МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет информационных технологий
Кафедра параллельных вычислений

Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Программная инженерия и компьютерные науки

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Усенко Никиты Сергеевича

Тема работы:

ОБНАРУЖЕНИЕ СЕМАНТИЧЕСКИХ ОШИБОК ВО ФРАГМЕНТИРОВАННЫХ ПРОГРАММАХ ДЛЯ СИСТЕМЫ LUNA ПРИ ПОМОЩИ ТЕХНОЛОГИИ MODEL CHECKING

«К защите допущена»
Заведующий кафедрой,
д.т.н., профессор
Малышкин В.Э./.....
(ФИО) / (подпись)
«……»……2025г.

Руководитель ВКР
к.т.н.
доц. каф. ПВ ФИТ НГУ
Власенко А.Ю./.....
(ФИО) / (подпись)
«……»……2025г.

Соруководитель ВКР

ст. преп. каф. ПВ ФИТ НГУ
Матвеев А.С./.....
(ФИО) / (подпись)
«……»……2025г.

Новосибирск, 2025

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет информационных технологий
Кафедра параллельных вычислений

Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Программная инженерия и компьютерные науки

УТВЕРЖДАЮ
Зав. кафедрой Малышкин В.Э.
(фамилия, И., О.)
…….
(подпись)
«17» января 2025 г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студенту Усенко Никите Сергеевичу, группы 21206
(фамилия, имя, отчество, номер группы)

Тема: Обнаружение семантических ошибок во фрагментированных программах для системы LuNA при помощи технологии Model Checking
(полное название темы выпускной квалификационной работы)

утверждена распоряжением проректора по учебной работе от 21 октября 2024 № 0377
скорректирована распоряжением проректора по учебной работе от 17.01.2025 № 0020
Срок сдачи студентом готовой работы 20 мая 2025 г.
Исходные данные (или цель работы): проектирование и разработка системы отладки LuNA-программ методом верификации на моделях.
Структурные части работы: предметная область, верификация на моделях, разработка анализатора MC-analyzer, тестирование.

Руководитель ВКР
к.т.н.,
доц. каф. ПВ ФИТ НГУ
Власенко А.Ю./……...
«17» января 2025 г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ
Матвеев А.С./……..
«17» января 2025 г.

Задание принял к исполнению
Усенко Н.С. /….....
(ФИО) / (подпись)
«17» января 2025 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Программы разделяют на два вида: последовательные и параллельные. Параллельное программирование позволяет добиться значительного ускорения при численном моделировании и обработке “больших данных”. Достигается это разбиением задач на несколько частей, которые выполняются на разных ядрах процессора / разных процессорах / разных компьютерах одновременно.

Написание параллельной программы требует высокой квалификации разработчика, поскольку порядок выполнения задач не определен, а результаты могут зависеть от их взаимодействия. Для таких программ характерны такие ошибки, как состояния гонки данных при попытке изменить общие ресурсы, состояния взаимной блокировки (дедлока) при циклическом ожидании ресурсов и прочие специфические ошибки, обусловленные взаимодействием процессов или потоков программы. Эти проблемы бывает тяжело заметить, поскольку они могут долгое время не проявлять себя.

Чтобы упростить разработку параллельных программ, существуют системы, позволяющие абстрагироваться от прямого управления потоками/процессами и их синхронизации. Одной из них является система LuNA (Language for Numerical Algorithms) [1, 2]. Она реализует концепцию активных знаний, т.е. позволяет автоматически конвертировать набор экспертных знаний о предметной области в программу.

LuNA на основе первичных знаний строит алгоритм с начала до конца. При этом она обладает автоматическим балансировщиком вычислительной нагрузки. Поскольку LuNA берет на себя всю работу по реализации параллелизма, это позволяет значительно снизить трудозатраты разработчика.

При написании кода на LuNA все еще есть возможность допустить ошибку. Примерами являются синтаксические ошибки, определяемые компилятором при нарушении формальной грамматики языка. Если программа синтаксически корректна, то могут возникнуть ошибки в участках кода, логика которых приводит к некорректному поведению программы (семантические ошибки).

Традиционные подходы к отладке (ручной анализ, диалоговая отладка, тестирование) часто не могут гарантировать полного покрытия всех возможных сценариев исполнения, особенно в случае асинхронных и параллельных программ. Поэтому необходимы автоматизированные методы отладки.

Верификация на моделях [3] – построение конечной модели программы и проверка выполнения на ней требуемых свойств, представленных в виде формул формальной логики, принимающих истинное значение в каждом состоянии модели.

При построении модели исходная программа упрощается путем исключения несущественных состояний. Количество состояний делается конечным, но при этом модель продолжает соответствовать исходной программе. Для такой модели гарантируется, что по нарушению свойств в модели можно будет говорить о нарушении свойств и в исходной программе.

Предполагается, что верификация на моделях позволит находить такие ошибки в LuNA-программах, которые не могут быть найдены традиционными способами анализа кода.

Цель ВКР – проектирование и разработка системы отладки LuNA-программ методом верификации на моделях.

Для достижения этой цели поставлены следующие **задачи**:

- проанализировать ошибки, присущие фрагментированным программам [4, 5];
- разработать программное средство, реализующее обнаружение ошибок в LuNA-программах методом верификации на моделях;
- интегрировать разработанное программное средство в существующую систему автоматизированной отладки ADAPT [6];
- провести нагрузочное тестирование.

Научная новизна работы состоит в том, что метод Model Checking впервые применен для автоматизированного обнаружения семантических ошибок во фрагментированных программах.

Практическая ценность работы состоит в автоматическом обнаружении ошибок в LuNA-программах во время разработки и повышения скорости выхода конечных программных продуктов.

Работа состоит из введения, трех глав и заключения. В первой главе дается определение предметной области и проводится обзор существующих подходов к анализу параллельных программ, включая верификацию на моделях и традиционные методы анализа. Во второй главе описан метод верификации на моделях и его практическая применимость для обнаружения ошибок. В третьей главе описан процесс разработки системы для автоматического обнаружения ошибок в LuNA-программах, тестовые эксперименты и их анализ.

Глава 1. Предметная область.

1.1 Введение в предметную область

1.1.1 Проблемы параллельного программирования

Разработка параллельных программ помогает увеличить производительность современных компьютеров, но она особенно трудоемка при отладке и тестировании. Разработчику нужно обращать внимание на очень много факторов, которых зачастую нет в последовательных программах. Одна из крупнейших проблем – это ошибки, возникающие из-за взаимодействия параллельно работающих частей.

Одна из самых частых и опасных ошибок в параллельном программировании – это гонка данных. Гонка за совместный доступ к данным возникает, когда два или более потока (или процесса) одновременно пытаются обратиться к одному и тому же ресурсу (как правило, объекту в памяти), при этом хотя бы один из них пытается ее изменить. Это приводит к неопределенности в программе, потому как результат зависит от того, в каком порядке они будут работать. Порядок их выполнения может с каждым запуском программы меняться. Состояние гонки находить наиболее трудно, так как она часто не является очевидной: программа может работать правильно очень долго до того, как проблема проявит себя. Чтобы избежать гонок – следует использовать механизмы синхронизации доступа к данным.

Другую типичную ошибку, которую могут допустить разработчики многопоточных программ, называют взаимной блокировкой (дедлок): два или несколько потоков ожидают друг от друга освобождение ресурсов, чтобы продолжить выполнение работы. В результате ни один из них не продвигается вперед. Так, например, даже один такой цикл в графе ожидания ресурсов в системе реального времени (например, в СУБД) – проблема. Дедлок может возникать, например, из-за неправильной последовательности захвата ресурсов, отсутствия таймаутов.

Инструменты распараллеливания, например MPI [7] и OpenMP [8], предоставляют разные парадигмы и, соответственно, свои собственные аспекты использования. При этом ошибки могут быть аналогичные [9].

Ошибки в OpenMP-программах

OpenMP – это стандарт, позволяющий создавать многопоточные приложения. При этом потоки работают в едином адресном пространстве процесса. Примеры ошибок, которые характерны для OpenMP:

- Гонки данных часто возникают из-за неправильного объявления переменных в параллельных секциях, например, когда переменная должна быть private, но по умолчанию используется как shared.
- Неправильное определение области действия переменных в параллельных блоках может привести к ошибкам, связанным с некорректным использованием значений.

Ошибки в MPI-программах

MPI (Message Passing Interface) – это интерфейс для передачи сообщений между процессами. Каждый процесс имеет свое собственное адресное пространство, а взаимодействие происходит через явно заданные отправку и прием сообщений. Ошибки встречающиеся в MPI-программах:

- Дедлок: может возникнуть, если два и более процесса вызывают MPI_Send друг к другу, образуя тем самым цикл, в котором ни один процесс не успевает вызвать MPI_Recv. Каждый процесс будут заблокирован на ожидании приема сообщения от другого.
- Забытые вызовы MPI_Init или MPI_Finalize могут привести к неопределенному поведению.
- Потеря данных: передача данных с некорректными типами и размерами (например, передали 100 целых чисел, а ожидали 50 чисел с плавающей точкой) может привести к переполнению или потере данных.
- При построении неэффективных коммуникационных схем (например, все ко всем) программа будет тратить больше времени на пересылку, чем на вычисления.

Способы синхронизации

Для обеспечения синхронизации доступа к ресурсам используют различные механизмы и примитивы. Ниже приведены наиболее часто используемые:

- Мьютекс – позволяет потокам/процессам поочередно получать доступ к разделяемому ресурсу.
- Семафор – позволяет ограничить количество потоков на основе счетчика потоков/процессов, имеющих доступ к разделяемому ресурсу.
- Условная переменная – применяется для организации ожидания наступления условия в одном потоке, о чем будет передано сообщение через другой поток.
- Барьер – синхронизирует выполнение группы потоков, которые будут ожидать завершения выполнения друг друга на определенной точке исполнения.

Общие проблемы параллельного программирования

Каждая параллельная программа подвержена проблемам с производительностью: многие алгоритмы трудно распараллеливать эффективно. Добавление новых потоков или узлов может не только не ускорить выполнение, но и замедлить его из-за накладных расходов на синхронизацию и обмен данными.

