Санкт-Петербургский государственный университет
информационных технологий, механики и оптики
Кафедра «Компьютерные технологии»

Е. В. Смирнов, Е. В. Селифонов, А. В. Тихомиров

Преобразование исходного кода в автоматную программу

Проект создан в рамках
«Движения за открытую проектную документацию»
http://is.ifmo.ru

Санкт-Петербург
2009

Оглавление

1. Введение

Автоматное программирование, разрабатываемое на кафедре «Компьютерные технологии» СПбГУ ИТМО, является, по мнению авторов, перспективным направлением развития программирования. Оно позволяет упростить такие задачи, как верификация программ, построение документации и сопровождение программного продукта. Однако в настоящее время автоматный подход еще недостаточно распространен. Одной из причин этого является то, что большинство имеющихся систем разработано с применением классических подходов. Они наиболее привычны для программистов. Поэтому, когда встает вопрос, какой подход использовать при написании программы, об автоматном подходе часто даже не вспоминают.

Рассмотрев сложившуюся ситуацию, авторы пришли к выводу о необходимости создания некоторого автоматического средства для преобразования «обычной» программы (процедурной или объектно-ориентированной) в автоматную программу (программу с явным выделением состояний). При наличии такого средства, программист сможет писать программу привычным для него способом, получая в то же время все преимущества от использования автоматного подхода.

Идея генерации автоматного кода по коду программы была впервые затронута в работах [1, 2], но в ней сам процесс построения осуществлялся вручную. В работе [3] было предложено строить автоматную программу по предварительно структурированному определенным образом исходному коду. При этом были предложены схемы автоматов, соответствующие базовым структурам языка, с доказательствами их корректности. На основе этой работы было создано инструментальное средство *Vizi* [4], используемое для визуализации алгоритмов дискретной математики с применением автоматного подхода. Однако оно является специализированным и требует особого формата входного файла, в котором программа уже явно разбита на состояния. Таким образом, проблема создания автоматического инструмента построения автоматного кода по произвольной программе остается актуальной.

Для иллюстрации преимуществ, предоставляемых автоматным подходом, в дополнение к основному проекту авторами был разработан так называемый «*Backtracking Tracer*»: «обратный трассировщик», позволяющий выполнять программу по шагам не только вперед, но и назад.

Необходимость отладки программ нельзя недооценивать, так как на поиск ошибок и отладку уходит значительная часть времени разработки приложения. Современные отладочные средства предоставляют широкий выбор способов для облегчения этой работы: пошаговое выполнение, точки останова, дампы памяти и некоторые другие.

Рассмотрим типичный процесс поиска ошибки. Предположим, что в некоторый момент работы выданный программой результат отличается от

ожидаемого. Что в этом случае делает программист? Как правило, пытается восстановить ход программы «назад во времени», пытаясь определить, например, какой из выполненных фрагментов мог повлиять на выданный неверный результат. Определив такие фрагменты, программист обычно выставляет на них новые точки останова. Потом программа запускается заново. После останова программист анализирует состояние программы и пытается определить, верно ли оно, и если нет, то из каких других фрагментов могли попасть ошибочные данные. После этого процесс повторяется, часто многократно. При этом каждый раз программа запускается с самого начала, что, в случае больших программ, может быть крайне неэффективно.

В этой ситуации помочь в отладке могла бы возможность возвращаться назад, восстанавливая уже пройденные состояния, без перезапуска программы. При этом в программе с явно выделенными состояниями выполнение в обратном направлении может быть легко реализовано как переход к предыдущим состояниям. Для решения этой задачи авторами разрабатывается обратный трассировщик. При его использовании программист с целью найти источник ошибки может по шагам проверять состояния программы, произошедшие до ее проявления. Возможность комбинировать шаги в обе стороны и в любой последовательности позволит анализировать один и тот же фрагмент несколько раз без перезапуска программы. Таким образом, наличие такого трассировщика способно существенно упростить и сократить процесс отладки.

При этом отметим, что существуют обратные трассировщики, которые могут обрабатывать программы без явного выделения состояний. Однако они были разработаны давно (60-90-ые годы прошлого века) и своего развития не получили. Известен также коммерческий обратный трассировщик [6], но его реализация закрыта.

1.1. Постановка задачи

Пусть имеется исходный код программы, написанной на некотором языке с использованием процедурного либо объектно-ориентированного программирования. Требуется разработать преобразователь, конвертирующий традиционно написанную программу в автоматную программу, которая реализует тот же алгоритм работы. Под «тем же алгоритмом работы» будем подразумевать, что при одинаковых условиях и на одинаковых входных данных обе программы (традиционная и автоматная) дадут одинаковые результаты.

В данной работе на вход преобразователя подается программа, написанная на языке *C*, а на выход выдается соответствующая ей автоматная программа в виде набора файлов на языке *C++*. Отметим, что автоматная программа генерируется именно на языке *C++*, так как она использует классы, которых нет в *C*). Язык *C* был выбран, так как он, с одной стороны, широко распространен, а с другой — относительно прост (его грамматика намного

меньше и проще, чем у языка *C++* или *Java*).

Работа преобразователя разбивается на несколько независимых этапов (рис. 1).

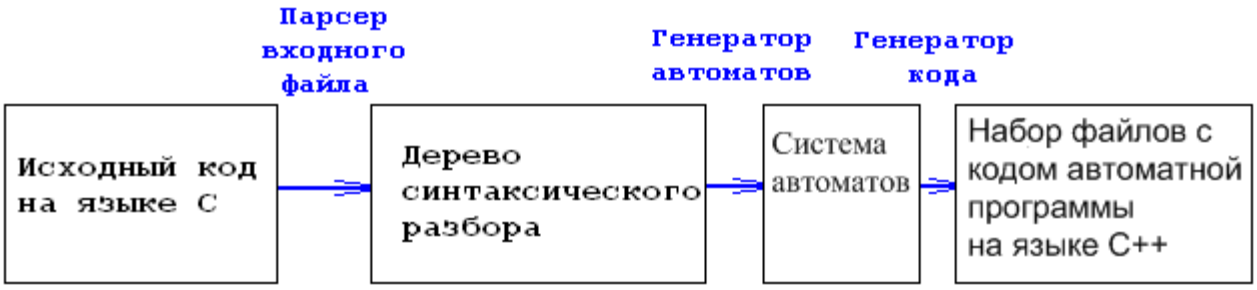

Рис. 1. Схема работы преобразователя

Разбиение на этапы выполнено для того, чтобы результаты одного этапа можно было использовать и для других целей. Например, построенный автомат можно затем вывести не в виде исходного кода на языке *C++*, а в формате описания автоматов какого-либо автоматического верификатора.

Дополнительной задачей работы было построение трассировщика, использующего сгенерированный автоматный код, с возможностью обратного хода. Этот трассировщик должен реализовывать набор основных функций:

- шаги вперед и назад;
- вывод необходимой отладочной информации (номер состояния, значения переменных);
- вывод информации о произошедших во время работы исключительных ситуациях и возможность продолжить работу после возникновения одной из них.

Трассировщик использует дополнительно сгенерированный *обратный автомат*. Обратный автомат — это автомат с множеством состояний, равным множеству прямого автомата, для которого после выполнения действия в состоянии будут восстановлены значения всех переменных предыдущего состояния.

Взаимодействие трассировщика с автоматной программой показано на рис. 2.

