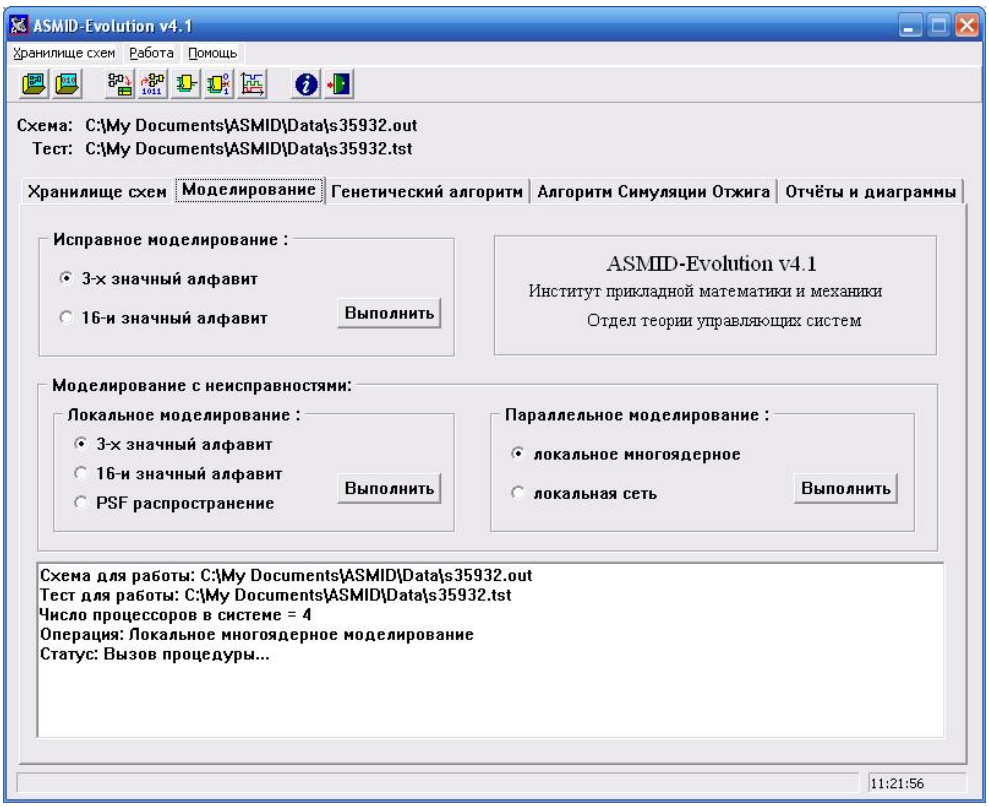


Рис.5.9. Вкладка «Моделирование» системы ASMID-Evolution.

правностей, реализует стратегию одиночного параллельного распространения неисправностей (PSFP). Цель данной стратегии – быстрая проверка диагностических свойст двоичной последовательности.

Входные данные: описание ЦУ - файл *.out, входная последовательность – файл *.tst. Выходные данные: файл со списком непроверенных неисправностей *.f. Файл с реакциями исправной схемы не строится.

4. Программа MCProofsParallelKernel5.exe моделирования с неисправностями для многоядерных рабочих станций с общей памятью.

   Входные данные: описание ЦУ - файл *.out, входная последовательность – файл *.tst. Выходные данные: файл со списком непроверенных неисправностей *.f. Файл с реакциями исправной схемы также не строится.
5. Программа ServerDProofs.exe, которая является серверным модулем алгоритма распределённого моделирования с неисправностями

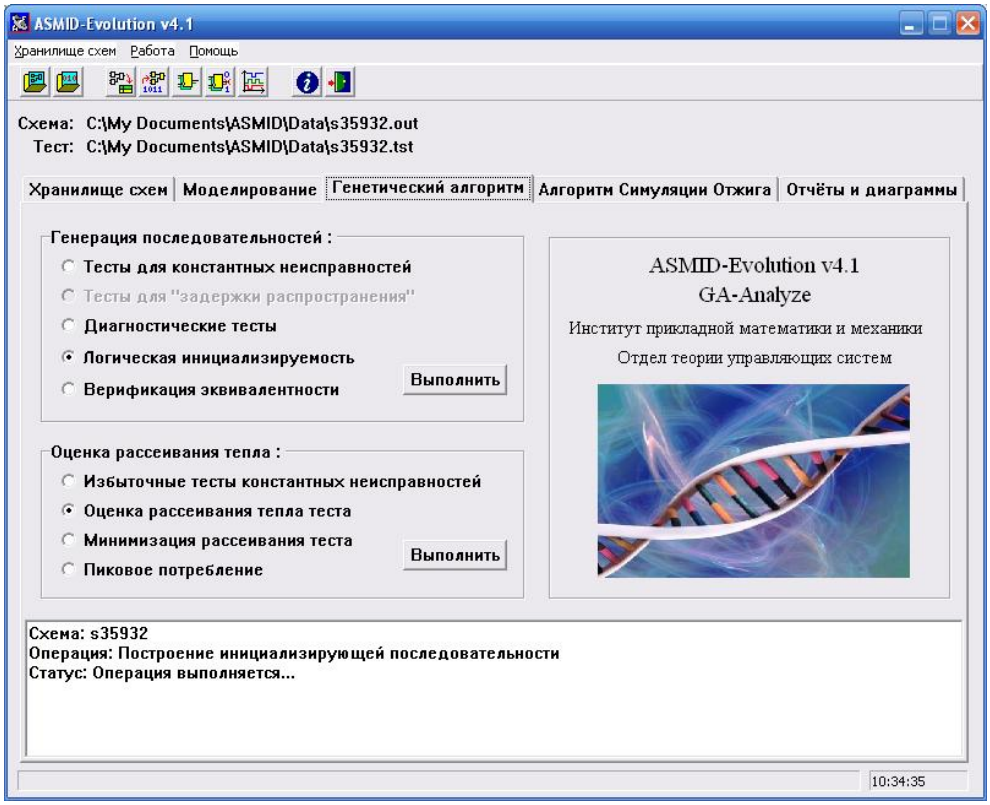


Рис.5.10. Вкладка «Генетический алгоритм» системы ASMID-Evolution.

для вычислительной системы с распределённой памятью - локальной сети. Данная программа взаимодействует с моделями ClientDProofs.exe, которые реализуют локальное моделирование с неисправностями на узлах ВС.

Входные данные: описание ЦУ - файл *.out, входная последовательность – файл *.tst. Выходные данные: файл со списком непроверенных неисправностей *.f. Файл с реакциями исправной схемы также не строится.

Вкладка «Генетический алгоритм» (рис.5.10) соответствует подсистеме «GA-Analyze». Из данной вкладки вызываются следующие программные модули.

1. Программа GenTest.exe, реализующая генетический алгоритм построения тестов (алгоритмы А3.2-А3.4).
   Входные данные: описание ЦУ - файл *.out. Выходные данные: тестовая последовательность – файл *.tst.
2. Программа DiagTest.exe построения диагностических тестов.
   Входные и выходные данные соответствуют программе

GenTest.exe.

3. Программа GALogicInitializer.exe построения инициализирующих последовательностей (алгоритм А2.3).
   Входные данные: описание ЦУ - файл *.out. Выходные данные: инициализирующая последовательность – файл *.ini.
4. Программа GAEquivalence.exe проверки эквивалентности двух заданных ЦУ (алгоритм 2.8).
   Входные данные: файлы *.out описания двух ЦУ. Выходные данные: отчёт об эквивалентности поведения.
   Если программа вызывается на многоядерной рабочей станции, то будет запущен модуль GAMultiCoreEquivalence.exe (алгоритм).
5. Программа RedundantGATestGenerator.exe построения множества избыточных тестов [174].
   Входные данные: описание ЦУ - файл *.out. Выходные данные: избыточные тесты – файл *.rts.
6. Программа PowerEstimator.exe оценки рассеивания тепла для набора подпоследовательностей.
   Входные данные: описание ЦУ - файл *.out, избыточные тесты – файл *.rts. Выходные данные: файл *.pwr с оценками рассеивания тепла и списками проверяемых неисправностей.
7. Программа PowerOptimizer.exe выбора субоптимального подмножества тестов с минимизированным рассеиванием тепла (алгоритм).
   Входные данные: файл *.pwr с оценками рассеивания тепла и списками проверяемых неисправностей. Выходные данные: файл с энергоэффективным тестов *.ets.
8. Программа PeakPowerEstimator.exe оценки пикового 1-тактного, пикового n-тактного и пикового устойчивого рассеивания тепла заданного ЦУ.
   Входные данные: описание ЦУ - файл *.out. Выходные данные: числовые оценки параметров.