Создание потоков, переключение контекста, синхронизация и коммуникация потребляют ресурсы. Зачастую вычислительные задачи настолько малы, что их параллельное выполнение становится неэффективным с точки зрения затрат на коммуникации между потоками/процессами.

Если один поток или процесс выполняет большую часть работы, в то время как другие простаивают (дисбаланс нагрузки), общая производительность значительно падает. Это критическая проблема особенно в OpenMP со статическими распределениями и в MPI при неправильной балансировке.

Из-за недетерминированного исполнения некоторые ошибки могут возникать редко, зависеть от количества потоков или от конкретной архитектуры. Это делает тестирование и верификацию параллельных программ в особенности трудоемкими.

Для решения вышеуказанных проблем используется множество инструментов: статические и динамические анализаторы кода, профилировщики, средства визуализации и трассировки исполнения. Тем не менее, даже при использовании таких инструментов разработка эффективных и надежных параллельных программ остается задачей, требующей экспертизы и глубокого понимания архитектурных особенностей используемой платформы.

1.1.2 Особенности LuNA

Система *LuNA* является технологией, реализующей концепцию активных знаний. Она разрабатывается в Институте вычислительной математики и математической геофизики Сибирского отделения Российской академии наук (ИВМиМГ СО РАН).

Обозначим ключевые термины, принятые в системе LuNA:

- Фрагмент кода (ФК) – минимальная самостоятельная часть алгоритма, реализуемая в виде процедуры с четко выделенными входами и выходами, которую можно использовать в разных местах алгоритма.
- Фрагмент данных (ФД) – конкретная часть информации, которая является входом или выходом для ФВ.
- Фрагмент вычислений (ФВ) – конкретное выполнение определенного ФК, которое служит для порождения новых ФД, используя уже вычисленные.

LuNA спроектирована таким образом, чтобы построить наиболее эффективный алгоритм исходя из набора экспертных знаний из предметной области. Она позволяет описывать алгоритмы через ФД и ФК. Программист задает только общую логику программы, а *LuNA* преобразует ее в оптимизированный для параллельных вычислений код, равномерно распределяя вычислительную нагрузку между доступными процессорными ядрами.

Есть несколько сфер применения системы *LuNA*. При обработке больших данных и численном моделировании программисту достаточно определить алгоритм обработки на уровне *ФД*. Данные автоматически распределятся между ядрами за счет разбиения алгоритма на независимые слои из ФВ, которые могут высчитываться параллельно.

1.1.3 Классы ошибок в LuNA-программах

Ошибки в *LuNA-программах* можно разделить на следующие две категории: синтаксические и семантические. Подробный перечень классов ошибок доступен в базе ошибок LuNA [10], созданной в том числе при участии автора данной работы. В настоящий момент она насчитывает 12 синтаксических и 14 семантических классов ошибок, некоторые из которых также делятся на подклассы.

Синтаксические ошибки

Ошибки, которые не соответствуют формальной грамматике языка способен определить компилятор *LuNA*. Далее идут примеры популярных ошибок, которые разработчик может допустить в LuNA-программах.

SYN4 – более одного описания фрагментов данных внутри блока

Листинг 1 содержит синтаксическую ошибку, поскольку допускается лишь единожды за блок кода объявить набор ФД.

Листинг 1 – Пример LuNA-программы с SYN4

```
sub main() { df a; df b; }
```

Вывод компилятора представлен в листинге 2.

Листинг 2 – Реакция компилятора на SYN4

```
luna: compile error: syntax error at ./main.fa:1
sub main() { df a; df b; }
                here--^
```

Компилятор не всегда может настолько информативно указать на причину возникновения ошибки. Чаще всего это будет трассировка, содержащая вызовы функций в компиляторе. К тому же он не может информировать о более чем одной ошибке за одну попытку компиляции.

SYN7 – Нет main

LuNA-программа не компилируется, если ФК с именем main не будет объявлен. Пример такой ошибки представлен в листинге 3.

Листинг 3 – Пример LuNA-программы с SYN7

```
C++ sub empty() ${{$}}
sub foo() { empty(); }
```

Компилятор сообщает об этой ошибке, если она возникает.

SYN9 – попытка использования необъявленного идентификатора

Использование такого имени ФД, который не был объявлен ранее, недопустимо в LuNA-программах и приведет к ошибке компиляции. Пример такой ошибки представлен в листинге 4.

Листинг 4 – Пример LuNA-программы с SYN9

```
C++ sub bar(real a) ${{$}}
sub main() { foo(x); }
```

SYN6.1 – импортирование ФК под именем уже существующего ФК

Также, можно отнести к синтаксическим ошибкам объявление нескольких ФК с одним и тем же именем. В листинге 5 представлен пример таких объявлений.

Листинг 5 – Пример LuNA-программы с SYN6.1

```
import printa(real) as print;
import printb(real) as print;
sub main(){ print(1.0); }
```

Из-за особенности *LuNA-программы* будет использован лишь последний ФК из объявленных. Это не всегда очевидно для разработчика, а компилятор никак не реагирует на такого рода ошибки.

SYN6.2 – объявление структурированного ФК с именем уже существующего ФК

Ошибка, аналогична предыдущей. Во время выполнения будет использоваться последний объявленный ФК с таким же именем. Пример программы с такой ошибкой представлен в листинге 6.

Листинг 6 – Пример LuNA-программы с SYN6.2

```
C++ sub print(int a) ${{ printf("%d\n", a); $}}
sub foo() { print(1); }
sub foo() { print(2); }
sub main() { foo(); }
```

Семантические ошибки

Семантические ошибки могут влиять на поведение запущенной программы. Подобные ошибки наиболее опасны, поскольку могут долго не проявлять себя, а компилятор не предоставляет достаточных механизмов для их автоматического обнаружения. К тому же информации, которую предоставляет среда выполнения, недостаточно для качественного анализа причин возникновения ошибки.

SEM2.1 – повторная инициализация одиночного ФД

В LuNA-программах нельзя присвоить значение ФД, если некоторое значение ему уже присвоено. Попытка сделать это, приведет к ошибке во время исполнения. Ошибка может долго оставаться незамеченной, например, если используется оператор delete для удаления ФД. В листинге 7 представлен пример повторной инициализации.

Листинг 7 – Пример LuNA-программы с SEM2.1

```
import set_value(name) as set_value;
sub main() {
    df a;
    set_value(a[1]);
    set_value(a[1]);
}
```

SEM3.6 – использование ФД после его удаления

В системе LuNA можно управлять жизненным циклом ФД. Для этого есть такой механизм, как сборка мусора. Есть различные способы, чтобы контролировать этот процесс. Например, если известно количество потреблений ФД, то при помощи рекомендации *delete* (либо оператора “**-->**”) можно указать системе, что очистка должна происходить после завершения

вычисления ФК (атомарного либо структурированного), завершения цикла for, while, условия if и т.д. В листинге 8 приведены примеры того, как можно удалить ФД в LuNA-программе.

Листинг 8 – Примеры удаления ФД

```
while (x == 1) i = 1..out iter { print(y); } @ { delete x; }
for i = 1..10 { print(y); } @ { delete x; }
if (x == 1) { print(y); } @ { delete x; }
cf(x, y, z) @ { delete x, y; };
cf(x, y, z) --> (x, y);
```

Ручная очистка ФД подвержена ошибкам на этапе разработки. Возможны такие ситуации, когда после сборки мусора какой-то из атомарных ФК запросит ФД, который был удален и никогда впоследствии не будет установлен. Осложняет ситуацию также то, что каждый атомарный ФК может начать выполняться асинхронно в любой момент (при условии, что все нужные ему для использования ФД были проинициализированы).

Если после удаления *ФД* где-то еще будет использован тот же *ФД*, то существует риск зависания либо ошибки времени выполнения. В листинге 9 приведен пример LuNA-программы, содержащий эту ошибку.

Листинг 9 – Пример LuNA-программы с SEM3.6

```
C++ sub init(name x, int v) ${{ x = v; $}}
C++ sub print(int x) ${{$}}
sub main() {
    df x;
    init(x, 1) @ { delete x; };
    print(x);
}
```

Вывод: `luna: fatal error: run-time error: errcode=-6`

SEM4 – неиспользуемый ФД

Поскольку в LuNA-программе ФК не могут иметь побочных эффектов, то инициализация ФД, который в дальнейшем ни при каких условиях не будет использоваться не имеет смысла. Данная ошибка сигнализирует о том, что программист либо забыл использовать ФД, либо может удалить его

инициализацию без нарушения логики программы. Пример этого приведен в листинге 10.

Листинг 10 – Пример LuNA-программы с SEM4

```
C++ sub init(name a, int v) ${{ a = v; $}}
sub main() {
    df a;
    init(a, 1);
}
```

SEM5 – формула в if/while тождественно истинна/ложна

Может случиться ситуация, когда выражение в условии может быть упрощено до тождественной истины или лжи. Это не всегда может быть очевидным из-за сложности выражений. Обнаружение такой ошибки помогает разработчику определить место в программе, где можно упростить ветвление. Пример тождественно истинной формулы в условном выражении приведен в листинге 11.

Листинг 11 – Пример LuNA-программы с SEM5

```
{... if a <= b || b > a { print(1); } ...}
```

SEM6 – формула в if/while истинна/ложна во всех путях выполнения

Выражение в условии может всегда давать одинаковый результат не только из-за логической тождественности, как в SEM5. Это может происходить в том числе из-за того, что все возможные пути исполнения идут по одному сценарию, как это представлено в листинге 12.

Листинг 12 – Пример LuNA-программы с SEM6

```
sub foo(int a) {
    if a > 0 { print(a); }
}
sub main() { foo(1); foo(2); }
```

1.2 Существующие подходы к анализу программ

Анализ выполняется в первую очередь для автоматизации обнаружения ошибок либо для оптимизации программы.