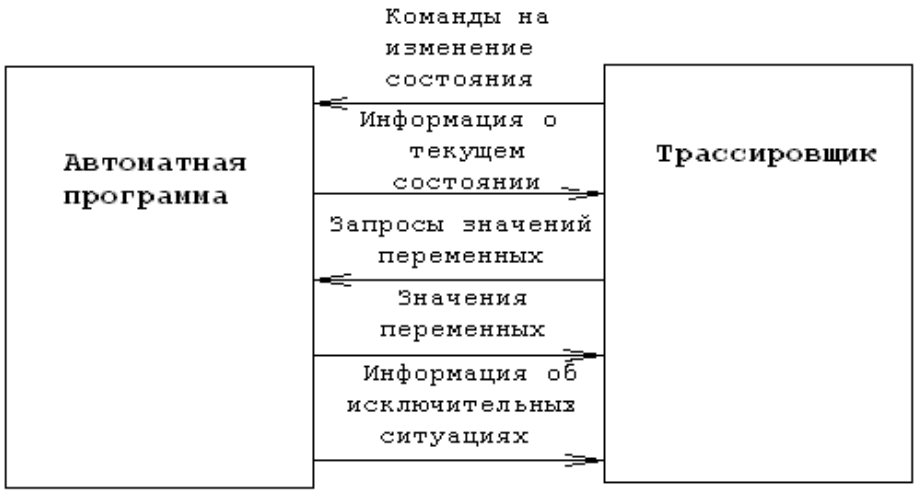

Рис. 2. Схема взаимодействия трассировщика и автоматной программы

2. Реализация

2.1. Генерация автоматного кода

Генерация кода автоматной программы разбита на несколько этапов:

- синтаксический разбор и построение дерева разбора;
- построение по дереву разбора системы автоматов: для каждой вершины дерева, соответствующей процедуре, строится два автомата: прямой и обратный;
- преобразование автоматов в программу на языке *C++* и вывод ее в выходные файлы.

Следует отметить, что несмотря на то, что генератор автоматного кода с виду похож на полноценный компилятор языка *C*, он таковым не является. В частности, он не производит проверку семантической модели языка и не обнаруживает ошибки с ней связанные, такие как:

- применение операторов к неподходящим типам (например, оператора «->» к переменным, не являющимся указателями на структуру, оператора «[]» к переменным, которые не являются массивами, и т.д.);
- несоответствия типов в операциях и вызовах функций;
- вызов объявленных, но не определенных функций.

Если такие ошибки были во входном файле, они будут перенесены и в выходной. Поэтому в дальнейшем будем исходить из предположения, что на вход был передан «правильный» *C*-файл: компилирующийся и линкующийся.

2.1.1. Разбор входного файла

Поступивший на вход файл с исходным кодом обрабатывается с помощью

лексического и синтаксического анализаторов, построенных с помощью программ *Flex* и *Bison* соответственно. Вначале входной поток символов разбивается на отдельные блоки (токены), соответствующие ключевым словам, операторам и идентификаторам языка. Поток токенов передается на вход синтаксического анализатора, который группирует их в соответствии с правилами грамматики языка *C* и строит по ним дерево синтаксического разбора. Это дерево состоит из элементов типа `SyntaxTreeNode`, определенного в файле `SyntaxTree.h`. Каждая вершина хранит следующую информацию:

- `int _nodeType` – тип вершины. Все возможные значения этого поля описаны как статические константные члены структуры `SyntaxTreeNode`. Тип вершины соответствует типу структуры языка, которая породила эту вершину: цикл, оператор ветвления, оператор присваивания, процедура, и особый тип, `SyntaxTreeNode::_SEMICOLON`, который служит для разделения элементов;
- `std::string _text` – текст, ассоциированный с вершиной. Конкретный смысл текста зависит от типа вершины: для условного оператора это будет условие, для оператора присваивания – само выражение, для процедуры – ее название. Данный атрибут (текст) присваивается на этапе синтаксически управляемой трансляции;
- `SyntaxTreeNode* _left, _right` – указатели на детей данной вершины в дереве;
- `std::string _nameOfVariableToSave` – имя переменной, которую необходимо будет сохранить в стек прежде, чем перейти к выполнению действия. Например, для оператора присваивания это будет имя переменной, которой присваивается некоторое значение.

Помимо построения дерева, на этом этапе производится сбор информации об объявленных переменных и функциях. Каждый раз, встретив последовательность токенов, соответствующих объявлению переменной, парсер добавляет ее к списку. Он хранится в виде `std::vector<Variable>`. Структура `Variable` описывает, в каком виде переменная хранится в памяти преобразователя. У нее есть два поля:

- `std::string _name` – строка с именем переменной;
- `CTypeID _type` – тип переменной. Класс `CtypeID` хранит информацию о типе и предоставляет методы для ее получения, в особенности при применении разного рода операторов. Например, для переменной, объявленной как `int a[6]`, применив соответствующий метод к имени “a”, получим тип `int*`, а для “`a[3]`” – `int`. Следует отметить, что такая идентификация типов предназначена не для проверок на их соответствие (что является довольно сложной задачей, не входящей

в поставленную), а для правильного определения размеров, необходимых для операций со стеком.

Помимо информации о переменных, собирается также информация об объявленных функциях. С каждой функцией ассоциирована ее сигнатура и список ее локальных переменных. Собранная на этом этапе информация используется для определения типа вызова: вызывать как вложеный автомат или как внешнюю функцию.

Построение дерева осуществляется вызовом процедуры `parse(const char* filename)`. Эта процедура производит необходимую инициализацию лексического и синтаксического анализаторов, а затем передает управление в процедуру `yyparse(void)`, которая создана по описанию грамматики *C* и выполняет сам разбор.

Детально на самом разборе останавливаться не будем, так как *LR*-разбор, синтаксически управляемая трансляция и управление атрибутами являются слишком обширной темой [5]. Вкратце используемый процесс можно описать как поиск подходящих грамматических правил для фрагментов файла с последующим применением связанных с ними действий: добавления новых вершин в дерево, добавления переменных и функций к спискам и т.д.

После завершения работы функция разбора возвращает указатель на вершину полученного дерева. В случае ошибки, бросается исключение `char*` со строкой, содержащей описание ошибки. После отображения этой строки работа программы прекращается.

2.1.2. Преобразование в систему автоматов

Полученное на первом этапе дерево синтаксического разбора передается для построения по нему системы автоматов, реализующих тот же алгоритм, что и исходная программа. Все описанные здесь методы находятся в файле `SyntaxToAutomata.cpp`.

Структуры, в виде которых автомат хранится в памяти, описаны в заголовочном файле `Automata.h`. Они предназначены для описания переходов и состояний, а также автомата в целом.

Структура перехода имеет следующий вид:

```
struct Transition
{
	// Номер состояния, в которое требуется перейти
	int _to;
	// Условие перехода
	string _condition;
	// Необходимо ли проверять условие при переходе
	bool _isConditional;
	// Действие, совершаемое при переходе
	string _action;
};
```

Состояние автомата описывается следующим образом:

```
struct AutomatonState
{
	// Номер состояния
	int _stateNumber;
	// Множество действий в состоянии
	vector<string> _actions;
	// Множество переходов из состояния
	vector<Transition> _transitions;
};
```

Следующая структура описывает автомат в целом:

```
struct Automaton
{
	// Имя автомата получается из имени процедуры
	string name_;
	// Номер конечного состояния
	int endState;
	// Множество состояний прямого
	AutomatonState* states;
	// Множество состояний обратного автомата
	AutomatonState* reverseStates;
	// Множество переменных, объявленных в функции,
	// по которой сгенерирован автомат
	vector<Variable> data;
};
```

Подробное описание фрагментов автоматов, соответствующих базовым структурам языка, с доказательством их корректности можно найти в работе [3]. Процедура построения автомата фактически состоит в последовательном применении структур, перечисленных ниже.

Вся работа по построению автомата происходит в процедуре `TreeTraverseWithReverse(const SyntaxTreeNode*)`. В этой процедуре происходит рекурсивный обход дерева и в зависимости от типа текущей вершины добавляет или не добавляет новые состояния, как в прямой, так и в обратный автоматы. Рассмотрим, что делает процедура для каждого типа вершины:

- вершина типа «Оператор». Если предыдущая вершина имела тот же тип, то необходимо добавить содержимое строки в список действий предыдущего состояния. Иначе следует создать новое состояние автомата, добавить в его список действий текст, а в список переходов – безусловный переход в следующее состояние. В обратном автомате необходимо провести аналогичные действия: создать или не создать состояние и добавить переход в предыдущее состояние.