Вкладка «Алгоритм симуляции отжига» соответствует подсистеме «SA-Analyze». Как было отмечено выше, данная подсистема функционально практически полностью аналогична подсистеме «GA-Analyze». На данной вкладке вызываются аналогичные программные модули, которые работают с теми же данными, что и «GA-Analyze». Сюда относятся: SALogicInitialization.exe – программа построения инициализирующих последовательностей заданного ЦУ;

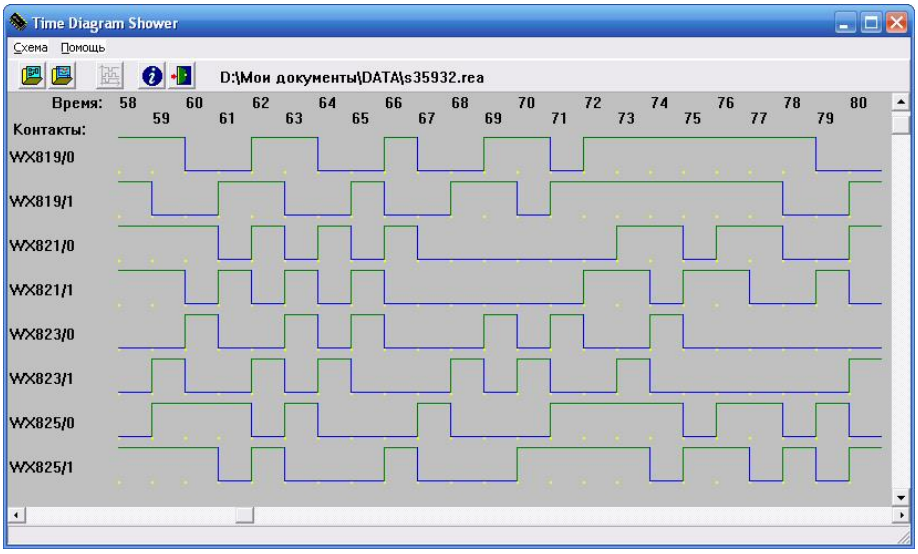


Рис.5.11. Программа просмотра временных диаграмм.

SAEquivalenceVerification.exe – программа верификации эквивалентности поведения двух заданных ЦУ; SATestGenerator.exe – программа построения тестов; SAPowerOptimization.exe – выбора набора подпоследовательностей с минимальным рассеиванием тепла.

Вкладка «Отчёты и диаграммы» соответствует подсистеме постобработки и просмотра результатов. Из данной вкладки вызываются следующие программные модули.

1. Программа Diagram.exe (рис.5.11), позволяющая просматривать временные диаграммы поведения исправного ЦУ.
   Входные данные: описание ЦУ - файл *.out, файл с реакциями исправного ЦУ *.rea.
2. Универсальная программа Viewer.exe (рис.5.12) просмотра файлов всех типов в системе, включая таблицы описания ЦУ и отчёты всех программных модулей.

5.4. Эксплуатационные характеристики системы

В данном разделе мы приведём численные результаты, которые характеризуют систему в целом. Апробация реализаций методов проводилась на наборе контрольных схем ISCAS-89. Для всех методов

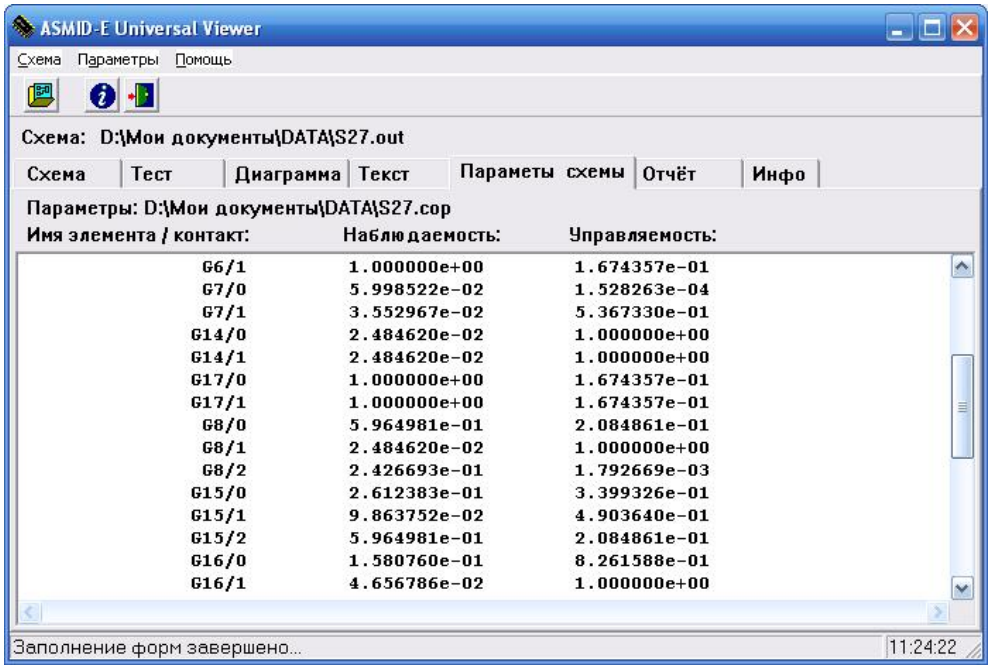


Рис.5.12. Программа просмотра отчётов.

общим замечанием является то, что выбор эвристических параметров осуществлялся таким образом, чтобы численные результаты были приемлемыми для всего набора схем. Программная реализация методов позволяет при запуске отдельно устанавливать значения параметров, влияющие на глубину перебора/время работы метода.

Также в зависимости от сложности метода и необходимости его применения результаты приводятся для различного подмножества ЦУ. Инструментальная платформа для разных методов также может быть различна, поскольку сами методы разрабатывались в разное время.

ГА-метод построения инициализирующих последовательностей.

Машинные эксперименты проводились на персональном компьютере с процессором Intel Celeron 1800Мгц, оперативной памятью 512Мб, в операционной системе Windows XP.

Результаты экспериментов позволили выбрать следующие значения констант алгоритма:

- $p_{мут} = 0.01$;
- $p_{скр} = 0.96$;

- $c_1 = 1$;
- $c_2 = \frac{N_{тр}}{N_{эл}}$, т.е. равна отношению числа триггеров к числу логических вентилей схемы, что позволяет сбалансировано учитывать активность внутри ЦУ при моделировании на заданной входной последовательности,
- $c_3 = 0.99$, уменьшение данной константы позволяло искать более короткие инициализирующие последовательности, однако при этом существенно увеличивается глубина поиска и, следовательно, затрачиваемое на поиск время;
- $L = 1$, поскольку многие схемы удаётся инициализировать даже последовательностями с одним входным вектором.

С приведёнными параметрами алгоритм запускался для всех схем из каталога ISCAS-89. Результаты проведённых экспериментов приведены в табл.5.1. Численные значения показывают, что все схемы, входящие в каталог либо очень легко инициализируются, либо, наоборот, построить для них последовательность практически невозможно даже при очень глубоком поиске.

Сравнение численных результатов с известными [103, 105, 175-176] показывает, что метод позволяет найти лучшие известные решения в терминах длины последовательности и числа инициализированных триггеров, либо даже улучшить их. При этом время работы оказывается существенно меньше в сравнении с точными методами.

Это говорит об эффективности подхода построения метода инициализации на основании ГА.

ГА верификации эквивалентности двух заданных ЦУ.

При проведении машинных экспериментов использовались следующие значения эвристических констант:

- $c_1 = 1$, если различие в поведении двух верифицируемых ЦУ достигнет внешнего выхода, то значение оценочной функции превысит единицу, что будет сигнализировать, что найдена требуемая входная последовательность;
- $c_2 = \frac{1}{N_{вых} \cdot N_{тр}}$, при таком выборе константы различие на всех линиях элементов состояний будет приравниваться к различию на

Таблица 5.1

Результаты машинных экспериментов работы ГА-метода построения инициализирующих последовательностей.

Имя схемы	Число триггеров в схеме	Число инициированных триггеров	Длина инициализирующей последовательности	Время работы генератора, сек	Рассмотрено вариантов при поиске
s344	15	15	2	0	1000
s349	15	15	2	0	1000
s382	21	21	1	0	1000
s386	6	6	2	0	1000
s510	22	0	-	4	5500
s526	21	21	2	0	1000
s635	32	12	1	5	6500
s641	19	19	1	0	1000
s713	19	19	1	0	1000
s938	32	0	-	4	5500
s953	29	10	1	5	6500
s967	29	10	1	5	6500
s991	19	0	-	4	5500
s1196	18	18	1	0	1000
s1238	18	18	1	1	1000
s1423	74	74	3	2	2500
s1488	6	6	1	0	1000
s1494	6	6	1	0	1000
s1512	57	53	2	8	7500
s3271	116	116	14	11	6000
s3330	131	131	3	6	4000
s3384	183	183	8	7	5000
s4863	104	104	2	3	2000
s5378	179	179	40	61	11500
s6669	239	239	5	19	7000
s9234	228	53	2	16	6500
s13207	669	207	42	191	15500
s15850	597	308	20	207	15000
s35932	1728	1728	1	8	1000
s38417	1636	372	15	459	12500
s38584	1452	1398	36	791	15500

одном внешнем выходе ЦУ, показывая высокую вероятность распространить данное отличие на какой либо внешний выход схемы в следующем такте времени;

- $c_3 = \frac{1}{N_{вых} \cdot N_{эл}}$, аналогично предыдущему: различие на выходах всех вентилей ЦУ приравнивается к различию на одном внешнем

выходе.