1.2.1 Динамический анализ

Основа проверки программы средствами динамического анализа [11] – ее анализ во время выполнения. Такой способ обладает минимальным количеством ложных срабатываний, но обладает рядом недостатков:

- так как исследуются только те части кода, которые выполняются в тестируемых сценариях, то будет ряд потенциальных ошибок, которые не проявились при данном запуске;
- анализ возможен только после того, как программа уже готова к запуску, что ограничивает его применимость и затрудняет поиск синтаксических ошибок.
- есть вероятность, что из-за наличия такого анализа, результаты будут искажены, из-за чего будут сделаны неверные выводы.

1.2.2 Статический анализ

Статический анализ [12, 13], в отличии от динамического, выявляет ошибки исходя из исходного кода программ. При этом запуск самой программы не совершается. Обычно для этого исследуются промежуточные представления [14] из исходной программы (AST [15], Control Flow Graph [16, 17] и т.п.), которые удобны для поиска определенных видов ошибок. Преимущества статического анализа:

- выявление ошибок на самых ранних стадиях разработки;
- возможность охватить весь код, включая редко исполняемые участки, которые проблематично тестировать при реальном выполнении;
- возможность использования формальных методов для доказательства корректности программы относительно заданных свойств.

Из недостатков статического анализа можно выделить высокую вероятность ложных срабатываний (отчет об ошибках, которые отсутствуют в программе), поскольку у него отсутствует информация о реальных данных, которые будут обработаны программой.

1.3 Рассмотренные средства статического анализа

1.3.1 Simple Promela INterpreter (SPIN)

SPIN [18] – инструмент для верификации параллельных программ методом *верификации на моделях* (*Model Checking*) [19]. Он проводит поиск всех возможных состояний системы, описанных на языке *Promela* и проверяет, удовлетворяют ли они заданным свойствам.

Promela – процедурный недетерминированный язык с возможностью для моделирования параллелизма, напоминающий язык *C*. Свойства модели задаются в нем при помощи определения спецификации через *LTL* [20].

1.3.2 TLC

TLA+ – специальный язык, который предложил Лесли Лампорт для описания и проверки корректности алгоритмов и систем, особенно в контексте параллелизма и распределенных вычислений. Он предназначен для описания поведения систем на высоком уровне абстракции, с помощью математических структур и логических формул. В TLA+ на основе состояний системы и их переходов строятся как модели, так и свойства системы.

В основе TLA+ лежит теория темпоральной логики действий, с помощью которой можно описать и проверить все основные свойства системы. Решение этих задач осуществляется с помощью специального инструмента TLC, который проверяет все возможные состояния, выведенные из спецификации, и находит нарушения заданных свойств.

1.3.3 SonarQube

SonarQube использует обширные базы правил и метрик для различных языков программирования и анализирует код с точки зрения соответствия этим правилам. Он легко интегрируется с *CI/CD*, позволяя настроить профили качества, обладает простым и понятным интерфейсом, но не имеет возможностей для детального анализа сложных логических ошибок.

SonarQube Server представляет собой центральный компонент платформы SonarQube. Он обладает веб-интерфейсом и базой данных для хранения

результатов анализа, метрик и т.п. *SonarQube Scanner* выполняет статический анализ кода и отправляет полученные данные на сервер.

1.3.4 Clang Static Analyzer (CSA)

CSA является частью инфраструктуры *Clang*. Он используется для поиска ошибок программ на C и C++. Он анализирует исходный код на разных этапах компиляции, умеет находить утечки памяти, неопределенное поведение, ошибки при работе с памятью.

CSA проходит несколько этапов, чтобы завершить анализ. *Clang Frontend* переводит код в промежуточное представление *AST*, после чего строит *Control Flow Graph* (множество всех возможных путей исполнения программы). Далее происходит анализ кода при котором отслеживаются значения переменных и состояния программы во всех возможных путях выполнения.

1.4 Анализ подходов и предложение решения

SPIN является перспективным для анализа кода за счет того, что обладает собственным языком для создания моделей программ. Эта универсальность позволяет производить верификацию средствами данного инструмента даже для системы *LuNA*. Для этого необходимо из *LuNA-программы* сделать модель на языке *Promela*, и определить свойства через *LTL*. Всю остальную работу сделает *SPIN*, предоставив разработчику подробный отчет при возникновении нарушения свойств.

Использование Верификации на моделях через систему *SPIN* путем автоматической трансляции произвольной *LuNA-программы* в Promela-программу позволит искать сложные семантические ошибки на ранних стадиях разработки.

1.5 Требования к анализатору

1.5.1 Функциональные требования

- Должны поддерживаться синтаксически корректные LuNA-программы.
- Программа должна автоматически транслировать исходный код на языке LuNA в модель на Promela, пригодную для анализа верификатором SPIN.

- Программа должна проводить проверку LTL-свойств над построенной моделью, с целью выявления ошибок.
- Информация о найденных ошибках должна быть представлена в формате, совместимым с ADAPT.

1.5.2 Нефункциональные требования

- Программа должна работать в Unix-подобных операционных системах.

Глава 2. Верификация на моделях

Верификация на моделях – методика, основанная на построении конечной модели системы и проверки выполняются ли на данной модели требуемые свойства. Проверка выполняется, как исчерпывающий поиск по пространству состояний, поэтому подвержена комбинаторному взрыву. Чтобы верификация была возможной, модель должна соответствовать исходной системе и проверяемым свойствам.

2.1 Основные понятия

У исходной системы (например, LuNA-программы) состояний может быть бесконечное количество. Под состоянием стоит понимать совокупность значений объектов данных и счетчика управления. Для прикладного кода максимальное количество состояний определяется числом всех возможных комбинаций значений переменных в каждой точке выполнения программы.

В ходе построения модели происходит абстрагирование от состояний, которые не важны для проверки интересующих свойств.

Свойство программы – совокупность значений объектов данных и счетчика управления. Для прикладного кода максимум состояний определяется числом всех возможных сочетаний значений переменных для каждой точки выполнения программы.

Достижимое состояние – такое состояние, которое может быть достигнуто в ходе вычисления программы, в котором оно присутствует. Заметив определенные закономерности в программе, можно сузить число достижимых состояний, но в общем случае эта задача алгоритмически неразрешима.

Модель должна достаточно точно повторять структуру исходной программы, чтобы можно было легко восстановить вычисление в случае нахождения контрпримера (последовательности состояний, приводящей к нарушению свойства).

2.2 Абстрагирование от состояний

У исходной системы состояний может быть большое или даже бесконечное количество. Чтобы избежать “комбинаторного взрыва” и сделать

верификацию выполнимой, используются различные методы абстрагирования от тех состояний, которые не влияют на проверку интересующих свойств.

Абстрагирование от лишнего кода – удаление кода, вычисление которого не сможет повлиять на проверяемые свойства. Это не относится к тому, что влияет на ветвление или иным образом может повлиять на значения переменных, которые определены в проверяемых свойствах модели.

Абстракция предикатов – замена ветвления по условию на недетерминированное ветвление. Позволяет абстрагироваться от переменных в условиях, используя вместо них истинность логических выражений.

Абстракция типов данных – замена диапазонов значений переменных на набор классов эквивалентности. Позволяет уменьшить число состояний модели, сохраняя логику ветвлений в ней. Например, если все положительные числа обрабатываются одинаковым образом, то не имеет смысла проверять их все, достаточным будет выделить таковые в отдельный класс.

Абстракция наблюдаемых переменных – удаление переменных, которые не влияют на выполнимость заданных свойств. Но при этом допускается только расширение множества возможных вычислений, иначе это может нарушить корректность модели, делая важные для проверки вычисления исходной системы недостижимыми.

Абстракция от функций и системных вызовов – замена функций на синхронный обмен сообщений с процессом-заглушкой. Это позволяет сохранить структуру вызовов функций, давая возможность проверять свойства без доступа к конкретной реализации.

2.3 Классы свойств

2.3.1 Безопасность

Свойства безопасности гарантируют, что нежелательные состояния в ходе выполнения системы никогда не будут достигнуты. Свойство выполняется, если условие никогда не срабатывает. Нарушение такого свойства означает наличие контрпримера, в котором достигается недопустимое состояние. Примеры свойств безопасности:

- невозможность одновременного доступа нескольких потоков к критической секции.
- отсутствие выхода за границы допустимых значение (деление на ноль, выход за границы массива)

2.3.2 Живучесть

В отличии от свойств безопасности, свойства живучести гарантируют, что необходимое состояние в конечном итоге будет достигнуто. Такие свойства предотвращают ситуации вечного ожидания и выполняется, если условие срабатывает хотя бы 1 раз. Примеры свойств живучести:

- корректный запрос пользователя будет обработан за конечное время;
- поток, получивший разрешение на вход в критическую секцию, когда-то в будущем войдет в нее.

2.4 Темпоральные логики LTL и CTL

LTL (Логика линейного времени) и *CTL* (логика деревьев вычислений) [21] – это виды темпоральной логики для спецификации и верификации свойств систем.

Темпоральные операторы	Кванторы пути
X – в следующем состоянии	***A*** – на всяком пути
F – рано или поздно	***E*** – существует путь
G – всегда	
U – до тех пор	

LTL описывает поведение систем вдоль линейных последовательностей событий с операторами ***X***, ***F***, ***G***, и ***U***. Она нужна для описания свойств, которым должны удовлетворять линейные последовательности наблюдаемых состояний (трассы). LTL состоит из формул вида ***A*** *f*, где *f* – это формула пути ($p \in$ ***AP***, $\neg f$, $f \wedge g$, $f \vee g$, ***X*** *f*, ***F*** *f*, ***G*** *f*, f ***U*** g). Здесь ***AP*** – множество атомарных высказываний.

Примеры LTL-свойств в виде графа приведены на рисунке 1. Здесь узел – какое-то состояние, а ребро – переход из одного состояния в другое.