  Если действие, выполняемое в вершине, изменяет значение какой-либо переменной, то ее имя будет записано в поле

`_nameOfVariableToSave`. В этом случае перед выполнением действия эту переменную требуется сохранить в стек, а при обратном проходе ее значение из стека восстановить;

- вершина типа «Оператор ветвления» (рис. 3). Здесь и далее на переходе сначала указано условие (если оно существует), а затем действие;

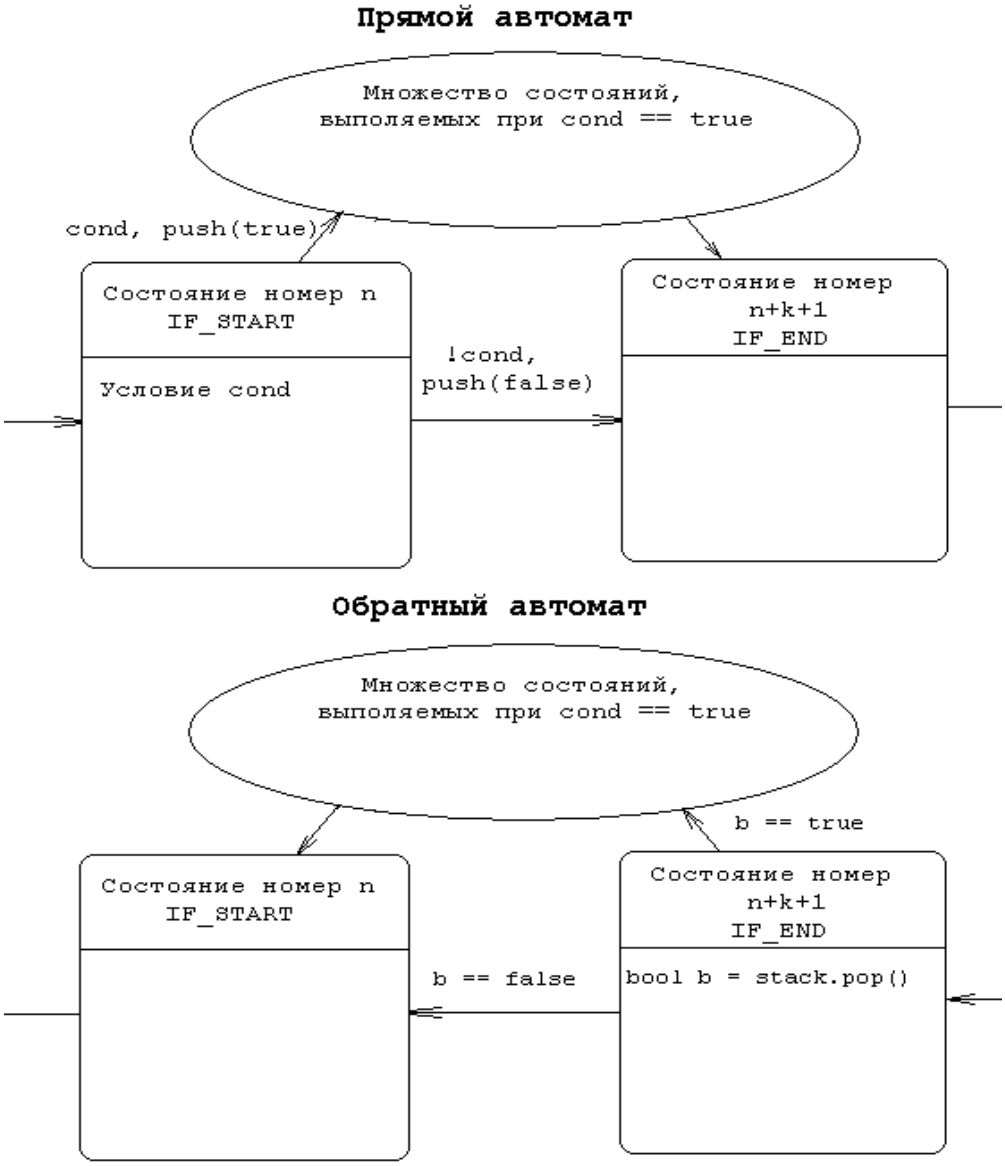

Рис. 3. Набор состояний, реализущих оператор ветвления

- вершина типа «Цикл с предусловием» (рис. 4);

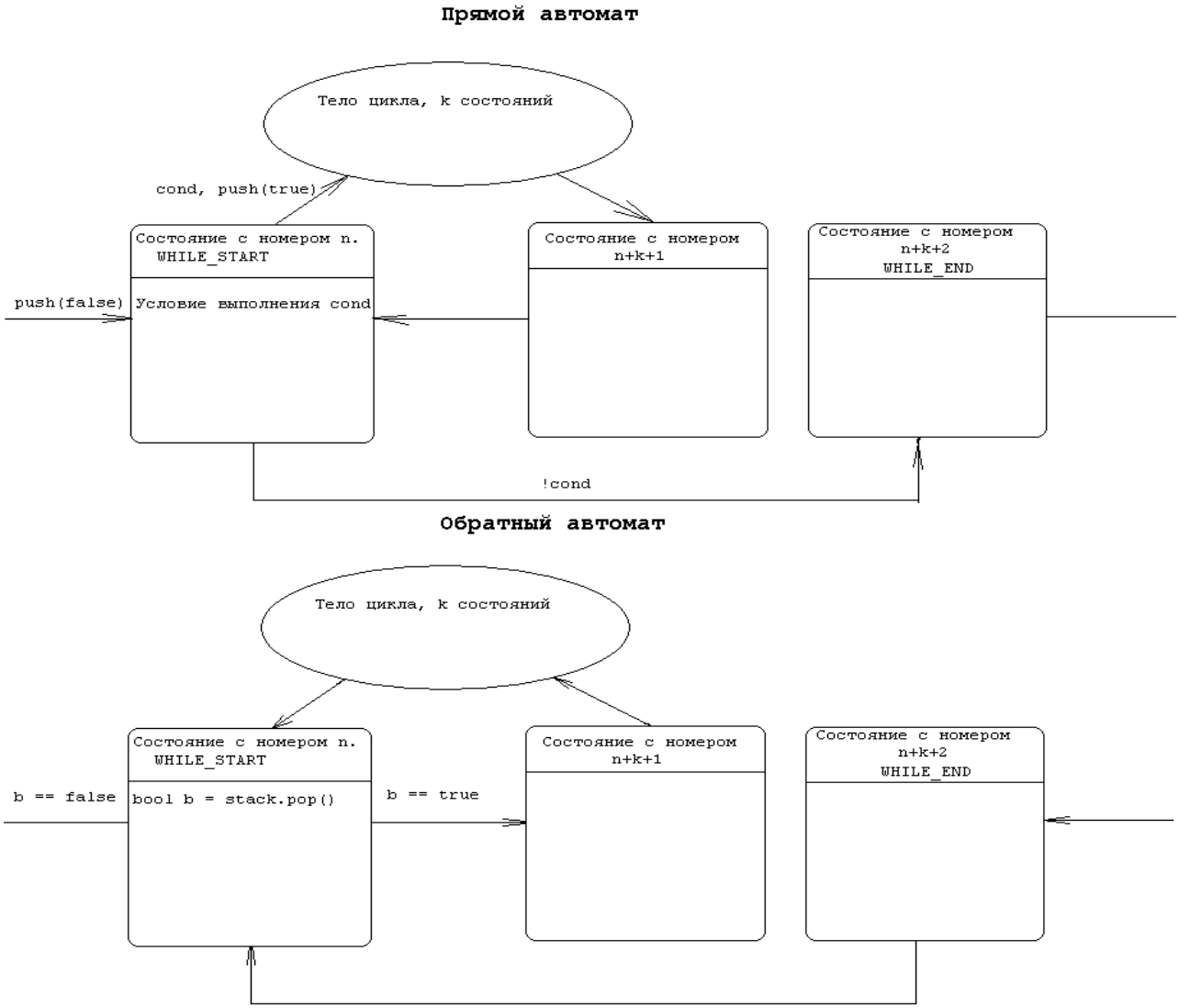

Рис. 4. Набор состояний, реализующих оператор цикла с предусловием

- вершина типа «Разделитель». В данном случае никакие состояния не порождаются. Функция обхода дерева рекурсивно запускается от левого и правого ребенка вершины;
- вершина типа «Возврат из функции». Создается новое состояние, из которого должен осуществляться переход в конечное состояние автомата. На данном этапе общее число состояний автомата неизвестно, номер добавленного состояния заносится в специальный список. После завершения обработки добавляется безусловный переход в конечное состояние во все состояния этого списка. В качестве действия в состоянии происходит сохранение в стек его номера для того, чтобы при обратном проходе в конечном состоянии его извлечь и перейти к требуемому состоянию, которое при прямом проходе и вызвало возврат из функции;
- вершина типа «Вызов функции с неизвестным кодом». Данный тип показывает, что вызывается функция, для которой недоступен исходный код. Следовательно, по ней не удается построить автомат и не имеется