При проведении машинных экспериментов с целью определения эффективности метода была выбрана стратегия, применённая в [106]. Её суть основывается на следующих рассуждениях. Эффективность метода следует признать большей, если он может различить поведение в двух ЦУ, одно из которых получается с помощью некоторых минимальных изменений первого ЦУ. При этом для таких ЦУ последовательностная структура останется неизменной, а различия будут вноситься в некоторый комбинационный блок. Это также соответствует тому, что алгоритмы оптимизации ЦУ вносят изменения в такие блоки. Минимальным комбинационным блоком в ЦУ на логическом уровне представления является логический элемент. Тогда для получения второго ЦУ для сравнения необходимо в начальном ЦУ внести изменение в один логический элемент. В нашем случае мы изменяли тип произвольного элемента на случайно выбранный другой.

Далее для каждой пары построенных таким образом ЦУ алгоритм верификации запускался 25 раз и фиксировались результаты работы.

В качестве экспериментальной платформы использовался персональный компьютер со следующими характеристиками: процессор Intel CoreQuad с частотой 2.4Ггц, объем оперативной памяти 2Гбайта.

Результаты машинных экспериментов для некоторых больших схем каталога ISCAS-89 приведены в табл.5.2.

Поскольку предлагаемый алгоритм не является точным, то объясним смысловое различие колонок «Не различено схем» и «Ответ под вопросом». Данные в столбце «Не различено схем» показывают число экспериментов, в которых достигнут предел числа итераций, при этом не найдено различающей последовательности, а оценочная функция всех последовательностей популяции равна нулю. Данные в столбце «Ответ под вопросом» показывают число машинных экспериментов, в которых достигнут предел построения популяций, однако оценочная функция отлична от нуля. Таким образом, этот столбец соответствует случаю, когда функционирование схем различно внутри ЦУ, но данное различие не удалось распространить на внешние выходы схемы. При этом остаётся вероятность того, что при углублении поиска (увеличении числа предельно допустимых итераций алгоритма) зафиксированное различие в поведении двух схем удастся распространить на внешние выходы схем, т.е. построить различающую их последовательность.

Таблица 5.2

Результаты численных экспериментов для ГА-метода верификации эквивалентности последовательностных ЦУ.

имя схемы	эксперименты			
	всего	различено схем	не различено схем	ответ под вопросом
s298	25	25	0	0
s344	25	25	0	0
s349	25	25	0	0
s382	25	23	1	1
s386	25	25	0	0
s967	25	21	4	0
s1196	25	25	0	0
s1238	25	25	0	0
s1423	25	25	0	0
s1488	25	23	0	2
s1494	25	23	0	2
s3271	25	25	0	0
s3330	25	21	2	2
s3384	25	24	1	0
s4863	25	25	0	0
s5378	25	22	0	3
s6669	25	25	0	0
Всего:	425 (100%)	407 (≈95,76%)	8 (≈1,88%)	10 (≈2,35%)

Приведённые в табл.5.2 числовые данные показывают эффективность предложенного алгоритма: число экспериментов, в которых удалось построить различающую функцию, составило 95.76%, что является очень высоки показателем. Число экспериментов, в которых различие функционирования не проявилось даже внутри схемы, составило менее 2%. Для сравнения приведём данные по другим из-

вестным алгоритмам верификации эквивалентности: алгоритм VEGA – 88,35% [118], алгоритм AQUILA – 65,00% [119]. Результаты для двух последних упомянутых алгоритмов приводятся только для схем небольшой и средней размерности в силу ограничения самих алгоритмов.

В целом данные экспериментов также подтверждают высокую эффективность предложенного подхода.

Численные результаты по оценке роста скорости работы для параллельных версий данных ГА-методов приведены в главе 4.

ГА-построения тестов.

Поскольку в главе 3 описаны достаточно общие методы, на основании которых могут быть получены реализации целого ряда алгоритмов, то здесь мы приведём результаты одного из них, который основан на двухуровневом ГА-методе с активизацией неисправностей и отличается следующим моментом.

В качестве основного алгоритма моделирования для оценки качества особей-последовательностей был выбран алгоритм параллельного одиночного распространения неисправностей [8]. Применение метода совместно с ГА построения тестов имеет следующие особенности. Указанный метод моделирования без изменений применяется в псевдослучайной генерации последовательностей и дополнительном моделировании (см. главу 3). Для оценки особей-последовательностей в популяции был разработан фактически новый метод моделирования. Он также основан на параллельном распространении влияния неисправностей. Во всех разрядах машинного слова параллельно выполняется моделирование на различных особях-последовательностях, однако вносимая неисправность для них одна – целевая неисправность нижнего уровня метода. Такая модификация позволяет существенно ускорить оценку особей в популяции, особенно, если их число кратно разрядности машинного слова инструментальной ЭВМ.

Результаты машинных экспериментов для однопроцессорной ЭВМ приведены в табл.5.3. Аппаратная платформа включала процессор Intel CoreQuad с частотой 2.4Ггц, объем оперативной памяти 2Гбайта.

К сожалению, прямое сравнение численных результатов с известными затруднено [48]. С одной стороны это касается используе-

Таблица 5.3

Результаты работы программы ГА-метода генерации тестов.

Имя схемы	Всего неисправностей	полнота теста	длина теста	время генерации, час:мин:сек
s298	308	85.71	149	0:21
s344	342	96.20	192	1:01
s349	350	95.71	162	0:27
s386	384	73.95	606	5:34
s641	467	86.30	727	0:34
s713	581	81.93	677	3:34
s1196	1242	96.38	1911	1:15
s1238	1355	90.18	1183	0:55
s1488	1486	92.19	1975	6:28
s1494	1506	95.22	1274	3:31
s3271	3270	97.98	1125	1:58
s3330	2870	67.39	1357	16:04
s3384	3380	90.71	3191	37:28
s5378	4603	67.37	6913	22:00
s6669	6684	98.34	7640	44:56
s35932	39094	74.28	597	1:09:54

мых инструментальных платформ, которые зачастую относятся даже к различным поколениям. Второй проблемой являются различные применяемые эвристики, что не позволяет точно выделить именно роль ГА в решении задачи. Наконец, самым существенным различием в подходах является проблема начального состояния. Практически все зарубежные авторы ориентируются на то, что целевое ЦУ имеет аппаратный сброс с известным начальным состоянием, которое и используется в процессе генерации тестов. Разрабатываемый авторами подход является более общим. Принимается, что в начальный момент времени перед приложением каждой последовательности ЦУ находится в полностью неопределённом состоянии. Видно, что решаемая

Таблица 5.4

Характеристики параллелизации метода распределённого моделирования.

Имя схемы	T_1 час.:мин.:сек	T_8 час.:мин.:сек	H_8, раз	E_8	f_8
s9234	0:05:46	0:01:19	4.38	0.55	0.12
s13207	0:16:34	0:03:06	5.34	0.67	0.07
s15850	0:28:15	0:05:24	5.23	0.65	0.08
s35932	0:59:09	0:11:31	5.14	0.64	0.08
s38417	3:30:41	0:30:11	6.98	0.87	0.02

задача в такой постановке является существенно более сложной.

Оценка роста быстродействия ГА-метода построения тестов для ВС с общей памятью может быть произведена на основании экспериментов в главе 4. Для систем с распределённой памятью для проведения такой оценки для некоторых больших ЦУ из контрольного набора были проведены дополнительные эксперименты с алгоритмом распределённого моделирования [92].

Компоненты алгоритма (сервер и клиенты) запускались на вычислительном кластере, построенном по технологии Ethernet 100BaseT. Каждый узел представлял собой персональный компьютер с процессором Intel Celeron 2000Мгц, объём ОЗУ 256Мбайт, операционная система Windows XP.

Результаты машинных экспериментов приведены в табл.5.4. Здесь:

- T_1 - время работы метода при однопроцессорной реализации;
- T_8 - время работы метода при восьмипроцессорной реализации;
- H_8 - ускорение работы метода при восьмипроцессорной реализации;
- E_8 - эффективность использования ядер;
- f_8 - доля последовательного кода.

В целом метод позволяет добиться существенного роста быстро-

действия, и, следовательно, получить хорошие характеристики параллелизации. При этом низкое значение параметра доли последовательного кода показывает, что в соответствии с законом Амдаля [157] с ростом числа клиентов быстродействие также будет расти.

Видно также, что характеристики параллелизации улучшаются с ростом размерности ЦУ. Это объясняется тем, что в этом случае уменьшаются накладные расходы на передачу информации: описания ЦУ, тестовых последовательностей. списков неисправностей и т.п. При этом возрастает часть времени, которая затрачивается непосредственно на обработку ЦУ.

ЗАКЛЮЧЕНИЕ

В предложенной монографии рассмотрен вопрос применения генетических алгоритмов к решению задач построения входных идентифицирующих последовательностей цифровых устройств.

На основании известных методов и алгоритмов, а также работ автора, предложен единый подход к конструктивному построению таких методов. Он заключается в задании шаблонов одно- и двухуровневых ГА-методов генерации ИдП, а также структурных компонент таких шаблонов: кодирование особей и популяций, эволюционные операции, операции построения новых поколений и т.д. Поскольку методы широко используют эвристики, предложено кроме основной части выделять зависящие от реализации компоненты, конечный вид которых или их числовые значения определяются в конце исследования на основании машинных экспериментов. Конечная реализация ГА-метода строится на основании шаблона и зависящих от реализации компонент.

На основании данного подхода и двух моделей применения ГА разработан ряд одно- и двухуровневых методов построения входных ИдП. Применение подхода с использованием шаблонов позволяет существенно сократить время разработки новых методов построения ИдП. Также показано, что двухуровневые ГА построения ИдП на нижнем уровне используют соответствующие одноуровневые методы.