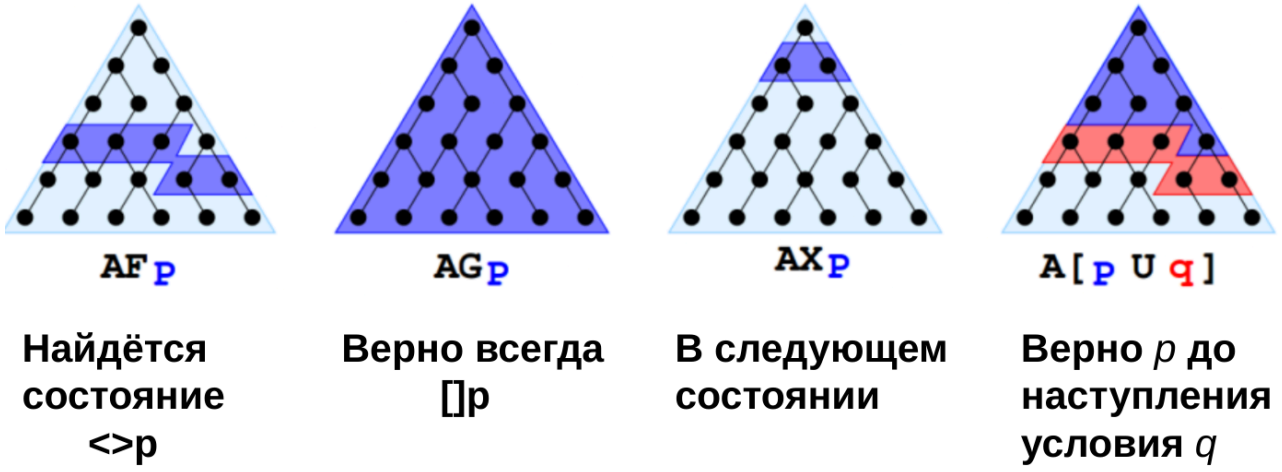

Рисунок 1 – Примеры свойств LTL

CTL работает с разветвляющимися временными структурами, используя кванторы ***A*** и ***E*** в комбинации с темпоральными операторами. Такие темпоральные логики позволяют формально доказывать соответствие системы определенным спецификациям. LTL подходит для свойств, справедливых для всех последовательностей состояний, а CTL полезна для анализа систем с разными возможными будущими исходами.

CTL представляет собой фрагмент CTL*, в котором каждый темпоральный оператор X, F, G, U должен следовать непосредственно за квантором пути (A или E).

Поскольку в CTL после квантора пути обязан идти темпоральный оператор не существует CTL-формулы, эквивалентной LTL-формуле:

***A (FG** p)*

(на всяком пути найдется состояние, начиная с которого p будет выполняться всегда)

Поскольку LTL не позволяет выражать свойства, связанные с существованием пути, не найдется такой LTL-формулы, которая была бы эквивалентна следующей CTL-формуле:

***AG (EF** p)*

(из любого состояния достижимо состояние, в котором истинно p)

Также, дизъюнкция этих двух формул ***A (FG p ∨ AG (EF p))*** является формулой CTL*, которую нельзя выразить ни в CTL, ни в LTL.

2.5 Этапы верификации

Верификация программы обычно содержит следующие этапы, представленные на рисунке 2. Сначала изучается исходная система, чтобы сформулировать требования, которым она должна удовлетворять. Исходя из этих требований, выделяются классы свойств (безопасность, живучесть), которые необходимо проверить.

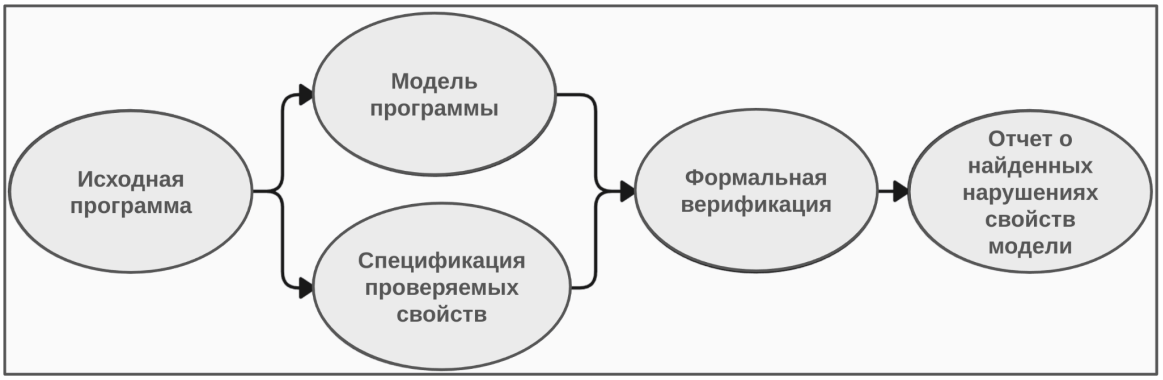

Рисунок 2 – Схема верификации

На основе структуры и поведения системы создается модель, подходящая для верификации. Свойства, подлежащие проверке, описываются при помощи формального языка спецификаций (например, темпоральной логики).

Модель и спецификация передается инструменту, производящему верификацию. По ее завершению возможны два варианта:

- Свойство выполнено – верификатор подтверждает, что система удовлетворяет заданному свойству.
- Свойство нарушено – верификатор предоставляет пользователю контрпример, демонстрирующий сценарий, при котором данное свойство нарушается.

2.6 Примеры практического применения верификации на моделях

Верификация на моделях имеет применение на практике, в сценариях, где важна корректность поведения системы.

2.6.1 Верификация алгоритма Паксос

Алгоритм Паксос используется в распределенных системах для достижения консенсуса вычислителей, которые могут выходить из строя. Важно, чтобы при любых сбоях или задержках все узлы согласовали одно и то же решение и не возникло ситуации, когда разные узлы принимают разные решения.

С помощью инструмента SPIN была создана модель алгоритма на языке Promela [22], которая позволила проверить все возможные состояния и убедиться, что алгоритм правильно работает в условиях сбоев и задержек. Это позволило гарантировать, что алгоритм Паксос будет функционировать надежно.

2.6.2 Обнаружение ошибок в Amazon Web Services (AWS)

Один из примеров использования TLA+ для верификации распределенных систем в AWS касается DynamoDB. Разработчики в AWS создали формальную модель [23] этой системы и в ходе верификации обнаружили три ошибки. Применение TLA+ позволило разработчикам AWS не только обнаружить ошибки, но и повысить уверенность в корректности проектируемых систем. Формальная спецификация обеспечила математическое описание желаемых свойств системы и позволило выявить потенциальные проблемы на ранних стадиях разработки.

2.6.3 SPIN

Promela

Модели в системе SPIN описываются языком Promela. Этот язык предоставляет следующие возможности.

Описание процесса конструкцией ***proctype name(args) {/*body*/}.*** Чтобы начать его выполнение, пишут код вида run ***name().*** Также, если написать ***active proctype***, то процесс будет добавлен в начальный набор процессов, которые начнут выполнение автоматически. Как минимум один активный процесс должен быть определен. SPIN анализирует файл с моделью и строит

конечный автомат каждого ***proctype****, где все процессы могут выполняться параллельно с разным порядком операций.*

Предоставляются целые типы данных (***bit, bool, byte, short, int***), которые различаются между собой диапазонами возможных значений. Для перечислений существует такой тип данных, как ***mtype***.

Для моделирования переходов в зависимости от условий существуют такие конструкции, как недетерминированное ветвление. Во время верификации, будет выбрано одно из ветвлений, в котором ***guard*** принимает значение ***true***. В листинге 13 приведен пример описания недетерминированного ветвления в Promela-программе.

Листинг 13 – Пример недетерминированного ветвления в Promela-программе

```
if
:: <guard1> -> <body>
...
:: <guardN> -> <body>
fi
```

Если необходима гарантия порядка порядка доступа к переменным, существует блок ***atomic { /*body*/ }.*** Он гарантирует, что все операции внутри блока будут выполнены без прерываний.

В языке Promela имеются директивы препроцессора для создания макросов и констант: ***#define***. В листинге 14 определяется константа MAX и макрос, проверяющий, что параметр меньше MAX.

Листинг 14 – Пример использования макросов в Promela-программе

```
#define MAX (10)
#define LT(x) ((x) < MAX)
```

В макросах можно конкатенировать токены. Делается это при помощи оператора “**##**”. В листинге 15 приведен пример использования конкатенации в макросах.

Листинг 15 – Пример конкатенации в макросах в Promela-программе

```
#define var(name) int var##name
var(1); // int var1;
```

LTL

Логические свойства системы формулируется при помощи LTL. Задаются они через конструкцию ***ltl name { /*formula*/ }***. При верификации может использоваться лишь одно из определенных свойств. Если верификация проходит успешно, то это значит, что во всех возможных сценариях исполнения, заданное свойство выполняется. Если же найден контрпример, то SPIN показывает последовательность действий, которая приводит к нарушению свойства. Примеры основных темпоральных операторов приведены в таблице 1.

Таблица 1 – Темпоральные операторы в Promela

Синтаксис	Описание	Пример
[]	в каждом состоянии	[](x >= 0)
<>	найдется состояние	<>(x == 1)
!	отрицание формулы	!error
&&	логическое И	(x != 0 && y != 0)
||	логическое ИЛИ	(x != 0 || y != 0)
->	логическая импликация	(x > 0 -> y > 0)

Стадии верификации

Верификация модели в SPIN происходит в несколько этапов.

Первый шаг – генерация кода верификатора командой ***spin -a model.pml***., где model.pml – Promela-файл описания модели. Эта команда создает ряд файлов на языке C. Основной файл, который реализует механизм проверки называется ***pan.c***.

Следующим шагом является компиляция сгенерированного кода командой ***gcc -o pan pan.c.*** Она создает исполняемый файл ***pan***, предоставляющий анализатор текущей модели. Через ключ ***-N*** можно указать LTL-свойство, которое будет проверяться: ***./pan -N LTL_name***.