никаких сведений о том, как функция может повлиять на состояние программы в целом (такая функция может просто вернуть значение, может изменить значения переданных по ссылке параметров, может произвольно поменять значение глобальных переменных). Поэтому перед вызовом этой функции происходит сохранение в стек всех доступных в ее области видимости переменных: локальных для данного автомата и глобальных для всей программы. При обратном проходе эти значения восстанавливаются из стека;

- вершина типа «Вызов функции с известным кодом». Для вызываемой функции на этапе разбора было найдено тело, по которому автомат либо уже был создан, либо будет создан далее. Этот автомат будет заниматься корректным обращением выполнения функции, поэтому перед ее вызовом не требуется сохранять значения переменных (рис. 5).

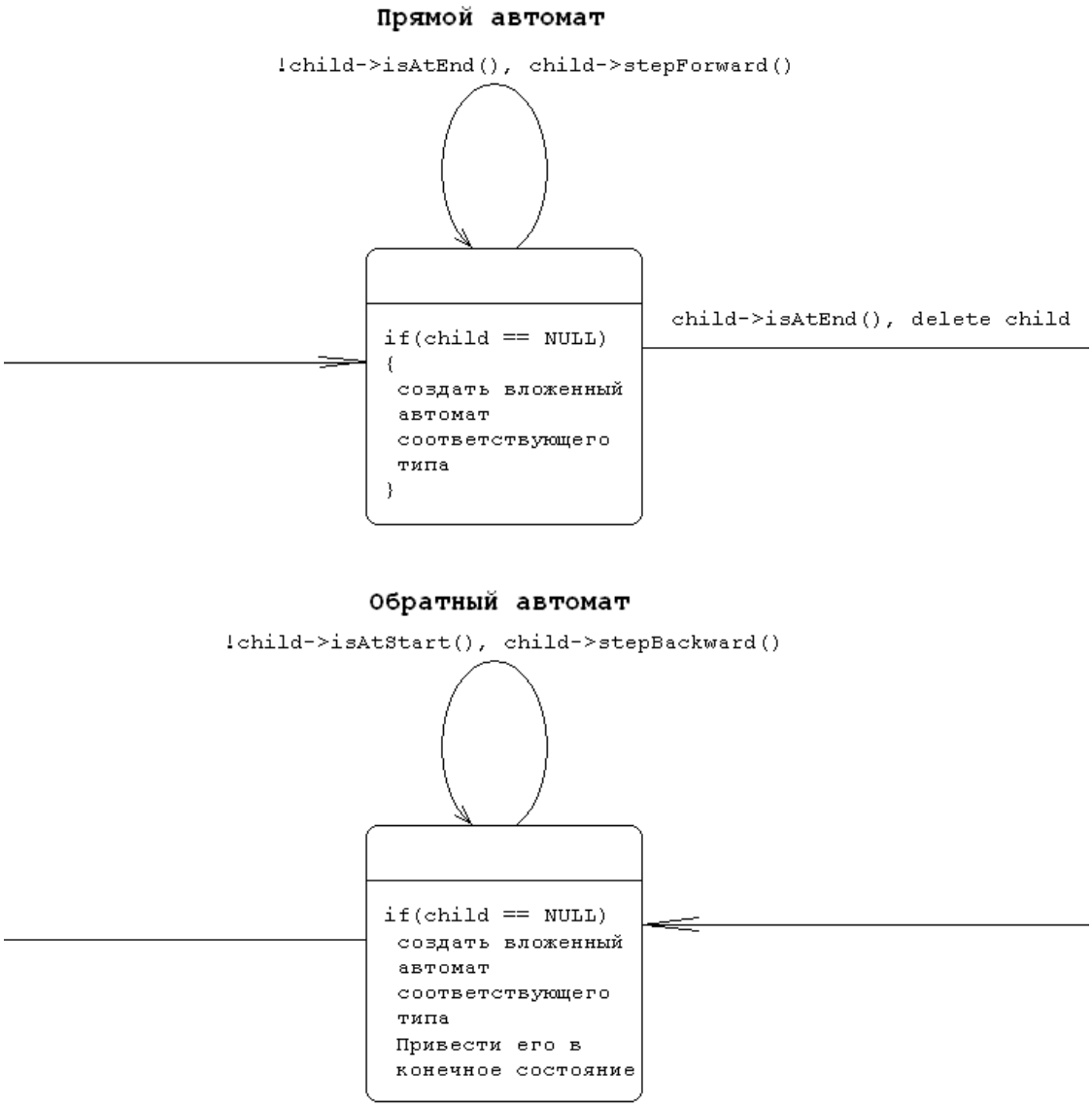

Рис. 5. Набор состояний, реализующих вызов вложенного автомата

Любой другой тип вершины сигнализирует об ошибке и вызовет исключение. Результатом работы процедуры будет указатель на

сгенерированнный автомат, или исключение, если генерацию не удалось произвести.

2.1.3. Генерация кода

После получения автомата наступает финальная стадия обработки: генерация выходных файлов.

Любой автомат на выходе состоит из двух частей:

- автомата с определенными методами для шагов вперед/назад;
- модели данных.

Модель данных – это структура, в которой содержатся все объявленные в исходной для рассматриваемого автомата процедуре переменные. Для каждой модели данных создается заголовочный файл с названием вида `имя_автоматаDataModel.h`, который подключается к исходному файлу `result.cpp` Помимо определений переменных он может содержать вспомогательные методы для их отображения и изменения извне.

Код самих автоматов располагается в одном `.cpp` файле. Каждый автомат представлен классом, сгенерированным на основе следующего шаблона:

```
class $NAME$Automaton: public AutomatonBase
{
	$NAME$DataModel *data;
public:
	$NAME$Automaton();
	~$NAME$Automaton();

	virtual void stepForward(int stepNumber = 1);
	virtual void stepBackward(int stepNumber = 1);

    std::string getVariable(const std::string &varName)
    {
		return data->getVariable(varName);
    }
};

$NAME$Automaton::~$NAME$Automaton()
{
	delete data;
}

$CONSTRUCTOR$

void $NAME$Automaton::stepForward(int stepNumber)
{
	$FORWARD$
}
```

```
void $NAMEA$Automaton::stepBackward(int stepNumber)
{
	$BACKWARD$
}
```

Этот шаблон копируется в выходной файл с заменой меток $NAME$, $CONSTRUCTOR$, $FORWARD$, $BACKWARD$ на имя автомата, код конструктора, код прямого и обратного автоматов соответственно. Базовый абстрактный класс автомата, от которого наследуются все конкретные автоматы, имеет следующий вид:

```
class AutomatonBase
{
protected:
	// Номер конечного состояния
	int endState_;

	// Указатель на вложенный автомат
	Automaton* child;

	// Номер текущего состояния
	int currentState_;

	// Указатель на стек
	CSStack *stack_;

public:
	explicit AutomatonBase(int endState)
	~AutomatonBase()

	// Функция, обеспечивающая выполнение
	// заданного числа шагов вперед
	virtual void stepForward(int stepNumber = 1) = 0;

	// Переводит автомат в конечное состояние
	void toEnd()

	// Находится ли автомат в начальном состоянии?
	bool isAtStart() const

	// Находится ли автомат в конечном состоянии?
	bool isAtEnd() const

	// Получить номер текущего состояния
	int getStep() const

	// Сменить номер текущего состояния
	void setStep(int newValue)

	// Получить номер конечного состояния
	int getMaxState() const
```

```
    // Выполнить заданное число шагов назад
    virtual void stepBackward(int stepNumber = 1)

    // Переводит автомат в начальное состояние
    void toStart()
};
```

Этот класс описан в заголовочном файле `ag_base.h`, подключаемом к выходному файлу и содержащем необходимые служебные классы и функции.