Показано конструктивное построение оценочных функций в данных методах на основе семантики качества решений-последовательностей.

Введены понятия функций достижения состояния элемента ЦУ и функций различия состояний двух ЦУ или множества ЦУ, которые отражают их функционирование на структурном уровне. Показано, что все динамические параметры, которые входят в оценки особей-последовательностей в разработанных методах, выражаются через данные функции.

Разработана методика построения параллельных версий предло-

женных ГА-методов. Для схемы «хозяин-рабочий» предложены схемы параллелизации для слабо-и сильнопараллельных ВС и исследованы характеристики масштабируемости новых ПГА. Для схемы «островов» РГА разработаны методы работы сервера и клиентов. Особенностью предложенного подхода является централизованное управление сервером всеми структурными компонентами. Это позволяет реализовывать произвольную топологию взаимодействия островов и стратегию адаптации их параметров. При этом за счёт вариации данных параметров, фактически, реализуется построение новых параллельных ГА-методов для доступного класса параллельных ВС. С другой стороны, достигается логическая непротиворечивость и беступиковое взаимодействие компонент в таких распределённых методах.

На основании рассмотренных в монографии ГА-методов показано построение новой версии системы моделирования и идентификации ЦУ. Её особенностью является ориентация на использование эволюционных алгоритмов оптимизации, которые являются основой реализации большинства методов построения ИдП. Использование ЭА в системе позволяет проводить обработку ЦУ большой размерности, сохраняя высокие эксплуатационные характеристики в терминах времени работы, полноты строящихся тестов, качества верификации и т.д.

СПИСОК ЛИТЕРАТУРЫ

1. Барашко А.С. Моделирование и тестирование дискретных устройств / А.С. Барашко, Ю.А. Скобцов, Д.В. Сперанский.- Киев: Наукова думка, 1992.- 288с.
2. Скобцов Ю.А. Логическое моделирование и тестирование цифровых устройств / Ю.А. Скобцов, В.Ю. Скобцов.- Донецк:ИПММ НАНУ, ДонНТУ, 2005.- 436с.
3. Богомолов А.М. Аналитические методы в задачах контроля и анализа дискретных устройств / А.М. Богомолов, Д.В. Сперанский.- Саратов: Из-во Саратов. ун-та, 1986.- 240с.
4. Основы технической диагностики / под ред. П.П. Пархоменко.- М: Энергия, 1976.- 463с.
5. Brgles F. Combinational profiles of sequential benchmark circuits / F. Brgles, D. Bryan, K. Kozminski // International symposium of circuits and systems, ISCAS-89. – 1989. – P.1929-1934.
6. Рабинович Ю.Г. Троичное моделирование БИС на функциональном уровне / Ю.Г. Рабинович // Автоматика и вычислительная техника (Рига).- 1982.- №1.- С.83-85.
7. Breuer M.A. Diagnosis and reliable design of digital systems / M.A. Breuer, A.D. Friedman. – Potomac, MD: Computer Sc. Press, 1976. – 308p.
8. Иванов Д.Е. Параллельное моделирование неисправностей для последовательностных схем / Д.Е. Иванов, Ю.А. Скобцов // Искусственный интеллект.- 1999.- №1.- С.44-50.
9. Pomeranz I. On fault simulation for synchronous sequential circuits / I. Pomeranz, S.M. Reddy // IEEE Transactions on Computers.- 1995.- №2.- P.335-340.
10. Биргер А.Г. Многозначное дедуктивное моделирование цифровых устройств / А.Г. Биргер // Автоматика и вычислительная техника (Рига). – 1982. - №4. – С.77-82.
11. Menon P.R. Deductive fault simulation with functional blocks / P.R. Menon, S.G. Chappel // IEEE Transactions on Computers. - 1978. - №8. - P.687-695.
12. Gai S. The Perfomance of the Concurrent Fault Simulation Algoritms in MOZART. / S. Gai, P.L. Montessoro, F. Somenzi // Proc. 25th Design Automation Conference. - 1988. – P.682-697.

13. Rogers W.A. Concurrent hierarchical fault simulation: a performance model and two optimizations / W.A. Rogers, J.F. Guzorek, J.A. Abraham // IEEE Transactions on Computer Aided Design.- 1987.- №5.- P.848-862.
14. Gai S. Advances in concurrent multilevel simulation / S. Gai S., F. Somenzi, E. Ulrich // IEEE Transactions on Computer Aided Design.- 1987.- №6.- P.1006-1012.
15. Cheng W.T. Differential fault simulation – a fast method using minimal memory / W.T. Cheng, M.-L. Yu // Proc. of the 26-th ACM/IEEE Design Automation Conference.- 1989.- P.424-428.
16. Thomson E.W. Digital Logic Simulation in a Time-Based, Table-Driven Environment – Part 2. Parallel Fault Simulation. / E.W. Thomson, S.A. Szygenda // Computer, IEEE Comp. Society.- 1975.- V.8.- №3.- P38-49.
17. Niermann T.M. PROOFS: A Fast, Memory-Efficient Sequential Circuits Fault Simulator / T.M. Niermann, W.-T. Cheng, J.H. Patel // IEEE Trans. CAD.- 1992.- V.11.- №2.- P.198-207.
18. Kung C.P. HyHope: A Fast Fault Simulator with Efficient Simulation of Hypertrophic Faults / C.P. Kung, C.S. Lin // Proc. of International Test Conference.- 1994.- P.714-718.
19. Lee H.K., Ha D.S. HOPE: An Efficient Parallel Fault Simulator for Synchronous Sequential Circuits // 29th Design Automation Conference. – 1992. – P.336-340.
20. Lee H.K., Ha D.S. New Method of Improving Parallel Fault Simulation in Synchronous Sequential Circuits. // Proc. Int. Conf. On Computer-Aided Design. – 1993. – P.10-17.
21. Характеристические последовательности в конечно-автоматных моделях дискретных устройств [Текст] / М. А. Бережная, Я. Ю. Королева // Вестник национального технического университета "ХПИ" : сб. науч. тр. : темат. вып. / Харьковский политехнический ин-т, нац. техн. ун-т. - Х. : НТУ “ХПИ”, 2008. - Вып. 56: Автоматика и приборостроение. - С. 19-25.
22. Fault-tolerant computing: Theory and techniques / Ed D.K.Pradhan. – Englewood clifts. N.Y.:Prentice Hall. – 1986. – vol.1. – 415p.
23. Niermann T., Patel J.H. HITEC: A Test Generation Package for Sequential Circuits // Proc. European Design Automation Conf. - 1991. - P.214-218.

24. Breuer M.A. A random and algorithmic technique for sequential circuits / M.A. Breuer // IEEE Transactions on Computers.- 1971.- №11.- P.1364-1370.
25. Мур Э.Ф. Умозаключительные эксперименты с последовательностными машинами.- В кн.: Автоматы. Под ред. Шеннона К.Э., Маккарти Дж.- М.:Иностр.Лит., 1956. - С.179-210.
26. Гилл А. Введение в теорию конечных автоматов. М.:Наука, 1966. – 272с.
27. Богомолов А.М., Барашко А.С., Грунский И.С. Эксперименты с автоматами. Киев:Наукова думка, 1973. – 144с.
28. Грунский И.С. Синтез и идентификация автоматов / И.С. Грунский, В.А. Козловский.- Киев: Наукова думка, 2004.- 245с.
29. Убар Р. Проектирование контролепригодных дискретных систем / Р.Убар.- Таллин:Из-во Таллиннского политехнического институтаю- 1988.- 68с.
30. Чжен Г., Менинг Е., Метц Г. Диагностика отказов цифровых вычислительных систем: перевод с анг. / Под ред. И.Б. Михайлова. – М.:Мир, 1972. –232с.
31. Sellers F.F., Hsiao M.Y., Bearnson L.W. Analysing errors with the boolean difference // IEEE Transactions on Computers. - 1967. - №5. - P.675-680.
32. Levendel Y., Menon P.R. The *-algorithm: critical traces for functions and CHDL constructs // Proc. of IEEE Test Conference. - 1983. - P.90-97.
33. Roth J.P., Bouricius W.G., Schneider R.P. Programmed algorithms to compute tests to detect and distinguish between failures in logic networks // IEEE Transactions on Electronic Computers. - 1968. - Vol.EC-16. - №7. - P.567-580.
34. Goel P., Rosales B.C. PODEM-X: An automatic test generation system for VLSI logic structures // 18th Design automation conference proceedings. - 1981. - P.260-268.
35. Fujivara H., Shimono T. On acceleration of test generation algorithm // IEEE Transactions on Computers. - 1983. - №12. - P.1137-1144.
36. Sziray J. Test calculation for logic networks by composite justification // Digital Process. - 1979. - №5. - P.3-15.
37. Cheng W.T., Chakraborty T.J. Gentest: An Automatic Test-Genaration System for Sequential Circuits. // Computer. – 1989. - v.22. – P.43-49.