Глава 3. Разработка и тестирование анализатора MC-analyzer

3.1 Основные структурные элементы моделей

Для основного метода анализа LuNA-программ, было выбрано средство SPIN. Определим то, по каким принципам и какие конструкции будут использоваться в Promela-модели и то, как они будут соответствуют тому, что записано в LuNA-программе.

<<varN>> – каждому ФД соответствует переменная в Promela-программе со своим номером, например: var1, var42, var101. Это нужно, чтобы идентификатор переменной оставался уникален, несмотря на вызовы структурированных ФК.

3.1.1 Объявление ФД

В LuNA программе базовое имя ФД может быть объявлено в начале блока кода при помощи ключевого слова df и последующего списка идентификаторов. Чтобы воспользоваться индексированным ФД, необходимо, чтобы его базовое имя было объявлено. Чтобы отразить объявление ФД в модели, используется макрос, приведенный в листинге 16, который устанавливает флаг объявления переменной.

Листинг 16 – Объявление ФД в Promela-программе

```
#define def(name) isdef_##name = true
```

Допустим, в LuNA-программе, присутствует следующий код:

```
{ df x; use(x[1][2]); ... }
```

Тогда в модели объявления будут выглядеть следующим образом:

```
def(var0); def(var1);
```

Как можно заметить из примера, индексированные ФД в модели используются как независимая переменная, что соответствует семантике их использования в языке LuNA.

3.1.2 Инициализация ФД

Инициализация ФД значением производится при помощи ФК. Он может быть как импортированный из .cpp файла, так и объявлен прямо в коде на

LuNA. Примеры, как это может выглядеть в LuNA-программе, приведены в листинге 17.

Листинг 17 – Пример объявления ФК с выходным ФД

```
import init_1(name, real) as init_1;
C++ sub init_2(name out_df, real value) ${{ out_df=value; $}}
```

В данных примерах первым параметром передается ФД с типом *name*. Данный тип указывает на то, что после того, как ФК отработает, переданный ФД будет проинициализирован каким-то значением.

Анализ будет производиться только для LuNA-программы. Из-за этого становится затруднительным вычисление конкретных значений, которые будут присвоены после инициализаций. Если принять во внимание данное ограничение, то при моделировании достаточно будет ввести счетчик инициализаций, как это представлено в листинге 18.

Листинг 18 – Инициализация ФД в Promela-программе

```
#define init(name) init_count_##name = init_count_##name + 1
```

Допустим, что встретился следующий код:

```
{ ... init_1(x, 1); init_1(x, 2); init_1(x[1], y); ... }
```

Здесь происходит 3 инициализации (две для ***x***, одна для ***x[1]***). ФД с базовым именем **y** в данном случае не инициализируется, поскольку передается с типом ***real***. Соответственно, фрагмент модели на Promela для данного примера будет выглядеть следующим образом:

```
init(var0); init(var0); init(var1);
```

3.1.3 Использование ФД

Использоваться ФД в LuNA-программе может только в случае, когда оно проинициализировано. Если же это не так, то вызов атомарного ФК, проверка условия с этим ФД и т.д. будут приостановлены до тех пор, пока значение не будет присвоено. Ниже можно увидеть примеры использования ФД **x**:

```
if x < 1 { ... }
while (x == 1), i=0..out it { ... }
use_value(x); //import use_value(real) as use_value;
```

Аналогично счетчику инициализаций, записывается счетчик использований. Макрос, служащий этой цели, представлен в листинге 19.

Листинг 19 – Использование ФД в Promela-программе

```
#define use(name) use_count_##name = use_count_##name + 1
```

Возьмем в качестве примера программу на LuNA:

```
import init(name, real) as init;
import use_value(real) as use_value;
{
    ...
    if x < y && x > 1 {
        init(a, y);
        use_value(b[1]);
    }
    ...
}
```

Для вычисления условия ***if*** здесь используются ***x*** и ***y***, причем x используется дважды. Для инициализации **a** используется **y**. Далее используется **b[1]**. Если положить:

```
x – var0, y – var1, a – var2, b[1] – var3,
```

то моделирование использований будет выглядеть так:

```
use(var0); use(var1); use(var0); // условие if
use(var1); use(var3);            // тело if
```

3.1.4 Зависимость по данным

Можно заметить, что инициализация ФД может зависеть от другого ФД. Для такого случая можно ввести такое понятие, как зависимость по данным. Например, пусть имеет место вызов следующего атомарного ФК:

```
import init(name, int) as init;
{... init(x, y); ...}
```

Чтобы проинициализировать **x**, необходим **y**. Положим x – var0, y – var1. В модели это запишется так: ***depends_on(var1, var0)***. Реализация этого макроса представлена в листинге 20 и определяется, как установка флага.

Листинг 20 – Зависимость по данным в Promela-программе

```
#define CONCAT3(a, b, c) a##b##c
#define depends_on(varA, varB) \
    CONCAT3(depends_on_, varA, _##varB)=true
```

3.1.5 Удаление ФД

Для моделирования вызова сборщика мусора через рекомендацию *delete* в LuNA программе будем использовать счетчик, связанный с переменной в модели на Promela. В листинге 21 представлен макрос, повышающий описанный счетчик на единицу.

Листинг 21 – Удаление ФД в Promela-программе

```
#define destroy(name)
destroy_count_##name=destroy_count_##name+1
```

Теперь, если встретится следующий фрагмент LuNA-программы:

```
{... print(a) @ { delete a; }; ...},
```

то его можно будет заменить на

```
{... use(var0); destroy(var0) ...}.
```

3.1.6 Конец видимости ФД

У всех ФД в LuNA есть зона видимости, после которой их использовать уже нельзя. Существуют такие свойства, которые имеет смысл проверять начиная с этого момента. В Promela можно использовать метки (именованные участки кода). Использование меток для этой цели приведено в листинге 22.

Листинг 22 – Конец видимости в Promela-программе

```
#define enddef(name) enddef_##name: isdef_##name = false;\
    use_count_##name = 0;\
    init_count_##name = 0;\
    destroy_count_##name = 0
```

Таким образом в модели для каждой переменной вида ***<<varN>>*** будет ***def(<<varN>>)*** в начале зоны видимости, и ***enddef(<<varN>>)*** в конце.

3.1.7 Условия

Совпадающие условия в разных if/while должны быть одновременно либо истинны, либо ложны. Рассмотрим фрагмент LuNA-программы в листинге 23.

Листинг 23 – Пример LuNA-программы с условными выражениями

```
{
...
    if (x == 1) {
        while (x == 1), i=0..out iter {
            if (x > y) { ... }
        }
    }
...
}
```

Пронумеруем все встреченные условия: (x == 1) – cond0, (x > y) – cond1. Будем считать, что если **<<*condN*>>** в модели принимает значение 1, то недетерминированное ветвление будет исполнять модель тела if/while, полагая условие истинным. Иначе будет считаться, что условие ложно. Производится это через подсчет количества раз, когда условие принимало истину или ложь. Реализовано это через макросы, представленные в листинге 24.

Листинг 24 – Счетчики истинных/ложных условий в Promela-программе

```
#define inc_true(name) true_count_##name =
true_count_##name+1
#define inc_false(name) false_count_##name =
false_count_##name+1
```

Листинг 25 – Пример Promela-модели для LuNA-программы в листинге 23

```
if // if (x == 1)
:: cond0 -> inc_true(cond0); ...
   if // while (x == 1), i=0..out tmp
   :: cond0 -> inc_true(cond0); ...
      if // if (x > y)
      :: cond1 -> inc_true(cond1); ...
      :: else -> inc_false(cond1); skip;
      fi
   :: else -> inc_false(cond0); skip;
   fi
:: else -> inc_false(cond0); skip;
fi
```

Как видно из листинга 25, для условия ***(x == 1)*** используется переменная cond0. Когда она принимает значение 1, увеличивается соответствующий счетчик через ***inc_true***. Моделирование тела if/while в данном случае опускается и предполагается. В случае же, когда переменная, описывающая условие, принимает 0, увеличивается счетчик через ***inc_false***, и тут же происходит выход из недетерминированного ветвления.

3.2 Поиск ошибок

3.2.1 Обнаружение ошибки SEM2.1

В качестве примера программы для поиска ошибки SEM2.1 (повторная инициализация ФД) рассмотрим программу из листинга 7, в которой дважды инициализируется ФД ***a[1]***.

Чтобы обнаружить ошибку методом верификации на модели, определим следующее LTL-свойство, представленное в листинге 26.

Листинг 26 – LTL-свойство для поиска SEM2.1

```
ltl SEM2_1_<<varN>> {[] (init_count_<<varN>> < 2)}
```

Ранее упоминалось, что при создании модели, потребление ФД с типом name заменяются на init, который увеличит счетчик инициализаций. Данное свойство проверяет, что в каждом состоянии модели счетчик инициализаций не превысит двух. В листинге 27 приведен пример использования этого свойства, при котором будет обнаружена ошибка.

Листинг 27 – Пример Promela-программы для поиска SEM2.1

```
ltl SEM2_1_var1 {[] (init_count_var1 < 2)}
active proctype main() {
   def(var0); def(var1);
   init(var1); init(var1);
   use(var1);
   enddef(var0); enddef(var1);
}
```

3.2.2 Обнаружение ошибки SEM3.1

Для поиска ошибки SEM3.1 (неинициализированный ФД используется вне цикла) определим вспомогательные конструкции. В листинге 28 приведен

шаблонный макрос, который будет конструировать цепочку импликаций на основе информации о количестве использований и инициализаций.

Листинг 28 – Проверка наличия инициализации, если было использование ФД

```
#define define_check_init(is_used, enddef_label, is_init)\
    (enddef_label -> (is_used -> is_init))
#define check_init(name)\
    define_check_init(use_count_##name > 0,\
                      main@enddef_##name,\
                      init_count_##name > 0)
```

Теперь при помощи check_init можно будет проверить, есть ли в конце видимости переменной хотя бы одна инициализация при наличии использования. LTL-формула и пример ее использования для обнаружения этой ошибки приведена в листингах 29-30.