Предположим, что во входном файле были определены две функции: `f1()` и `f2()`. В результате на выходе получается следующая структура файлов (рис. 6).

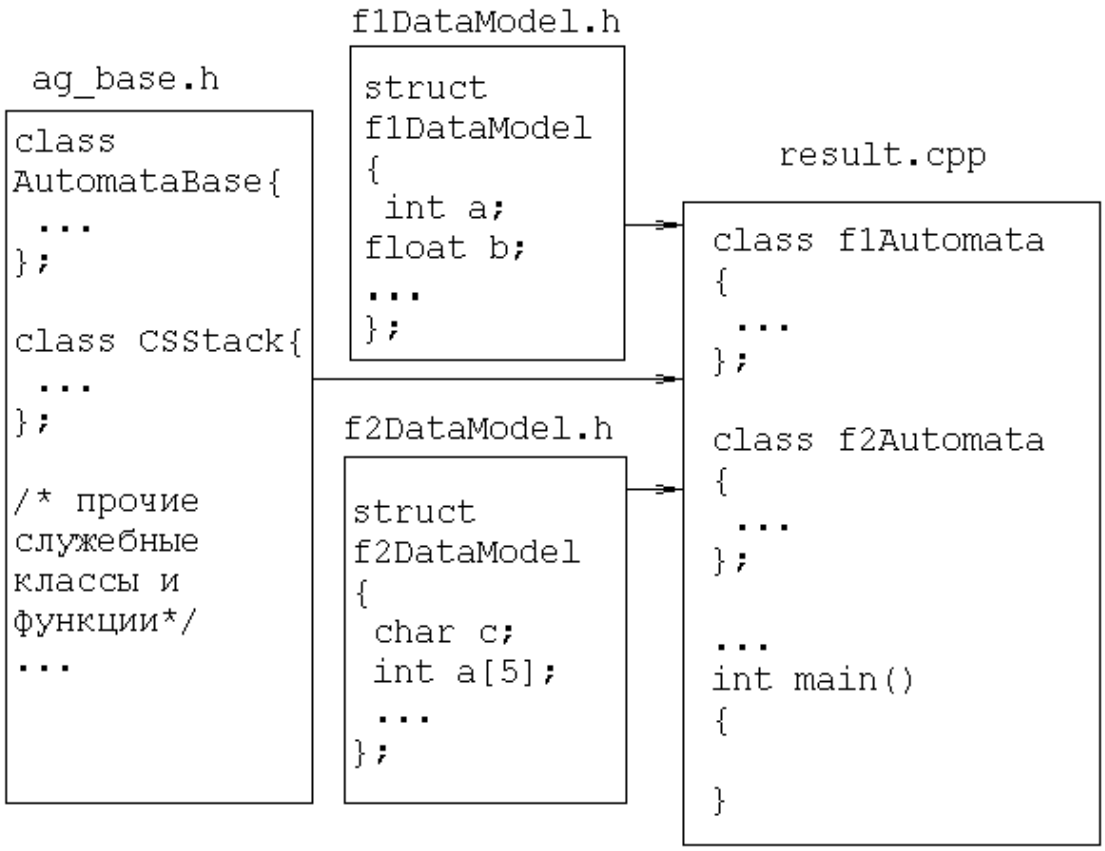

Рис. 6. Структура выходных файлов

Если при генерации автоматного кода была выбрана генерация со встроенным трассировщиком, то к результату будет добавлена библиотека `tracer.dll`, содержащая его код и предоставляющая пользовательский интерфейс.

2.2.1. Принцип работы

Состояние программы в любой момент времени можно характеризовать двумя параметрами: значением всех ее переменных и текущим местом выполнения. Для построения трассировщика, способного делать шаги назад, необходимо найти способ восстанавливать предыдущие состояния. Для этого

требуется в каждый момент уметь восстанавливать как значения переменных, так и положение исполняемой команды.

Сама идея обратного выполнения программы не является новой. Она впервые была реализована в 1960 году в отладчике *EXDAMS* для языка программирования *Fortran*, однако широкого распространения не получила. На эту тему был написан ряд работ (например, работа [6]) и была предпринята попытка интеграции обратного трассировщика в набор средств разработки *GCC* (*GNU Compiler Collection*), однако она провалилась. Авторам настоящей работы удалось найти только один современный отладчик, предоставляющий функцию обратного выполнения [7], однако он является коммерческим продуктом с закрытыми кодами и документацией, так что невозможно определить, какой подход был использован при его построении.

Существуют два подхода к реализации восстановления состояний программы, выделенные в работе [6]:

- с полным сохранением истории выполнения. С каждой операцией, изменяющей значения переменных, ассоциируется изменяемое множество. При прямом проходе перед выполнением каждой такой операции элементы ее изменяемого множества кладутся в стек, откуда извлекаются при обратном проходе;
- структурный. В этом случае изменяемое множество ассоциируется с управляющими операторами (ветвления, цикла и т.д.). Как и в первом случае, при прямом проходе значения переменных из изменяемого множества сохраняются, а при обратном проходе восстанавливаются. При шаге назад выполняется переход не к предыдущему оператору, а к оператору, предшествовавшему текущему блоку

Различие этих двух подходов можно пояснить на простом примере. Рассмотрим фрагмент программы:

```
оператор 1;
while(условие)
{
    оператор 2;
    оператор 3;
}
```

Предположим, на некоторой итерации цикла выполнен оператор 3, а после этого требуется сделать шаг назад. При использовании подхода с полным сохранением истории, был бы выполнен переход к состоянию, предшествовавшему выполнению оператора 3 на данной итерации (после выполнения оператора 2). Продолжая делать шаги назад, обращение тела цикла было бы выполнено столько раз, сколько раз оно было выполнено при прямом проходе. После этого был бы сделан переход к обращению оператора 1 и всего,

что ему предшествовало. При использовании структурного подхода, сразу было бы восстановлено состояние, соответствовавшее моменту входа в цикл (после выполнения оператора 1). Если бы затем понадобилось восстановить состояние на какой-то конкретной итерации цикла, то пришлось бы выполнять тело требуемое число раз для того, чтобы достичь заданного положения.

Предлагаемый в данной работе трассировщик использует подход с полным сохранением истории. Сохранение истории выполнения программы и последующее выполнение тех же инструкций в обратном порядке обеспечивается построением обратного автомата.

Рассмотрим типичную обработку состояния. Пусть это будет состояние операторного типа: выполняющее некоторые вычисления и не влияющее на порядок выполнения других команд. Фрагмент кода, по которому было создано состояние:

```
a = 5;
b += a;
```

Пусть это состояние получило номер `n`. В прямом автомате оно будет выглядеть следующим образом:

```
switch(currentState){
    ...
    case n-1:                    // Переход в состояние
        currentState = n;
    break;
    case n:                      // Переход из состояния
        currentState = n+1;
    break;
    ...
}
switch(currentState){
    ...
    case n:                      // Действия в состоянии
        stack.push(data->a);
        data->a = 5;
        stack.push(data->b);
        data->b += a;
    break;
    ...
}
```

А такой вид этот фрагмент будет иметь в обратном автомате:

```
switch(currentState)                    // Откат изменений,
{                                       // сделанных при прямом проходе
     ...
     case n:
          stack.pop(&data->b);
          stack.pop(&data->a);
     break;
     ...
}
switch(currentState)
{
     ...
     case n:
          currentState = n-1;     // Переход в предыдущее состояние
     break;
     case n+1:                   // Переход из следующего состояния
          currentState = n;
     break;
     ...
}
```

Обработка состояния разбивается на два этапа, каждый из которых реализован своим оператором `switch`, переходы рассматриваются отдельно от действий в состоянии. Рассмотрим, что будет делать автомат, дойдя до этого фрагмента.

Пусть текущий номер состояния `n-1`:

- пройдя первый оператор `switch`, автомат изменит свой номер состояния на `n`;
- войдя во второй `switch`, автомат выполнит все действия, записанные в состоянии `n`.