38. Akers S.B. A logic system for fault test generation // IEEE Trans. Comput. - 1976. - №6. - P.620-630.
39. Скобцов Ю.А. Структурно-аналитический подход в задачах диагностики синхронных последовательностных схем / Ю.А. Скобцов, Д.В. Сперанский // Электронное моделирование.- 1980.- №4.- С.32-38.
40. Pomeranz I. The multiple observation time strategy / I. Pomeranz, S.M. Reddy // IEEE Transactions on Computers.- 1992.- №5.- P.627-637.
41. Cheng K.-T. A Simulation-Based Method for Generating Tests for Sequential Circuits / K.-T. Cheng, V.D. Agrawal, E.S. Kush // IEEE Transactions on Computers.- 1990.- V.39.- №12.- P.1456-1463.
42. Schulz M.H. ESSENTIAL: An Efficient Self-Learning Test Pattern Generation Algorithm for Sequential Circuits / M.H. Schulz, E. Auth // Proc. of Int. Test Conference.- 1989.- P.28-37.
43. Saab D.G. CRIS: A test cultivation program for sequential VLSI circuits / D.G. Saab, Y.G. Saab, J.A. Abraham // Proc. Int. Conf. Computer-Aided Design.- 1992.- P.216-219.
44. Srinivas M. A simulation-based test generation scheme using genetic algorithms / M. Srinivas, L. M. Patnaik // Proc. Int. Conf. VLSI Design.- 1993.- P.132-135.
45. Rudnick E.M. Application of Simple Genetic Algorithms to Sequential Circuit Test Generation / E.M. Rudnick, J.G. Holm, D.G. Saab, J.H. Patel // Proc. European Design & Test Conf.- 1994.- P.40-45.
46. Rudnick E.M. Sequential Circuit Test Generation in a Genetic Algorithm Framework / E.M. Rudnick, J.H. Patel, G.S. Greenstein, T.M. Niermann // Proc. Design Automation Conf.- 1994.- P.698-704.
47. Prinetto P. An automatic test pattern generator for large sequential circuits based on genetic algorithms / P. Prinetto, M. Rebaudengo, M. Sonza Reorda // Proc. Int. Test Conf.- 1994.- P.240-249, 1994.
48. Corno F. Experiences in the use of evolutionary techniques for testing digital circuits / F. Corno, M. Sonza Reorda, M. Rebaudengo // Proc. of Conf. Applications and science of neural networks, fuzzy systems, and evolutionary computation, San Diego CA.- 1998.- P.128-139.
49. Иванов Д.Е. Генерация тестов цифровых устройств с использованием генетических алгоритмов / Д.Е. Иванов, Ю.А. Скоб-

цов.- Труды института прикладной математики и механики НАН Украины.– Т.4.– Донецк, ИПММ.– 1999.– С.82-88.
50. Иванов Д.Е. Применение генетических алгоритмов при генерации тестов последовательностных устройств / Д.Е. Иванов, Ю.А. Скобцов // Вестник ТРТУ – ДонГТУ. Материалы второго научно-технического семинара «Практика и преспективы развития институционного партнёрства».- Донецк, ДонГТУ.- 2001, №1.- С.100-105.
51. Иванов Д.Е. Ускорение работы генетических алгоритмов при построении тестов / Д.Е. Иванов, Ю.А. Скобцов.- Искусственный интеллект.- №1, 2001.- С.52-60.
52. Иванов Д.Е. Генетические алгоритмы в генерации проверяющих тестов цифровых систем / Д.Е. Иванов, Ю.А. Скобцов, В.Ю. Скобцов.- Інформаційно-керуючі системи на залізничному транспорті.- №4, 2001.- С.52-55.
53. Hsiao S. Alternating Strategies for Sequential Circuit ATPG / M.S. Hsiao, E.M. Rudnick, J.H. Patel // Proceedings of European Design and Test Conference.- 1996.- P.368 – 374.
54. Иванов Д.Е. Генетические алгоритмы в диагностике и проектировании цифровых схем. / Д.Е. Иванов, Ю.А. Скобцов, В.Ю. Скобцов.- Искусственный интеллект.- №2, 2002.- С.250-258.
55. Saab D.G. Iterative [Simulation-Based+Deterministic Techniques]=Complete ATPG / D.G. Saab, Y.G. Saab, J. Abraham // Proc. Int. Conf. on Computer Aided Design.- 1994.- P.40-43.
56. Rudnick E.M Combining deterministic and genetic approaches for sequential circuit test generation / E. M. Rudnick and J. H. Patel // Proc. Design Automation Conf.- 1995.- P.183-188.
57. Hsiao M.S. Dynamic state traversal for sequential circuit test generation / M.S. Hsiao, E.M. Rudnick, J.H. Patel // ACM Transactions on Design Automation of Electronic Systems (TODAES).- Vol.5, Issue 3.-July 2000.- P.548 - 565.
58. Diagnostic Test Generation for Sequential Circuits / X. Yu, J. Wu, E.M. Rudnick // Proc. of International Test Conferenece.- 2000.- P.225-234.
59. Corno F. GARDA: a Diagnostic ATPG for Large Synchronous Sequential Circuits / F. Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda // Proc. of IEEE European Design and Test Conference, Paris, March 1999, pp.267 - 271.

60. Иванов Д.Е. Применение генетических алгоритмов при построении проверяющих тестов цифровых схем / Д.Е.Иванов, Ю.А.Скобцов, В.Ю.Скобцов. // Компьютерные науки и информационные технологии, Тезисы докладов международной конференции.–Саратов:Изд–во Сарат.ун–та.- 2002.- С.64–65.
61. Иванов Д.Е. Эволюционный подход к генерации проверяющих тестов цифрових систем / Д.Е.Иванов, С.А.Закусило, В.Ю.Скобцов, Ю.А.Скобцов // Труды конференций "Интеллектуальные системы" и "Интеллектуальные САПР".- Москва:Физматлит.- 2003.- С.76-81.
62. Skobtsov Y.A. Evolutionary approach to the test pattern generation for the sequential circuits / Y.A.Skobtsov, D.E.Ivanov // Радиоэлектроника и информатика.- 2003.- №3.- С.46-51.
63. Krishnaswamy D. Parallel genetic algorithms for simulation-based sequential circuit test generation / D. Krishnaswamy, M. Hsiao, V. Saxena, E.M. Rudnick, J.P.Patel // IEEE VLSI Design Conference, 1997.- P.475-481.
64. Иванов Д.Е. Сжатие списка неисправностей с помощью генетического алгоритма / Д.Е. Иванов, Ю.А. Скобцов.- Наукові праці Донецького Державного технічного університету, серія “Обчислювальна техніка та автоматизація», випуск 25.– Донецьк.– 2001.– С.161-167.
65. Миронов С. В. Генетические алгоритмы для сокращения диагностической информации / С.В. Миронов, Д.В. Сперанский // Автоматика и телемеханика.- 2008.- № 7.- С.146-156.
66. Иванов Д.Е. Эволюционный подход к функциональному тестированию цифровых схем / Д.Е. Иванов, Ю.А. Скобцов.- Наукові праці Донецького національного технічного університету. Серія: “Обчислювальна техніка та автоматизація” Випуск 74.- Донецьк:ДонНТУ.- 2004.- С.135-140.
67. Skobtsov Y.A. Evolutionary approach to the functional test generation for digital circuits / Y. A. Skobtsov, D. E. Ivanov, V. Y. Skobtsov, R. Ubar // In Proc. of 9th Biennial Baltic Electronics Conf., BEC 2004 (Tallinn, Oct. 2004).- Tallinn Univ. of Techn., 2004.- P.229-232.
68. Иванов Д.Е. Эволюционные методы построения проверяющих тестов для дискретных устройств / Д.Е. Иванов, Ю.А.Скобцов, В.Ю.Скобцов.- Вестник Томского государственного университета. Приложение.- №9(1), август 2004.- С.153-158.

69. Skobtsov Y.A. Evolutionary Approach to Test Generation for Functional BIST / Y.A. Skobtsov, D.E. Ivanov, V.Y. Skobtsov, R. Ubar, J.Raik // 10 European Test Symposium. Informal Digest of Papers.- May 22-25, 2005. Digest of Papers.- P.151-155.
70. Corno F. A Parallel Genetic Algorithm for Automatic Generation of Test Sequences for Digital Circuits / F. Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda // International Conference on High-Performance Computing and Networking, Brussels (Belgium).- 1996.- V.1067.- P.454-459.
71. Deepak A. An evolutionary multi population approach for test data generation / A. Deepak, P. Samuel // World Congress on Nature & Biologically Inspired Computing, 2009, NaBIC.- P.1451-1456.
72. Rivera W. Scalable Parallel Genetic Algorithms / W. Rivera // Artificial Intelligence.- Netherlands: Kluwer Academic Publishers.- 2001, Review 16.- P.153–168.
73. Hidalgo J.I. A Method for Model Parameter Identification Using Parallel Genetic Algorithms / J.I. Hidalgo, M. Prieto, J. Lanchares, F. Tirado, B. de Andrés, S. Esteban and D. Rivera // Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface.- London:Springer-Verlag, UK, 1999.- P.291 – 298.
74. Skobtsov Y.A. Distributed Fault Simulation and Genetic Test Generation of Digital Circuits / Y.A. Skobtsov, El-Khatib, D.E. Ivanov // Proceedings of IEEE East-West Design&Test Workshop (EWDT’06).- 2006.- P.89-94.
75. Skobtsov Y.A. Distributed Genetic Algorithm of Test Generation For Digital Circuits / Y.A. Skobtsov, El-Khatib, D.E. Ivanov // Proceedings of the 10th Biennial Baltic Electronics Conference.- Tallinn Technical University.- 2006.- P.281-284.
76. Иванов Д.Е. Распределённые генетические алгоритмы генерации проверяющих тестов цифровых систем / Д.Е. Иванов, Ю.А. Скобцов, А.И. Эль-Хатиб.- Радіоелектронні комп’ютерні системи. - ХАІ:2007.- №7.- С.176-181.
77. Иванов Д.Е. Распределенные генетические алгоритмы в построении тестов для цифровых схем / Д.Е. Иванов, Ю.А. Скобцов, А.И. Эль-Хатиб // Тезисы Докладов Международной научной конференции «Компьютерные науки и информационные технологии».- Саратов:изд-во СГУ.- 2007.- С.53-54.