Листинг 29 – LTL-свойство для поиска SEM3.1

```
ltl SEM3_1_<<varN>> {[] (check_init(<<varN>>))}
```

Листинг 30 – Пример Promela-программы для поиска SEM3.1

```
ltl SEM3_1_var0 {[] (check_init(var0))}
active proctype main() {
   def(var0); use(var0);
   enddef(var0);}
```

Нарушение свойства возникает после ***enddef(var0)***, поскольку ***use_count_var0*** в этот момент равен 1, но ***init_count_var0*** равен 0.

3.2.3 Обнаружение ошибки SEM3.2

Листинг 31 – Пример LuNA-программы с SEM3.2

```
C++ sub int_set(name x, int v) ${{ x = v; $}}
sub main() {
    df a, b, c;
    int_set(a, b);
    int_set(b, c);
    int_set(c, a); //программа зависает.
}
```

В листинге 31 приведен пример ошибки SEM3.2 (циклическая зависимость по данным), которая возникает вследствии возникновения цикла из

ФД (***a***, ***b***, ***c***), которые требуются для инициализации. При генерации модели на Promela, строится граф смежности, в котором определяются все возможные зависимости по данным, которые потенциально могут возникнуть во время исполнения. Этот граф реализован через словарь, в котором каждому ФД сопоставляются с множеством ФД, которые предположительно могут образовать зависимость по данным.

Во время верификации системой SPIN учитывается ветвление и, тем самым, фиксируются конкретные зависимости для каждого состояния модели. Потенциальные зависимости на этапе генерации исследуются при помощи цикла обхода в глубину для нахождения цикла в графе. При нахождении такового, формируется LTL-свойство, которое необходимо будет проверить средствами SPIN.

LTL-свойство представлено в листинге 32.

Листинг 32 – LTL-свойство для поиска SEM3.2

```
ltl SEM3_2_<<varM>>_..._<<varN>> {
    [] !(depends_on_<<varM>>_<<varK>> &&
         ...
         depends_on_<<varN>>_varM)}
```

<<varM>> ... <<varN>> – идентификаторы переменных в модели из которых формируется цикл (порядок имеет значение). Расшифровывается LTL следующим образом: для любого состояния модели верно, что все зависимости, входящие в состав цикла, не активны одновременно.

3.2.4 Обнаружение ошибки SEM3.6

Ошибка SEM3.6 (использование ФД после его удаления) имеет место в ситуации, когда после удаления ФД использовался как минимум дважды. Действительно, поскольку порядок выполнения инструкций в LuNA не определен, при конкретном запуске не исключается следующий порядок исполнения: использование, удаление и использование ФД. Для поиска возможных ошибок, связанных с использованием ФД после его удаления воспользуемся двумя соответствующими счетчиками.

Листинг 33 – LTL-свойство для поиска SEM3.6

```
ltl SEM3_6_<<varN>> {
    [] (destroy_count_<<varN>> > 0 -> use_count_<<varN>> <
2)}
```

Приведенная формула в листинге 33 описывает такое свойство модели, при котором в любом состоянии счетчик удалений может быть больше нуля только в том случае, если использований меньше двух. Пример обнаружения ошибки SEM3.6 приведен в листинге 34.

Листинг 34 – Пример Promela-программы для поиска SEM3.6

```
ltl SEM3_6_var0 {[] (
    destroy_count_var0 > 0 -> use_count_var0 < 2)}
active proctype main() { // sub main() {
   def(var0);            //   df a;
   init(var0);           //   init(1, a);
   use(var0);            //   print(a) @ {
   destroy(var0);        //     delete a; };
   use(var0);            //   print(a);
   enddef(var0);         // }
}
```

Из примера можно видеть принцип поиска данной ошибки:

- ***use(var0)*** установит ***use_count_var0*** в значение 1;
- ***destroy(var0)*** установит ***destroy_count_var0*** в 1;
- очередной вызов ***use(var0)*** вызовет нарушение свойства.

3.2.5 Обнаружение ошибки SEM4

Поиск данной ошибки SEM4 (неиспользуемый ФД) схож с поиском SEM3.1. Вспомогательные конструкции определены в листинге 35.

Листинг 35 – Проверка наличия использования, если была инициализация ФД

```
#define define_check_usage(is_used, enddef_label, is_init)\
    (enddef_label -> (is_init -> is_used))
#define check_usage(name)\
    define_check_usage(use_count_##name > 0,\
                       main@enddef_##name,\
                       init_count_##name > 0)
```

Воспользовавшись ***check_usage*** можно будет проверить, есть ли в конце видимости переменной хотя бы одно использование при наличии инициализации. Формула в LTL и пример ее использования приведены в листингах 36-37.

Листинг 36 – LTL-свойство для поиска SEM4

```
ltl SEM4_<<varN>> {[] (check_usage(<<varN>>))}
```

Теперь для программы из листинга 10, можно построить модель, которая обнаружит нарушение свойства.

Листинг 37 – Пример Promela-программы для поиска SEM4

```
ltl SEM4_var0 {[] (check_usage(var0))}
active proctype main() {
   def(var0);
   init(var0);
   enddef(var0);
}
```

Можно видеть, что после ***enddef(var0)*** возникает нарушение свойства. В этом состоянии ***init_count_var0*** в равен 1, но ***use_count_var0*** равен 0.

3.2.6 Обнаружение ошибки SEM5

С точки зрения вывода анализатора, SEM5 (формула в if/while тождественно истинна/ложна) – предупреждение (warning), которое помогает разработчику обнаружить места в программе, где условие всегда (или никогда не) выполняется. В листинге 38 приведен пример программы, в которой возникает эта ошибка.

Листинг 38 – Пример Promela-программы для поиска SEM5

```
import c_print_int(int) as print; // Объявление ФК print
sub main() {          // Объявление ФК main
    df x, y;          // Объявление ФД x, y
    if x>y || x<=y { // тождественная "истина"
        print(x);     // Вызов ФК print. Использование x
    }
}
```

Для обнаружения данного типа ошибки на этапе генерации модели выполняется анализ истинности условия в if/while. Применяется система правил вывода, при помощи которых упрощаются выражения, и определяется возможно ли вывести тождественную истину или ложь.

Основной процесс упрощения использует специализированную библиотеку “mathjs” (https://mathjs.org), предоставляющую как правила для упрощений формул по умолчанию, так и возможность задания пользовательских логических правил для упрощения формул. Данные правила позволяют последовательно преобразовывать исходное выражение в его упрощенный вид. Примеры таких правил включают:

- 'n1 < n2 -> n1 - n2 < 0'
- 'n1 < n2 or n2 < n1 -> n1 != n2'
- 'n1 < n2 and n1 >= n2 -> 0'
- 'n1 > n2 and n1 <= n2 -> 0'
- 'n1 == n2 -> n2 == n1'
- 'n1 != n2 -> n2 != n1'

Здесь n1, n2 – любые выражения. Например, правило (2) может быть использовано для упрощения (x + 1) < y || y < (x + 1) до (x + 1) != y.

Успешным упрощением считается такое, которое свелось к константе. Ложью считается константа, значение которой 0, а любое другое число считается истинным, что соответствует тому, как работают условные выражения в LuNA.

После этого, уже можно заключить, что ошибка возникает. При помощи LTL, приведенного в листинге 39, можно проверить, что в программе достижимо такое состояние, которое дойдет до проверки условия.

Листинг 39 – LTL-свойство для поиска SEM5

```
ltl SEM5_%condN% {
    [] ((true_count_%condN% == 0) &&
       (false_count_%condN% == 0))
}
```

Здесь проверяется такое свойство модели, что никогда не увеличится счетчик истинных или ложных ветвлений для условия condN. Его смысл в том,

чтобы сообщать об ошибке SEM5 только в случае достижимости конкретного условия в программе. В листинге 40 представлен фрагмент модели для поиска этой ошибки в LuNA-программе.

Листинг 40 – Пример Promela-программы для поиска SEM5

```
ltl SEM5_cond0 {
   [] ( (true_count_cond0 == 0) && (false_count_cond0 == 0) )
}
active proctype main() {
   ...
   init(var0); init(var1); // init(1, x); init(1, y)
   use(var0); use(var1); use(var0); use(var1);// x>y || x<=y
   if
   :: true -> inc_true(cond0);
              use(var0);
   fi
   ...
}
```

3.2.7 Обнаружение ошибки SEM6

Листинг 41 – Пример LuNA-программы с SEM6

```
import c_init_int(name, int) as init;
import c_print_int(int) as print;
sub foo(name x, int a) {
    if a > 0 {    // Всегда истина, учитывая все вызовы foo
        print(x); // Вызов ФК print. Использование x
    }
}
sub main() {
    df x;       // Объявление ФД x
    init(x, 1); // Вызов ФК. Инициализация x
    foo(x, 1);  // if 1 > 0
    foo(x, 2);  // if 2 > 0
}
```

В листинге 41 представлен пример ошибки SEM6 (формула в if/while истинна/ложна во всех путях выполнения), которая является предупреждением

с точки зрения анализатора. Для выявления этой ошибки используется LTL-свойство, представленное в листинге 42.

Листинг 42 – LTL-свойство для поиска SEM6

```
ltl SEM6_%condN% { [] (
  ((true_count_%condN% > 0) -> (<>(false_count_%condN% > 0)))
    &&
  ((false_count_%condN% > 0) -> (<>(true_count_%condN% > 0)))
)}
```

Расшифровывается данная формула LTL следующим образом. Для любого состояния модели:

- Если хотя бы раз условие было истинным, то найдется такое состояние, когда условие будет ложным.
- Если хотя бы раз условие было ложным, то найдется такое состояние, когда условие будет истинным.

Листинг 43 – Пример Promela-программы для поиска SEM6

```
active proctype main() {
   ...
   init(var0);
   if
   :: true -> inc_true(cond0); // if a > 0 (2 > 0)
              use(var0);       // print(x)
   fi
   if
   :: true -> inc_true(cond0); // if a > 0 (2 > 0)
              use(var0);       // print(x)
   fi
   ...
}
```

Рассмотрим Promela-программу, приведенную в листинге 43. Можно заметить, что в данном случае ветвление заменяется на true, несмотря на то, что в исходной программе было выражение a > 0. Происходит это потому что при генерации модели каждый вызов ФК обрабатывается отдельно и переданные в

них константы заменяются во всех встреченных ФД и иных использованиях (в if, while, for и т.д.).