Если теперь станет необходимо остановить выполнение и сделать шаг назад, то будет выполнена следующая последовательность действий:

1. После полной обработки состояния `n`, значению `currentState` присвоится `n`. Поэтому после первого оператора `switch` обратного автомата будут выполнены команды на восстановление данных из стека.
2. Выполнив второй оператор `switch`, автомат совершит переход в

предыдущее состояние (в данном случае это состояние будет иметь номер `n-1`).

Если после этого остановить обратное выполнение, то автомат будет находиться в состоянии `n-1`, полностью отменив все изменения, сделанные в состоянии `n`.

Для операторов ветвления при проверке условия в стек сохраняется его значение (`true` или `false`) для того, чтобы при обратном проходе можно было определить, в какую ветвь идти.

Для оператора цикла в стек один раз сохраняется значение `false`, а затем при каждом выполнении условия – значение `true`. Обращать тело цикла потребуется столько раз, сколько раз будет снято значение `true`, до первого `false`.

Таким образом, с помощью стека для переменных и прямого/обратного автоматов возможно в любой момент вернуть программу в состояние, соответствовавшее прошедшему моменту времени, а затем снова продолжить ее выполнение с тем же результатом (при условии совпадения внешних условий). Исключением являются моменты, когда выполняется функция, для которой недоступен исходный код. Для такой функции невозможно сгенерировать автомат, поэтому обрабатываться она будет «атомарно», ее вызов и вся выполненная ей работа будут находиться в одном состоянии.

Рассмотрим в качестве примера вызов функции `f()`, для которой недоступен исходный код. Данный вызов будет иметь следующий вид (оператор `switch`, осуществляющий переход, опущен):

```
switch (currentState){
    ...
    case n:
    stack->push(data);        // Сохранение всей модели данных
    f();
    break;
    ...
}
```

и обратный автомат будет иметь вид:

```
switch (currentState){
    ...
    case n:
    stack->pop(data);         // Восстановление всей модели данных
    break;
    ...
}
```

2.2.2. Предоставляемые функции

Помимо собственно восстановления хода выполнения программы, от трассировщика требуется некоторое число функций для облегчения отладки программы. Такими функциями являются возможность просматривать значения переменных и указание текущего шага выполнения.

Получение значения переменной реализовано в структуре модели данных того автомата, в котором эта переменная была объявлена. Функция возвращает по имени строку, состоящую из типа и текущего значения, и выглядит примерно так:

```
std::string getVariable(const std::string &variable_name)
     {
          if(variable_name == "a")
               return "Type: " + string("int[6]").append("; Value:
").append(toString(a));
          if(variable_name == "counter")
               return "Type: " + string("int").append("; Value:
").append(toString(counter));
          if(variable_name == "max")
               return "Type: " + string("int").append("; Value:
").append(toString(max));
          return " Unrecognized variable name.";
     }
```

Эта функция создается на последнем этапе генерации кода, используя информацию о локальных переменных автомата, собранную на этапе разбора исходного файла. Функции `toString()` определены от различных базовых типов и предназначены для получения строковых представлений значений.

Информация о номере текущего шага также предоставляется автоматом.

В дополнение к этим функциям трассировщик позволяет выполнять одиночные шаги вперед и назад, а также запускать выполнение программы до конца (или, в обратную сторону, до начала). Если в процессе работы произойдет исключение, оно будет поймано, выполнение отлаживаемой программы будет остановлено, и управление вернется пользователю. Для обнаружения исключений используется механизм *SEH* (*Structured Exception Handling*), который позволяет обрабатывать в том числе и аппаратные исключения (например, деление на ноль).

3. Ограничения на входную программу

Ограничения на программу можно условно разбить на две категории: методические и технические.

3.1. Методические ограничения

Методические ограничения связаны с выбранным алгоритмом обращения

выполнения программы. Для их определения сперва более подробно рассмотрим, что же входит в понятие «состояние программы». Уже упоминавшиеся значения всех переменных и программного счетчика можно назвать «внутренним» состоянием. Программа имеет полный контроль над своим внутренним состоянием. Но помимо этого есть еще и «внешнее» состояние: все то, что влияет на ход выполнения программы, но непосредственно ей не контролируется. Сюда входит как получение данных извне (от пользователя, по сети, из другого потока), так и *контекст выполнения*: ресурсы, дескрипторы, открытые файлы, загруженные библиотеки.

Приведем простой пример, в котором влияние внешнего состояния приводит к неправильному обращению хода выполнения:

```
HWND hWnd = CreateWindow(…); // Создание окна
… // Некоторые действия с окном
DestroyWindow(hWnd); // Удаление окна
```

При попытке обратной трассировки после обращения действий по удалению окна будет восстановлено значение переменной `hWnd`. Однако восстановления значения дескриптора недостаточно для восстановления объекта. Данный дескриптор останется *невалидным (invalid)*, не ссылающимся на существующий объект. Любая операция над ним приведет к ошибке.

Возможным решением будет назначение пользователем пар команд «создание ресурса – удаление ресурса». Тогда при обнаружении одной из таких команд в обратный автомат будет добавлена парная ей. К сожалению, такой подход неприменим в случае объектов с нетривиальной инициализацией и удалением.

Более простым выглядит поиск таких проблемных мест (последовательностей состояний между созданием и удалением) и предупреждение пользователя при попытке использования обратной трассировки.

3.2. Технические ограничения

Технические ограничения обусловлены степенью реализации возможностей языка в конвертере. Текущей версией проекта является 0.3. Все данные приведены именно для этой версии.

Схема программы

Входная программа может состоять из произвольного числа процедур. Процедуры могут иметь параметры базовых типов. Возвращаемые значения не поддерживаются. Не поддерживаются также глобальные переменные и директивы препроцессора. При наличии не поддерживаемых компонент генерируется ошибка парсера. Одна из процедур обязательно должна

называться `main`, именно ее автомат будет запускаться самым первым.

Операторы

Поддерживаются операторы `if` с `else`, операторы цикла с предусловием `while`, все возможные операторы присваивания, операторы префиксного/постфиксного инкремента и декремента, операторы доступа к элементам массива, оператор `return`. Не поддерживается использование унарных операторов и вызовов функций как часть сложного выражения.

Пример недопустимого в данный момент кода:

```
some_var = another_var*some_func();
some_var = another_var – third_var++;
```

Указатели и работа с памятью

Не поддерживаются. Грамматические правила для них присутствуют, поэтому никаких ошибок обработки сгенерировано не будет, но корректность поведения автоматов при операциях с памятью не гарантируется.

Объявления

Переменные могут быть объявлены только с базовым типом, без модификаторов (таких как `static` или `const`). Переменные могут быть инициализированы при объявлении. Никаких проверок совпадения типов не производится.

Функции могут быть определены с любым базовым типом в качестве возвращаемого значения и с произвольным числом параметров. Не поддерживается объявление функций без тела.

Прочее

Инициализация массивов через перечисление списка элементов (например, `int a[6] = {0,1,2,3,4,5};`) поддерживается, но пока недостаточно протестирована, предпочтительнее инициализация каждого элемента в отдельности.

Не поддерживается использование в одной процедуре нескольких переменных с одинаковыми именами, но разными типами. Например, такой код не будет правильно обработан:

```
void f()
{
    int a;
    ...
    if(some_condition)
    {
        float a;
```

```
        ...
    }
}
```

При этом ошибки обработки сгенерировано не будет, но, скорее всего, будет получена ошибка времени выполнения или времени компиляции полученной программы.

4. Пример использования трассировщика

Рассмотрим в качестве примера программу (сортировка вставками):

```
int source[8];

int sorted[8];

void insert(int n)
{
    int i = 0;
    while (i < 8)
    {
        if (n < sorted[i])
        {
            int k = 6;
            while (k >= i)
            {
                sorted[k+1] = sorted[k];
                k--;
            }
            sorted[i] = n;
            i = 100;
        }
        i++;
    }
}

void main()
{
    int i = 7;

    source[0] = 1;
    source[1] = 2;
    source[2] = 5;
    source[3] = 4;
    source[4] = 3;
    source[5] = 7;
    source[6] = 9;
    source[7] = 2;
```

```
        while (i >= 0)
        {
              sorted[i] = 1337;
              i--;
        }
        i = 7;
        while (i >= 0)
        {
              insert(source[i]);
              i--;
        }
}
```

На основе данной программы будет построена система, состоящая из двух автоматов. Их схемы (сгенерированные в процессе построения) приведены на рис. 7, 8.