78. Skobtsov Y.A. Evolutionary distributed test generation methods for digital circuits / Y.A. Skobtsov, D.E. Ivanov, V.Y. Skobtsov // Proc. of 8th International Workshop on Boolean Problems, Freiberg, Germany.- 2008.- P.213-218.
79. Skobtsov Y.A. Parallel Genetic Algorithm of Test Generation for Digital Circuits / Y.A. Skobtsov, El-Khatib, D.E. Ivanov // Proceedings of the International Conference "Modern problems of Radio Engineering, Telecommunications and Computer Science".- Lviv-Slavsko:2006.- P.129-131.
80. Иванов Д.Е. Распределенные алгоритмы моделирования и генерации тестов / Д.Е.Иванов, Ю.А.Скобцов, А.И. Эль-Хатиб.- Радіоелектронні і комп'ютерні системи.- ХАІ:2006.- №6.- С.97-102.
81. Corno, F. A Parallel Genetic Algorithm for Automatic Generation of Test Sequences for Digital Circuits / F. Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda // International Conference on High-Performance Computing and Networking, Brussels (Belgium), April, Lecture Notes In Computer Science.- Vol.1067.- 1996.- P.454-459.
82. Corno F. A portable ATPG tool for parallel and distributed systems / F. Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, E. Veiluva // Proc VLSI Test Symp, 1995.- P.29-34.
83. Intel® Software Insight. Multi-core Capability / R. Wirt.- USA: Intel Corporation, 2005, July.- 11p.
84. Ivask E. Distributed fault simulation with collaborative load balancing for VLSI circuits / E. Ivask, S. Devadze, R. Ubar // Scalable Computing: Practice and Experience.- 2011.- Vol.12, №1.- P.153–163.
85. Kochte M.A. Efficient Fault Simulation on Many-Core Processors / M.A. Kochte, M. Schaal, H.-J. Wunderlich, C.G. Zoellin // Proceedings of the 47th Design Automation Conference ACM, New York, NY, USA .- 2010.- P.380-385.
86. Krishnaswamy D. SPITFIRE: Scalable Parallel Algorithms for Test Set Partitioned Fault Simulation / D. Krishnaswamy, E.M. Rudnick, J.H. Patel, P. Banerjee // Proc. of 15th IEEE VLSI Test Symposium, 1997.- P.274-281.
87. Patil S. Parallel test generation for sequential circuits on general-purpose multiprocessors / S. Patil, P. Banerjee, J.H. Patel // Proc. Design Automation Conf.- 1991.- P.155-159.

88. Ravikumar C.P. Distributed Fault Simulation Algorithms on Parallel Virtual Machine / C. P. Ravikumar, V. Jain, A. Dod // VLSI Design.- 2001.- Volume 12, Issue 1.- P.81-99.
89. Ghosh, S. A distributed algorithm for fault simulation of combinatorial and asynchronous sequential digital designs, utilizing circuit partitioning, on loosely coupled parallel processors / S. Ghosh // Microelectronic Reliability.- 1995.- №35(6).- P.947- 967.
90. Muller-Thuns R.B. Portable parallel logic and fault simulation / R.B. Muller-Thuns, D.G. Saab, R.F. Damiano, J.A. Abraham // Digest of paper, International Conference on Computer Aided Design / Santa Clara, USA.-1989.- P.506-509.
91. Иванов Д.Е. Параллельный алгоритм моделирования цифровых схем с неисправностями для многоядерных систем с общей памятью / Д.Е. Иванов // Электронное моделирование.- 2011.- Т.33, №1.- С.93-106. (1.5 д.а.)
92. Иванов Д.Е. Распределённое параллельное моделирование цифровых схем с неисправностями / Д.Е. Иванов, Ю.А. Скобцов, Эль-Хатиб А.И.// Наукові праці Донецького національного технічного університету. Серія: “Обчислювальна техніка та автоматизація”. Випуск 107.- Донецьк:ДонНТУ.- 2006.- С.128-134.
93. Ivanov D. Parallel fault simulation algorithm of digital circuits for many-core workstations with common memory / D. Ivanov // Book of abstracts of 18th Conference on applied and industrial mathematics CAIM 2010.- Iasi University press, 2010.- P.46.
94. Pospichal P. Parallel Genetic Algorithm on the CUDA Architecture / P. Pospichal, J. Jaros, J. Schwarz // Proceedings of the 2010 international conference on Applications of Evolutionary Computation.- Berlin:Springer-Verlag, 2010.- Part I.- P.442-451.
95. Geronimo L. A Parallel Genetic Algorithm Based on Hadoop MapReduce for the Automatic Generation of JUnit Test Suites // L. Geronimo, F. Ferrucci, A. Murolo, F. Sarro // Proc. of 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, Montreal, Quebec Canada. - P.785-793.
96. Haghbayan M. H. Test Pattern Selection and Compaction for Sequential Circuits in an HDL Environment / M. H. Haghbayan, S. Karamati, F. Javaheri, Z. Navabi // Proceeding of ATS '10 Proceedings of the 2010 19th IEEE Asian Test Symposium.- IEEE Computer Society Washington, DC, USA ©2010.- P.53-56.

97. Hou Y. A New Method of Test Generation for Sequential Circuits Communications / Yanli Hou, Chunhui Zhao, Yanping Liao // Proceedings of International Conference on Circuits and Systems.- 2006.- P.2181-2185.
98. Corno F. SAARA: a simulated annealing algorithm for test pattern generation for digital circuits / F. Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda // Proceedings of the 1997 ACM symposium on Applied computing, San Jose, California.- 1997.- P.228-232.
99. Corno F. Exploiting the Selfish Gene Algorithm for Evolving Hardware Cellular Automata / F. Corno, M. Sonza Reorda, G. Squillero // CEC2000: Congress on Evolutionary Computation, San Diego (USA).- 2000.- P.1401-1406.
100. Скобцов Ю.А. Генерация тестов для последовательностных схем с использованием кратной стратегии наблюдения выходных сигналов / Ю.А.Скобцов, В.Ю.Скобцов, Ш.Н.Хинди // Науковий вісник Чернівецького університету, 2008.- Випуск 423. Фізика.Електроніка.- С.29-36.
101. Хинди Ш. Н. Иерархические эволюционные методы генерации тестов цифровых систем : дис. ... канд. техн. наук : 05.13.05 / Шукри Насри Али Хинди (Иордания) ; ГВУЗ "Донецкий национальный технический университет".- Донецк, 2010.- 141с.
102. Pixley C. Exact Calculation of Synchronization Sequences based on Binary Decision Diagrams / C. Pixley, S. Jeong, G. Hatchel // IEEE Trans. on CAD.- 1994.- V.13.- P.1024-1034.
103. Wehbeh J.A. On the Initialization of Sequential Circuits / J.A. Wehbeh, D.G. Saab // Proc. IEEE Int. Test Conf.- 1994.- P.233-239.
104. Wehbeh J.A. Initialization of Sequential Circuits and its Application to ATPG / J.A. Wehbeh, D.G. Saab // Journal of Electronic Testing: Theory and Applications.- 1998.- Vol.-13, №3.- P.259-271.
105. Corno F. Initializability Analysis of Synchronous Sequential Circuits / F. Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, G. Squillero // ACM Transactions on Design Automation of Electronic Systems.- 2002.- P.249-264.
106. Corno F. A Genetic Algorithm for the Computation of Initialization Sequences for Synchronous Sequential Circuits / F. Corno, P. Prinetto, M. Rebaudengo etc. // Proceeding ATS '01 Proceedings of the 10th Anniversary Compendium of Papers from Asian Test Symposium 1992-2001.- 2001.- P.213.