Поскольку любая проверка в if будет возвращать истину, повышая тем самым соответствующий счетчик, то это нарушит условие того, что false_count_cond0 станет больше нуля.

3.3 Автоматическая верификация LuNA-программы

Система LuNA содержит стандартный парсер (***parser***), который используется для формирования AST в формате JSON, который используется компилятором. Поскольку верификация на моделях имеет смысл только для синтаксически корректных программ, то имеет смысл использовать готовое решение. При помощи него на первом этапе происходит трансляция LuNA-программы в его внутреннее представление.

Рассмотрим AST LuNA-программы из *листинга 41:*

```
{
 "main": {
  "type": "struct",
  "body": [
   { "type": "dfs", "names": ["x"] },
   { "type": "exec", "code": "init", "args": [{ "ref": ["x"] }, 1] },
   { "type": "exec", "code": "foo", "args": [{ "ref": ["x"] }, 1] },
   { "type": "exec", "code": "foo", "args": [{ "ref": ["x"] }, 2] }
  ]
 },
 "foo": {
  "type": "struct",
  "args": ["x", "a"],
  "body": [
   {
    "type": "if",
    "cond": { "type": "gt", "left": { "ref": ["a"] }, "right": 0 },
```

```
      "body": [
        { "type": "exec", "code": "print", "args": [{ "ref": ["x"] }] }
      ]
    }
  ]
 }
}
```

Для трансляции в Promela-программу из AST LuNA-программы воспользуемся языком TypeScript, который хорошо подходит для работой обработки JSON. Стоит отметить, что на этапе генерации модели, формируется список только тех потенциальных ошибок, которые имеет смысл проверять. Это означает, что их поиск не будет осуществляться, если нет свидетельств того, что ошибка может возникнуть. Также это относится к тем ошибкам, которые невозможно обнаружить во время верификации.

После того, как модель будет построена, она будет перенесена в Promela-файл (см. приложение А), который можно запустить системой SPIN.

Если в процессе верификации модели обнаруживается нарушение свойства, то это свидетельствует об ошибке. Она фиксируется в формате JSON, содержащем всю необходимую информацию об ошибке.

3.4 Интеграция в ADAPT

ADAPT – это комплекс для автоматизированной отладки LuNA-программ. Он объединяет несколько разных подходов к анализу, которые работают вместе, благодаря унифицированному JSON-формату ошибок. Каждый анализатор сохраняет найденные ошибки в файл. Итоговый JSON преобразуется в подробный отчет, понятный человеку. Схема работы ADAPT представлена на рисунке 3.

Каждая ошибка представлена в виде отдельного JSON-объекта в общем массиве. Объект содержит код ошибки “error_code” и поле “details”, структура которого зависит от типа ошибки, но обязательно содержит информацию, необходимую для локализации ошибки.

Пример элемента такого массива для ошибки SEM6:

```
{ "error_code":"SEM6" // код ошибки
  "details":{                  // описание ошибки
     "condition":"a > 0",  // условное выражение в if
     "type":"true",            // всегда истина
     "where":{                 // конкретная строка в файле, где найдена ошибка
        "file":"/tmp/main.fa",    // путь к исполняемому файлу
        "line":"5",                  // номер строки в этом файле
        "name":"if" }}}           //  ошибка в конструкции if
```

Вывод ADAPT представляет собой краткое описание обнаруженных ошибок. Пример вывода представлен в листинге 44.

Листинг 44 – Пример вывода системы ADAPT для ошибки SEM6

```
(1) warning[SEM6]: Condition a > 0 is always true.
In:
  File "/tmp/main.fa", line 5, in if
```

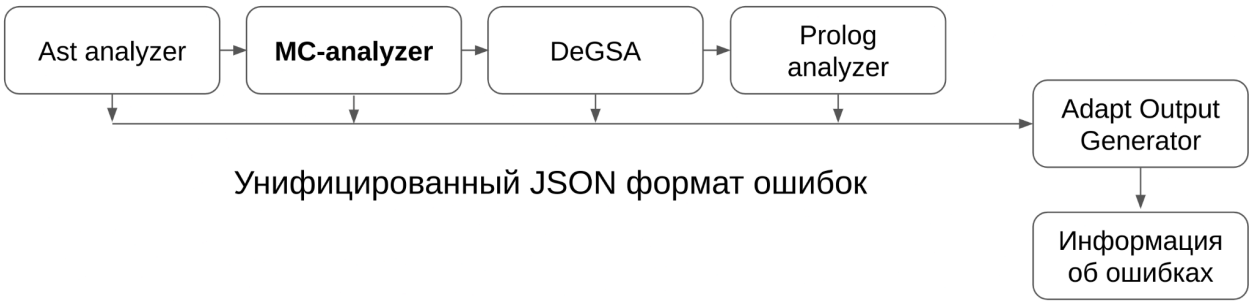

Рисунок 3 – Схема работы ADAPT

На данный момент в состав ADAPT входят следующие анализаторы:

- Ast analyzer – строит и анализирует программу по абстрактному синтаксическому дереву (AST) [24].
- **MC-analyzer** – использует метод верификации на модели для поиска семантических ошибок в программе.
- DeGSA – занимается поиском ошибок на основе графа информационных зависимостей [25].

- Prolog analyzer – анализирует логическое представление программы описанное на языке Prolog [26].

Разные анализаторы хорошо справляются с разными типами ошибок. Такой гибкий подход делает автоматическую отладку более надежной, поскольку то, что не заметит один анализатор, может обнаружить другой.

3.5 Тестирование

Для оценки производительности ключевых этапов анализа LuNA-программы проведено нагрузочное тестирование. Для генерации тестовых программ используется вспомогательный скрипт на *Python*, записанный в файл ***test.py***, приведенный в листинге 45.

Листинг 45 – Скрипт на Python для генерации LuNA-программ

```
print(
    "import c_init_int(int, name) as init;",
    "import c_print_int(int) as print;",
    "sub main() {",
    "    df a;",
    "".join([f'print(a[{i}]);' for i in
range(int(input()))]),
    "}",
    sep="\n"
)
```

Он формирует код LuNA-программы, содержащий заданное число вызовов *print(a[{i}])*, которое задается через стандартный ввод. Добавление одного такого фрагмента приведет к появлению одного LTL-свойства, проверяющего отсутствие потенциальной ошибки SEM3_1 (наличие инициализации используемого ФД). Это увеличит количество запусков верификаций модели на одно. Тем самым можно будет узнать зависимость времени верификации системой SPIN от количества LTL-свойств. Пример для N=3 приведен в листинге 46. Являясь входным параметром для файла test.py, N задает количество вызовов атомарных ФК print, каждый из которых будет

вызван с новым индексированным ФД a[i], где i начинается с нуля и заканчивается переданным N.

Листинг 46 – Пример работы скрипта для генерации LuNA-программы

```
import c_init_int(int, name) as init;
import c_print_int(int) as print;
sub main() {
    df a;
    print(a[0]);
    print(a[1]);
    print(a[2]);
}
```

Проведем серию запусков при помощи приведенного в листинге 47 Bash-скрипта.

Листинг 47 – Скрипт на bash для запуска серии экспериментов

```
for N in {1..100..5}
  do for k in {1..10}; do
    echo $N | python3 test.py > /tmp/main.fa && echo $N:$k
    ts-node main.ts /tmp/main.fa
  done
done
```

Для каждого значения N выполнялось по 10 независимых запусков из которых выбиралось минимальное время. Тестирование было проведено на компьютере с процессором AMD Ryzen 7 7730U CPU @ 2GHz 4.50 GHz, с 8 ядрами и 64 Гб оперативной памяти. Результаты запусков приведены на таблице 2 и рисунке 4. Исследовались следующие показатели времени:

- Генерация Promela-программы – время, которое понадобилось для исследования AST LuNA-программы и получения необходимой информации для формирования ее модели и LTL-свойств.
- Компиляция Promela-программы – время, прошедшее с начала записи модели в файл и до момента, когда она будет скомпилирована в запускаемый файл, который можно верифицировать.
- Проверка всех свойств – время проверки всех LTL-свойств.

Таблица 2 – Время работы в зависимости от N (количества LTL-свойств)

N	Генерация Promela-программы, мс	Компиляция Promela-программы, мс	Проверка всех свойств, мс
1	11.338	242	64
11	14.168	294	377
21	15.464	355	753
31	16.507	423	1217
41	18.202	479	1987
51	20.133	540	2254
61	24.919	631	3087
71	26.52	689	3648
81	30.022	765	4452
91	30.006	841	5934