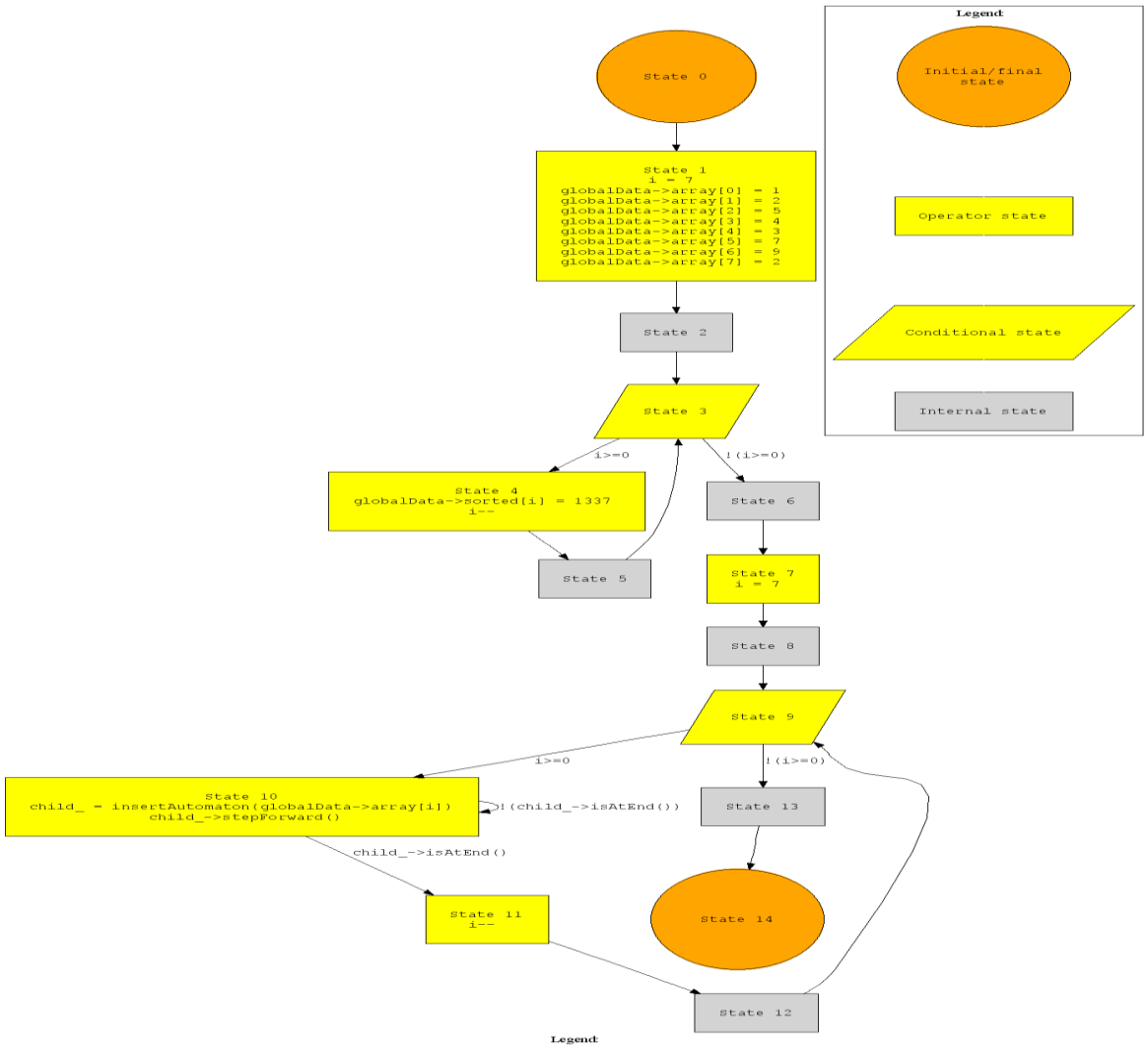

Рис. 7. Схема автомата для процедуры `main`

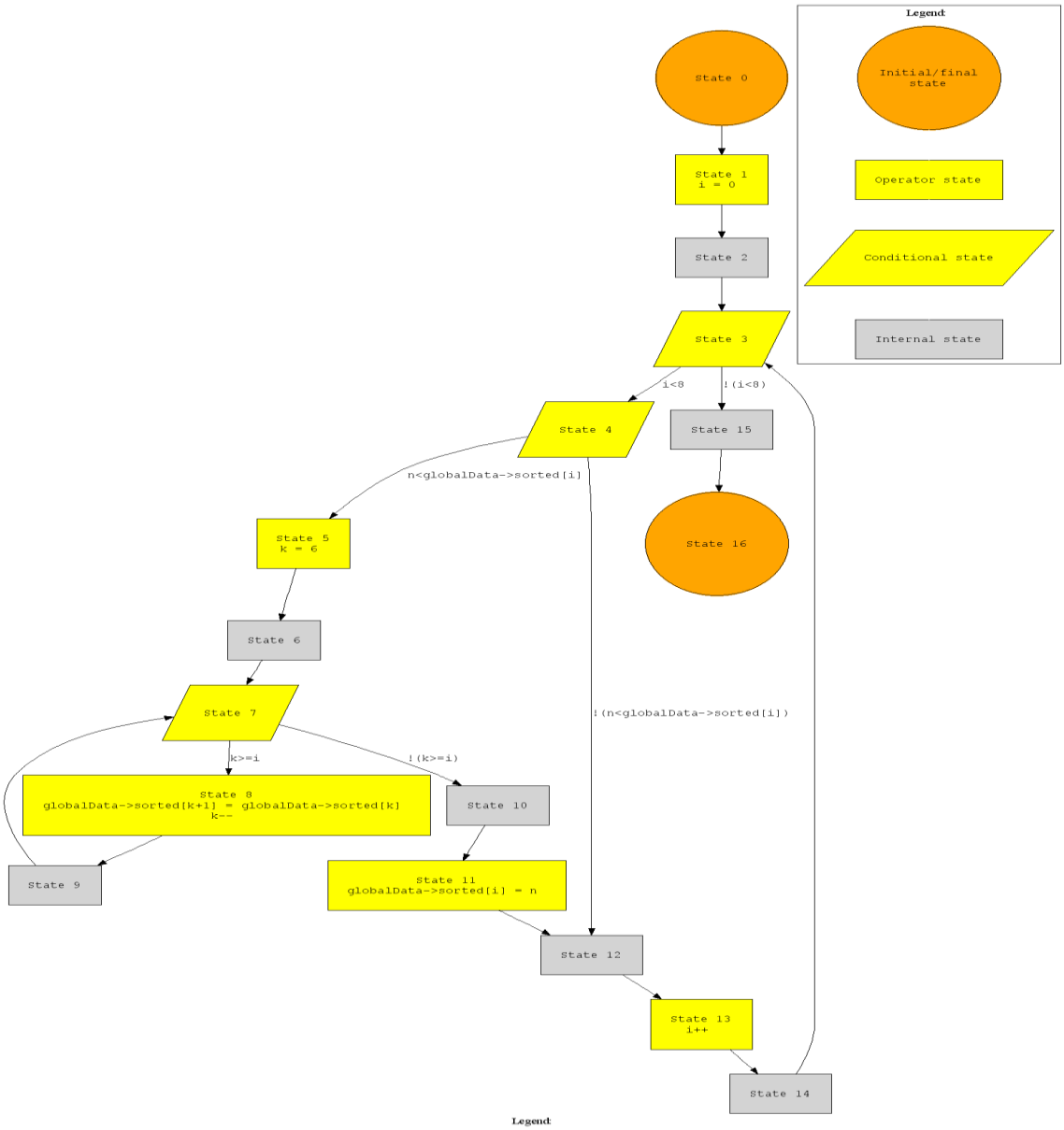

Рис. 8. Схема автомата для процедуры `insert`

Интерфейс трассировщика с загруженным исходным кодом примера представлен на рис. 9.