107. Xiaojing H., Zhengxiang S. Ant Colony Optimizations for Initialization of synchronous sequential circuits // IEEE Circuits and Systems International Conf., 2009. – P. 5–18.
108. Иванов Д.Е. Алгоритм построения инициализирующих последовательностей цифровых схем, основанный на стратегии симуляции отжига / Д.Е. Иванов, Р. Зуауи.- Искусственный интеллект, 2009.- №4.- С.415-424.
109. Preserving synchronizing sequences of sequential circuits after retiming / M.N. Mneimneh, K.A. Sakallah, J. Moondanos // Proceedings of the 2004 Asia and South Pacific Design Automation Conference.- 2004.- P.579 - 584.
110. Morkūnas K., Šeinauskas R. Circuit Reset Sequences based on Software Prototypes // Electronics and Electrical Engineering. – Kaunas: Technologija, 2010. – No. 7(103). – P. 71–76.
111. K. Morkunas, R. Seinauskas. Verification of Initialization Sequences for Sequential Circuits // Electronics and Electrical Engineering. – Kaunas: Technologija, 2011. – No. 6(112). – P. 61–64
112. Pixley C. Theory and implementation of sequential hardware equivalence / C. Pixley // IEEE Trans. Comput. Aided Design.- 1992.- V.11-12.- P.1469-1494.
113. Pomeranz I. On achieving complete testability of synchronous sequential circuits with synchronizing sequences / I. Pomeranz, S.M. Reddy // In Proceedings of the IEEE International Test Conference.- Los Alamitos, CF:IEEE Computer Society Press, 1994.- P.1007-1016.
114. Huang S.-Y. Formal Equivalence Checking and Design Debugging / S.-Y. Huang, K.-T. Cheng.- Boston: Kluwer Academic Publishers, 1998.- 229p.
115. Burch J. Symbolic model checking: 1020 states and beyond / J. Burch, E. Clarke, K. McMillan, D. Dill, L. Hwang // Proc. of IEEE Symp. Logic in Comp. Sci.- 1990.- P.1-33.
116. Huang S.-Y. Verifying sequential equivalence using ATPG Techniques / S.-Y. Huang, K.-T. Cheng, K.-C. Chen // ACM Transactions on Design Automation of Electronic Systems.- 2001.- V.6, №2.- P.244-275.
117. Corno F. Approximate Equivalence Verification for Protocol Interface Implementation via Genetic / F. Corno, M. Sonza Reorda, G. Squillero // Proceedings of the First European Workshops on Evolu-

tionary Image Analysis, Signal Processing and Telecommunications.- 1999.- P.182-192.
118. Corno F. VEGA: A Verification Tool Based on Genetic Algorithms / F. Corno, M. Sonza Reorda, G. Squillero // ICCD98, International Conference on Circuit Design, Austin, Texas (USA).- 1998.- P.321-326.
119. Huang S.Y. AQUILA: An Equivalence Checking System for Large Sequential Designs / S.-Y. Huang, K.-T. Cheng, K.-C. Chen etc // IEEE Transactions on Computers.- 2000.- V.49, №5.- P.443-464.
120. Corno F Approximate Equivalence Verification for Protocol Interface Implementation via Genetic Algorithms / F. Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, G. Squillero // Evolutionary Image Analysis, Signal Processing and Telecommunications, Lecture Notes in Computer Science, 1999.- Vol.1596.- 1999.- P.182-192.
121. Corno F. Evolutionary Simulation-Based Validation / F. Corno, M. Sonza Reorda, G. Squillero // International Journal on Artificial Intelligence Tools (IJAIT).- 2004.- Vol.14, 1-2, Dec.- P. 897 916.
122. Иванов Д.Е., Зуауи Р. Верификация эквивалентности цифровых схем с использованием стратегии симуляции отжига // «Науковий вісник Чернівецького університету». Випуск №479. Комп'ютерні системи та компоненті», 2009.- С.33-41
123. Иванов Д.Е. Применение стратегии симуляции отжига для верификации эквивалентности последовательностных схем / Д.Е. Иванов, Р. Зуауи // X Международная научно-техническая конференция «Искусственный интеллект. Интеллектуальные системы» (ИИ-2009).- Таганрог: Изд-во ТТИ ЮФУ, 2009.- С.30-32.
124. Sekanina L. Evolutionary design of digital circuits: where are current limits? / L. Sekanina // Proceedings of the first NASA/ESA conference on adaptive hardware and systems, Istanbul.- 2006.- P.171-178.
125. Goldberg D.E. Genetic Algorithm in Search, Optimization, and Machine Learning / D.E. Goldberg.- Boston, MA:Addison-Wesley Longman Publishing Co.- 1989.- 412p.
126. Holland J.P. Adaptaton in Natural and Artificial Systems: An Introductionary Analysis with Application to Biology, Control and Artificial Intelligence / J.P. Holland.- Ann Arbor MI:University of Michigan, 1992.- 228p.
127. Скобцов Ю.А. Основы эволюционных вычислений / Ю.А. Скобцов.- Донецк:ДонНТУ, 2008.- 326с.

128. Whitley D. A Genetic Algorithm Tutorial / Darrell Whitley // Statistics and Computting.- 1994.- №4.- P.65-85.
129. Гладков Л.А. Генетические алгоритмы / Л.А.Гладков, В.В. Курейчик, В.М. Курейчик.- М.:Физматлит.- 2006.- 319с.
130. Иванов Д.Е. Применение алгоритмов симуляции отжига в задачах идентификации цифровых схем / Д.Е. Иванов // Вісник Національного технічного університету "Харківський політехнічний інститут". Збірник наукових праць. Тематичний випуск: Інформатика і моделювання.– Харків: НТУ "ХПІ", 2011.– № 17.– С.60-69.
131. Грунский И.С. Синтез и идентификация автоматов / И.С. Грунский, В.А. Козловский.- Киев: Наукова думка, 2004.- 245с.
132. Luke S. Essentials of Metaheuristics / S. Luke // George Mason University Press.- 2009.- 239p.
133. Иванов Д.Е. Применение стратегии симуляции отжига для задачи построения инициализирующих последовательностей цифровых схем / Д.Е. Иванов, Р. Зуауи // «Вісник східноукраїнського національного університету ім.В.Даля», 2009.- №1(131), частина 2.- С.161-168.
134. Иванов Д.Е. Построение инициализирующих последовательностей синхронных цифровых схем с помощью генетических алгоритмов / Д.Е. Иванов, Ю.А. Скобцов, А.И. Эль-Хатиб.- Проблеми інформаційних технологій.-2007.-№1.- С.158-164.
135. Иванов Д.Е. Генетические алгоритмы построения инициализирующих последовательностей цифровых схем / Д.Е. Иванов, Ю.А. Скобцов, А.И. Эль-Хатиб // Тезисы Докладов Международной научной конференции «Компьютерные науки и информационные технологии».-Саратов:изд-во СГУ.-2007.- С.51-52.
136. Иванов Д.Е. Алгоритмы достижения состояний в цифровых устройствах и их применение в задачах диагностики / Д.Е. Иванов // Вісник Хмельницького національного університету. Технічні науки.- Хмельницький, 2012.- №3(189).- С.104-110.
137. Иванов Д.Е. Генетический подход проверки эквивалентности последовательностных схем / Д.Е. Иванов.- «Радіоелектроніка. Інформатика. Управління».- Запоріжжя, ЗНТУ.- 2009.- №1(20).- С.118-123.
138. Skobtsov Y.A. Genetic algorithms in test generation for digital circuits / Y.A. Skobtsov, D.E. Ivanov, V.Y. Skobtsov // Proceedings of

the 8th Biennial Baltic Electronics Conference.- Tallinn Technical University, 2002.- P.291-294.
139. Иванов Д.Е. Генетические алгоритмы построения идентифицирующих последовательностей для цифровых схем с памятью / Д.Е. Иванов.- Наукові праці Донецького національного технічного університету. Серія: “Обчислювальна техніка та автоматизація”. Випуск 14(129).-Донецьк: ДонНТУ.- 2008.- С.97-106.
140. Lee D.H. A new test generation method for sequential circuits / D.H. Lee, S.M. Reddy // Proc. Int. Conf. Computer-Aided design.- 1991.- P.446-449.
141. Ghosh A. Test generation for highly sequential circuits / A. Ghosh, S. Devadas, A.R. Newton // Proc Int. Conf Computer-Aided Design.- 1989.- P.362-365.
142. Wunderlich H.-J. Multiple distributions for biased random test patterns / H.-J. Wunderlich // IEEE Trans. Computer-Aided Design.- 1990.- Vol.9, №6.- P.584-593.
143. Іванов Д.Є. Еволюційні методи побудови перевіряючи тестів для цифрових схем / Д.Є. Іванов, Ю.О. Скобцов, В.Ю. Скобцов.- Вісник технологічного університету Поділля. Хмельницький, 2004.- С135-139.
144. Черемисинова Л.Д. Проверка схемной реализации частичных булевых функций / Л.Д. Черемисинова, Д.Я. Новиков // Вестник Томского государственного университета, Управление, вычислительная техника и информатика.- 2008.- № 4(5).- С.102-111.
145. Venkataraman S. Rapid diagnostic fault simulation of stuck-at faults in sequential circuits using compact lists / S. Venkataraman, I. Hartanto, W.K. Fuchs, E.M. Rudnick, S. Chakravarty, J.H. Patel // Proc. Design Automation Conf.- 1995.- P.133–138.
146. Cabodi G. Sequential circuit diagnosis based on formal verification techniques / G. Cabodi, P. Camurati, F. Corno, P. Prinetto, M. Sonza Reorda // Proc. of International Test Conf., 1992.- Pp.187-196.
147. Иванов Д.Е. Генетический алгоритм построения диагностических последовательностей цифровых устройств / Д.Е. Иванов.- Вісник східноукраїнського національного університету ім.В.Даля, 2010.- №10(152).- С.72-79.
148. Grüning T. DIATEST: a fast diagnostic test pattern generator for combinational circuits / T. Grüning, U. Mahslstedt, H. Koopmeiners // Proc. Int. Conf. on Computer Aided Design, 1991.- Pp.194-197.

149. Kubiak K. Exact Evaluation of Diagnostic Test Resolution / K. Kubiak, S. Parkes, W.K .Fuchs, R. Saleh // Proc. 29th Design Automation Conf., June1992.- Pp.347-352.
150. Rudnick, E.M. Diagnostic Fault Simulation of Sequential Circuits / E.M. Rudnick, W.L. Fuchs, J.H. Patel // Proc. of International Test Conference, 20-24 Sep. 1992.- Pp.178-186.
151. Lavo D. Making Cause-Effect Cost Effective: Low-resolution Fault Dictionaries / D. Lavo, T. Larrabee // in Proc. International Test Conf., 2001.- Pp.278-286.
152. Bodoh D. Diagnostic Fault Simulation for the Failure Analyst / D. Bodoh, A. Blakely, T. Garyet // Conference Proceedings from the 30th International Symposium for Testing and Failure Analysis (ASM International), Oct. 2004.- Pp.181-190.
153. Ivanov D.E. Parallel fault simulation on multi-core processors / D.E. Ivanov // «Радіоелектронні і комп'ютерні системи», 2009.- №6(40).- С.109-112.
154. Иванов Д.Е. Подходы к построению параллельных генетических алгоритмов идентификации цифровых схем для многоядерных систем / Д.Е. Иванов // Вісник Хмельницького національного університету. Технічні науки.- Хмельницький, 2011.- №1(172).- С.111-117. (0.8 д.а.)
155. Иванов Д.Е. Алгоритм параллельного вычисления оценок особей при верификации эквивалентности последовательностных схем / Д.Е. Иванов.- Проблемы информационных технологий, 2009.- №1(005).- С.105-112.
156. Иванов Д.Е. Верификация эквивалентности цифровых схем с использованием стратегии симуляции отжига / Д.Е. Иванов, Р. Зуауи.- «Науковий вісник Чернівецького університету». Випуск №479. Комп'ютерні системи та компоненті», 2009.- С.33-41.
157. Гергель, В.П. Основы параллельных вычислений для многопроцессорных вычислительных систем. Учебное пособие / Гергель В.П., Стронгин Р.Г.- Нижний Новгород: Изд-во ННГУ им. Н.И. Лобачевского.- 2003.- 184с.
158. Иванов Д.Е. Параллельные генетические алгоритмы построения идентифицирующих последовательностей для многоядерных систем / Д.Е. Иванов // Тезисы докладов 8й Российской конференции с международным участием «Новые информационные технологии в исследовании сложных структур».- Томск: Издательство научно-технической литературы, 2010.- С.11.

159. Иванов Д.Е. Масштабируемый параллельный генетический алгоритм построения идентифицирующих последовательностей для современных многоядерных вычислительных систем / Д.Е. Иванов // «Управляющие системы и машины».- 2011.- №1.- С.25-32. (1.0 д.а.)
160. Ivanov D. A scalable genetic algorithm for constructing of identifying sequences for VLSI circuits / D. Ivanov // Book of abstracts of 19th Conference on applied and industrial mathematics CAIM 2011.- Iasi University press, 2011.- P.68-69.
161. Копысов, С.П. Методы привязки параллельных процессов и потоков к многоядерным узлам вычислительных систем / С.П. Копысов, А.К. Новиков, Л.Е. Тонков, Д.Е. Береснев // Вестник Удмуртского университета.- 2010.- №1.- С.123-132.
162. Parker, S. A parallel algorithm for fault simulation based on PROOFS / S. Parker, P. Banerjee, J. Patel // Proc. IEEE Int. Conf. Computer Design.- 1995.- P.616-621.
163. Gillespie, M. Масштабирование программных архитектур для многоядерных вычислительных систем будущего [Электронный ресурс] / Matt Gillespie // Intel® Software Network.- Режим доступа: http://software.intel.com/ru-ru/articles/scaling-software-architectures-for-the-future-of-multi-core-computing/.- Загл. с экрана.- (1.06.2009).
164. Иванов Д.Е. Взаимодействие компонент в распределённых генетических алгоритмах генерации тестов / Д.Е. Иванов, П.А. Чебанов // Наукові праці Донецького національного технічного університету. Серія: “Обчислювальна техніка та автоматизація”. Випуск 16(147).- Донецьк:ДонНТУ, 2009.- С.121-127.
165. Іванов Д.Є. Генерація тестів та логічне моделювання цифрових схем в системі АСМІД-Е / Д.Є. Іванов, Ю.О. Скобцов, Іванов Д.Є., В.Ю. Скобцов.- Вісник Технологічного університету Поділля.– Хмельницький, 2002.- Т1.- С125-128.
166. Иванов Д.Е. Генетический подход к генерации проверяющих тестов в системе АСМИД-Е / Д.Е.Иванов, С.А.Закусило, В.Ю.Скобцов, Ю.А.Скобцов // Труды конференций "Искусственные интеллектуальные системы" и "Интеллектуальные САПР".-Москва:Физматлит.- 2002.- С.49-55.
167. Иванов Д.Е. Моделирование и построение тестов цифровых схем в системе АСМИД–Е / Д.Е.Иванов, Ю.А.Скобцов, В.Ю.Скобцов. // Компьютерные науки и информационные тех-

нологии, Тезисы докладов международной конференции.– Саратов:Изд–во Сарат.ун–та.- 2002.- С.63–64.
168. Скобцов Ю.А. Автоматизированная система моделирования и диагностики цифровых устройств / Ю.А. Скобцов, Г.Г. Пономаренко, А.П. Шатохин // УсиМ.- 1988.- №2.- С.11-16.
169. Скобцов Ю.А. Система логического моделирования и генерации тестов АСМИД-П / Ю.А. Скобцов Ю.А., В.Ю. Скобцов // УсиМ.- 1996.- №1/2.- С.39-45.
170. Иванов Д.Е. Система моделирования и генерации тестов цифровых схем / Д.Е. Иванов, Ю.А. Скобцов.- Наукові праці Донецького державного технічного університету, серія “Обчислювальна техніка та автоматизація», випуск 12.– Донецьк.– 1999.– С.143-150.
171. Иванов Д.Е. Автоматизированная система моделирования и генерации тестов АСМИД-Е. / Д.Е. Иванов, Ю.А. Скобцов.- Техническая диагностика и неразрушающий контроль.- №2, 2000.- С.54-59.
172. Иванов Д.Е. Автоматизированная система моделирования и идентификации цифровых устройств АСМИД- Evolution / Д.Е. Иванов.- Проблемы информационных технологий.- 2011.- №1 (009).- С.114-124.
173. Skobtsov Y.A. Distributed Fault Simulation and Genetic Test Generation of Digital Circuits / Y.A. Skobtsov, El-Khatib, D.E. Ivanov // Proceedings of IEEE East-West Design&Test Workshop (EWDT’06).- 2006.- P.89-94.
174. Mazumder P. Genetic algorithm for VLSI Design, Layout and Test Automation / P. Mazumder, E.M. Rudnick // N.-Y.: Prentice-Hall, Englewood Cliffts, 1998.- 338p.
175. Keim M. On the (non-)restability of synchronous sequential circuits / M. Keim, B. Becker, B. Stenner // Proc. of the IEEE VLSI Test Symposium.- 1996.- Pp.240-245.
176. Alba E. Parallelism and evolutionary algorithms / Alba E., Tomassini M. // IEEE Trans. on evolutionary computation.- 2002.- Vol.6, №5.- P.443-462.
177. Іванов Д.Є. Адаптивні механізми в генетичних алгоритмах / Д.Є. Іванов, Ю.О. Скобцов, С.А. Закусило.- Наукові праці Донецького Державного технічного університету, серія «Обчислювальна техніка та автоматизація», випуск 38.– Донецьк.– 2002.– С.104-109.

178. Иванов Д.Е. Применение адаптивных генетических алгоритмов для генерации тестов цифровых схем / Д.Е. Иванов, Ю.А. Скобцов, В.Ю. Скобцов, С.А. Закусило.- Наукові праці Донецького Національного Технічного Університету, серія «Обчислювальна техніка та автоматизація», випуск 47.– Донецьк.– 2002.– Вип.47.- С.249-255.
179. Иванов Д.Е. Исследование влияния параметров генетического алгоритма при генерации тестов для последовательностных схем / Д.Е. Иванов, Ю.А. Скобцов, П.А. Чебанов.- Вісник Донецького університету, Сер. А: Природничі науки, 2005.- Вип.2.- С.397-402.

ГЕНЕТИЧЕСКИЕ АЛГОРИТМЫ ПОСТРОЕНИЯ ВХОДНЫХ ИДЕНТИФИЦИРУЮЩИХ ПОСЛЕДОВАТЕЛЬНОСТЕЙ ЦИФРОВЫХ УСТРОЙСТВ

Автор:
Иванов Дмитрий Евгеньевич

Научное издание

В авторской редакции

Підписано до друку 11.11.2012.
Формат 60x84 1/16. Папір офсетний. Гарнітура Times.
Друк лазерний. Умов. друк. арк. 16,10. Обл. вид. арк. 17,02.
Тираж 300 прим. Вид. №707. Зам. №690.

Надруковано в типографії
ТОВ «Цифрова типографія»
83121, м. Донецьк, вул. Челюскінців, 291а
Тел.: (62) 388-07-31, 388-07-30