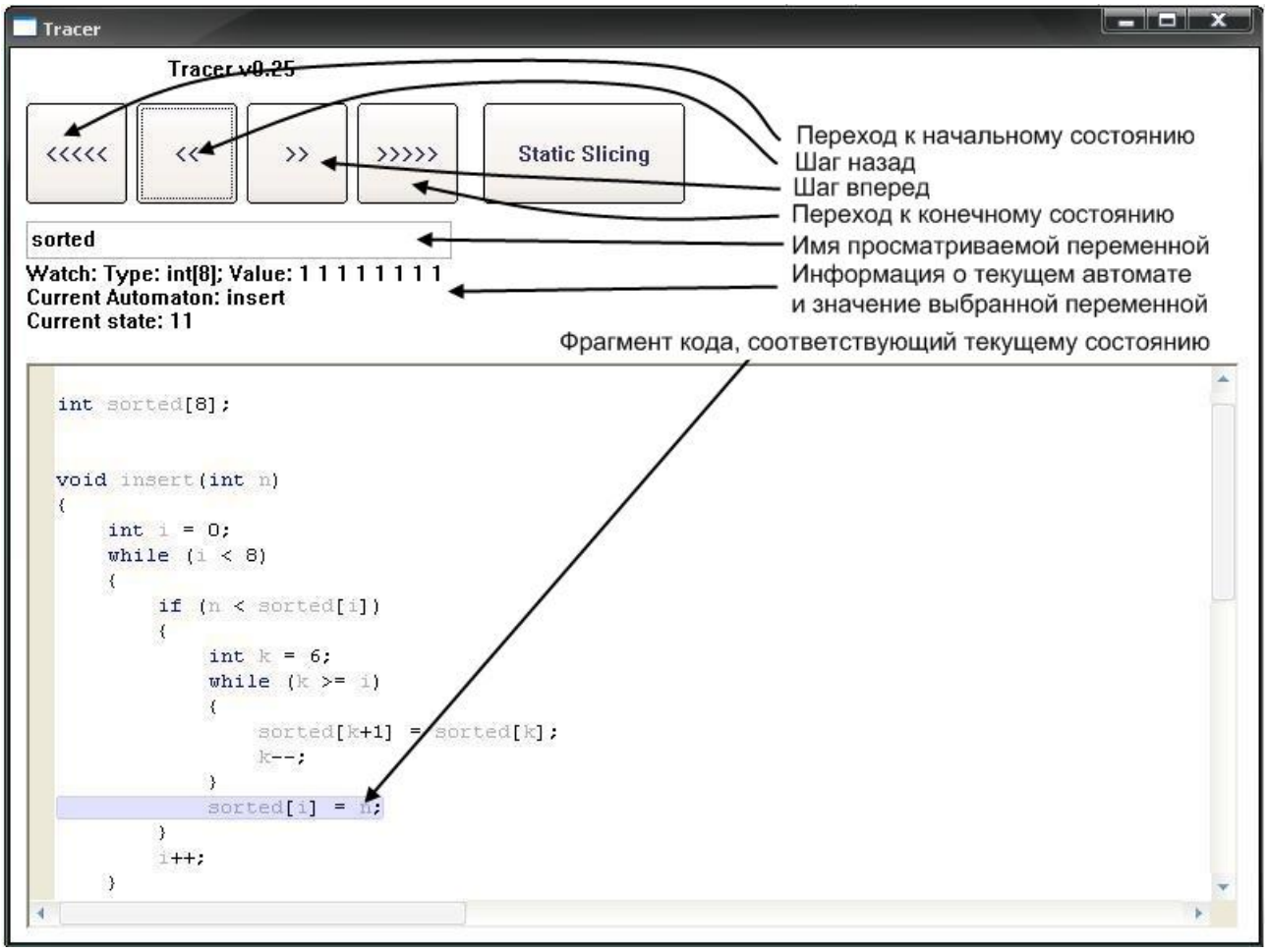

Рис. 9. Интерфейс трассировщика

Перспективы

Преобразование программы в систему взаимодействующих автоматов позволяет упростить реализацию еще одной технологии отладки программ: *слайсинга (slicing)*. *Слайсом* для данной переменной в данном месте программы является множество всех операторов, которые прямо или косвенно могли повлиять на значение этой переменной к моменту выполнения указанного места. При этом под словом «могли» подразумевается «могли повлиять хотя бы при каком-то наборе входных данных». Такое множество также называется *статическим слайсом,* так как его вычисление не требует запуска программы. Существует и *динамический слайс* — множество операторов, которые прямо или косвенно могли повлиять на значение переменной при **указанном наборе входных данных.** Построение динамического слайса уже требует запуска программы, но при этом дает гораздо более точную информацию.

Авторами работы [8] для построения слайсов предлагается использовать *Program Dependece Graph,* который, в свою очередь, состоит из двух частей: графа зависимости управления и графа зависимости данных. При этом при наличии графа переходов автомата, реализующего выбранную программу, оба эти графа можно легко построить: множества вершин у них будут совпадать, а переходы можно построить на основе хранящейся в состояниях автомата

информации.

Использование слайсинга позволяет заметно облегчить труд по поиску мест программы, которые требуется детально изучить с целью поиска ошибки. Обнаружив выданное программой неправильное значение переменной, достаточно будет получить автоматически построенный слайс этой переменной, для того чтобы найти все потенциальные источники ошибок.

Авторы надеются в следующих версиях трассировщика реализовать статический, а затем и динамический слайсинг. Это позволит заметно упростить процесс отладки программ.

Заключение

Как было показано на примере, построение автоматного кода по исходному коду программы является полезным и может быть использовано, в частности, для упрощения отладки приложения. Построенный по программе автомат фактически является ее **моделью,** причем с конечным числом управляющих состояний. К такой модели можно применить автоматический верификатор для определения ошибок в ее логике (о требованиях к верифицируемой программе можно прочитать, например, в работе [9]). Таким образом, при достаточном развитии генератора автоматов и трассировщика, после добавления к ним верификации, будет получен комплекс для тестирования и отладки программ с использованием автоматного подхода. Такой подход обеспечит простоту применения перечисленных выше эффективных методов отладки программ, которые в общем случае не могут быть использованы, либо требуют значительных ухищрений. Авторы надеются, что в скором времени по любой программе на языке *C* будет возможно построить автоматную программу, и это сможет решить проблему со слабой распространенностью указанных выше методов.

Источники

1. *Туккель Н. И., Шалыто А. А.* Преобразование итеративных алгоритмов в автоматные // Программирование, 2002. № 5, с.12–26. http://is.ifmo.ru/works/iter/
2. *Туккель Н. И., Шамгунов Н. Н., Шалыто А. А.* Реализация рекурсивных алгоритмов на основе автоматного подхода // Телекоммуникации и информатизация образования, 2002, №5, с.72-99
3. *Корнеев Г. А., Шалыто А. А.* Метод преобразования программ в систему взаимодействующих автоматов // Труды II межвузовской конференции молодых учёных. 2005, с. 65–72. http://is.ifmo.ru/works/_a_formalization.pdf
4. *Корнеев Г.А., Шалыто А.А.* Vizi — язык описания логики визуализаторов алгоритмов // 2005, 8 с. http://is.ifmo.ru/works/_a_language.pdf
5. *Ахо А., Сети Р., Ульман Д.* Компиляторы. Принципы, технологии, инструменты. М.: Вильямс, 2002, 767 с.
6. *Agrawal H., Spafford E.*. An execution backtracking approach to program debugging / Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference. 1988, pp. 283–299.
7. *Сайт компании Undo Software* – разработчика коммерческого обратного трассировщика и отладчика *UndoDB*. http://www.undo-software.com/
8. *Agrawal H., Horgan J.* Dynamic Program Slicing. Technical Report SERC-TR-56-P. Software Engineering Research Center. Purdue University, 1989.
9. *Корнеев Г. А., Парфенов В. Г., Шалыто А. А.* Верификация автоматных программ /Тезисы докладов Международной научной конференции, посвященной памяти профессора А. М. Богомолова. 2007, с. 66–69. http://is.ifmo.ru/verification/_KNIT-2007.pdf