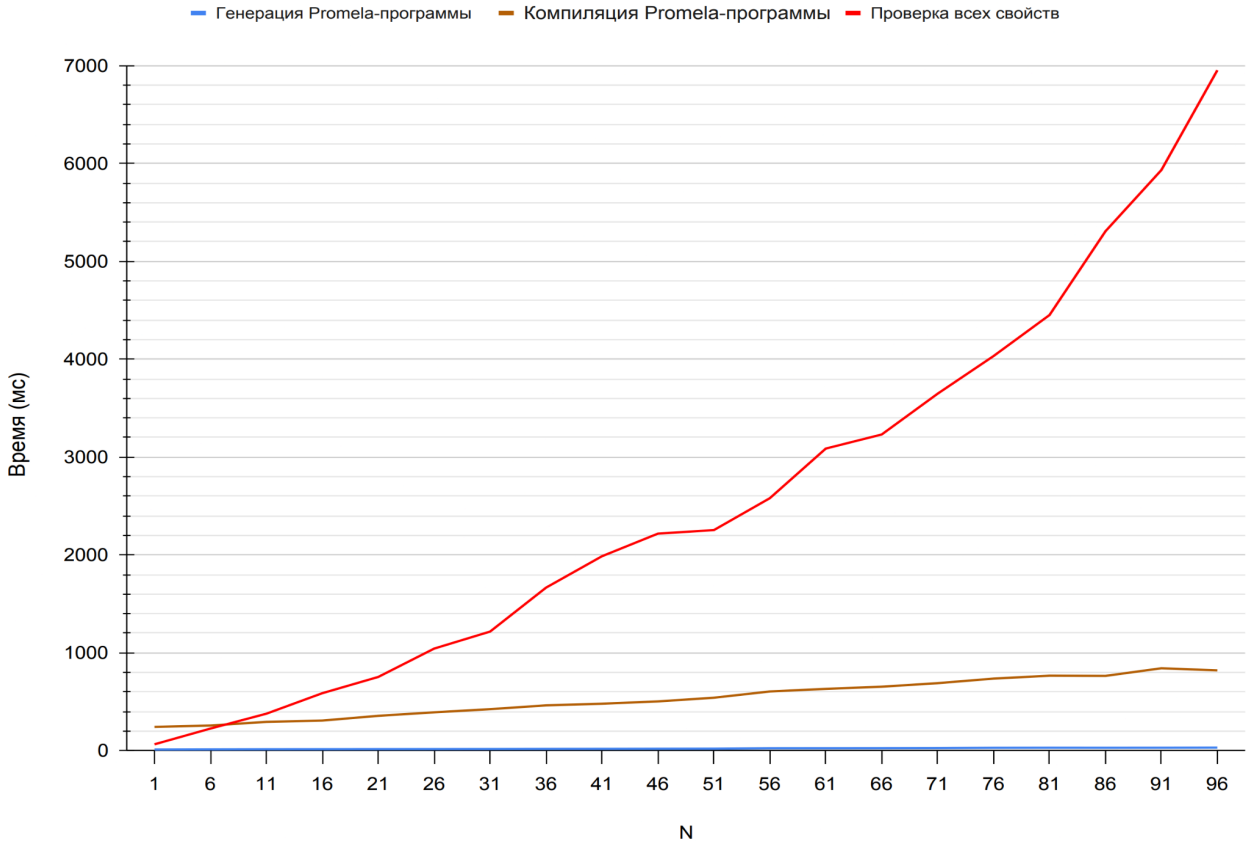

Рисунок 4 – Время работы анализатора в зависимости от N

Можно заметить, что основное время занимает непосредственно запуск и проверка модели системой SPIN. Существенным преимуществом предлагаемой методики является возможность формулировки набора независимых LTL-свойств, каждое из которых проверяется относительно одной Promela-модели. Это позволит в дальнейшем реализовать стратегию параллельного выполнения верификации, поскольку LTL-свойства независимы.

Общая трудоемкость анализа имеет тенденцию к росту по мере увеличения сложности исходной LuNA-программы. Несмотря на то, что параллельная проверка независимых LTL-свойств может существенно ускорить процесс анализа, данная оптимизация не устраняет ограничения, связанные с ростом пространства состояний модели.

ЗАКЛЮЧЕНИЕ

В ходе выполнения ВКР было произведено ознакомление с системой и языком программирования LuNA. Были выявлены и проанализированы характерные ошибки, свойственные LuNA-программам. Был произведен обзор современных средств статического анализа программ. Был спроектирован, реализован и протестирован анализатор LuNA-программ на основе метода верификации на моделях. Для обеспечения совместимости с комплексом автоматизированной отладки ADAPT был использован унифицированный формат ошибок. В течение работы над ВКР была опубликована научная статья [27]. На МНСК-2025 были изложены результаты работы.

Выпускная квалификационная работа выполнена мной самостоятельно и с соблюдением правил профессиональной этики. Все использованные в работе материалы и заимствованные принципиальные положения (концепции) из опубликованной научной литературы и других источников имеют ссылки на них. Я несу ответственность за приведенные данные и сделанные выводы.

Я ознакомлен с программой государственной итоговой аттестации, согласно которой обнаружение плагиата, фальсификации данных и ложного цитирования является основанием для не допуска к защите выпускной квалификационной работы и выставления оценки «неудовлетворительно».

_____ _____

ФИО студента *Подпись студента*

« ____ »_____20 __г.

(заполняется от руки)

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Перепелкин В.А. Система LuNA автоматического конструирования параллельных программ численного моделирования на мультикомпьютерах // Проблемы информатики. – 2020. – № 1 (46). – С. 66–74.
2. Malyshkin V.E., Perepelkin V.A. LuNA Fragmented Programming System, Main Functions and Peculiarities of Run-Time Subsystem // Parallel Computing Technologies. – 2011. – Vol. 6873. – P. 53–61.
3. Миронов А.М. Верификация программ методом model checking. – М. : Изд-во МГУ, 2012. – 74 с.
4. Малышкин В.Э. Технология фрагментированного программирования // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. – 2012. – № 46 (305). – С. 45–55.
5. Киреев С.Е., Литвинов В.С. Анализ исполнения фрагментированных программ на основе факторов SLOW // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. – 2024. – Т. 13, № 2. – С. 77–96. – DOI: https://doi.org/10.14529/cmse240205.
6. Власенко А.Ю., Мичуров М.А., Царёв В.Д., Курбатов М.А. Построение комплекса автоматизированной отладки фрагментированных программ // Вестник НГУ. Серия: Информационные технологии. – 2024. – Т. 22, № 1. – С. 5–20. – DOI: https://doi.org/10.25205/1818-7900-2024-22-1-5-20.
7. Маркова В.П., Остапкевич М.Б. Сравнение возможностей MPI и LuNA на примере реализации модели клеточно-автоматной интерференции волн // Проблемы информатики. – 2017. – № 2 (35). – С. 53-64.
8. Красикова И.Е., Картузов В.В., Красиков И.В. Применение средств распараллеливания вычислений в реализации алгоритма вычисления фрактальной размерности двумерных изображений // International Journal of Open Information Technologies. – 2015. – № 12. – С. 7–11.

9. Афанасьев К.Е., Власенко А.Ю. Семантические ошибки в параллельных программах для систем с распределенной памятью и методы их обнаружения современными средствами отладки // Вестник Кемеровского государственного университета. – 2009. – № 2. – С. 13–20.
10. База ошибок [Электронный ресурс] // GitHub. – URL: https://github.com/LuNA-Static-Analysis/LuNA-Static-Analysis-Repository/wiki/База-ошибок (дата обращения: 19.05.2025).
11. Катаев Н.А., Смирнов А.А., Жуков А.Д. Динамический анализ зависимостей по данным в системе SAPFOR // Научный сервис в сети Интернет. – 2019. – № 21. – С. 400–412.
12. Белеванцев А.А. Многоуровневый статический анализ исходного кода программ для обеспечения качества программ // Программирование. – 2017. – Т. 43, № 6. – С. 3–26.
13. Галиев Р.М., Евдошенко О.И. Обзор существующих решений для статического анализа кода Golang // Инженерно-строительный вестник Прикаспия. – 2024. – № 2 (48). – С. 73–76.
14. Ицыксон В.М., Глухих М.И., Зозуля А.В., Власовских А.С. Исследование средств построения моделей исходного кода программ на языках C и C++ // Информатика, телекоммуникации и управление. – 2009. – № 1 (72). – С. 122–129.
15. Красновидов А.В., Логинов П.А. Об определении названия функции по абстрактному синтаксическому дереву с помощью нейронной сети // Интеллектуальные технологии на транспорте. – 2020. – № 2 (22). – С. 36–45.
16. Касьянов В.Н., Евстигнеев В.А. Графы в программировании: обработка, визуализация и применение. – СПб. : БХВ-Петербург, 2003. – 1104 с.
17. Allen F. Control flow analysis // ACM SIGPLAN Notices. – 1970. – Vol. 5, No. 7. – P. 1–19.

18. Лукин М.А., Шалыто А.А. Верификация автоматных программ с использованием верификатора SPIN // Научно-технический вестник информационных технологий, механики и оптики. – 2008. – № 53. – С. 145–162.
19. Кларк Э.М., Грамберг О., Пелед Д. Верификация моделей программ : Model Checking. – 2002. – №1 (416). – С. 64-79.
20. Кораблин Ю.П., Косакян М.Л. Анализ моделей программ на языке асинхронных функциональных схем средствами темпоральной логики линейного времени // Программные продукты и системы. 2014. №2 (106). С. 5-10.
21. Вельдер С.Э., Шалыто А.А. Методы верификации моделей автоматных программ // Научно-технический вестник информационных технологий, механики и оптики. – 2008. – № 53. – С. 123–137.
22. Delzanno G., Tatarek M., Traverso R. Model Checking Paxos in SPIN // Proceedings of the Fifth International Symposium on Games, Automata, Logics and Formal Verification (GandALF 2014). – 2014. – Vol. 161. – P. 131–146. – DOI: https://doi.org/10.4204/EPTCS.161.13.
23. Newcombe C., Rath T., Zhang F., Munteanu B., Brooker M., Deardeuff M. How Amazon Web Services Uses Formal Methods // Communications of the ACM. – 2015. – Vol. 58, No. 4. – P. 66–73. – DOI: https://doi.org/10.1145/2699417.
24. Zubov M.V., Pustygin A.N., Starcev E.V. The Use of Universal Intermediate Representations for Static Analysis of the Source Code // Reports of Tomsk State University of Control Systems and Radioelectronics. – 2013. – No. 1 (27). – P. 64–68.
25. Kuck D.J., Kuhn R. H., Padua D.A., Leasure B., Wolfe M. Dependence Graphs and Compiler Optimizations // Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). – 1981. – P. 207–218. – DOI: https://doi.org/10.1145/567532.567555.

26. Мичуров М.А. Разработка инструмента статического анализа фрагментированных программ с использованием Prolog // Параллельные вычислительные технологии : XVIII Всероссийская конференция с международным участием, ПаВТ'2024, г. Челябинск, 2–4 апреля 2024 г. – Челябинск : Изд. центр ЮУрГУ, 2024. – С. 188.
27. Усенко Н.С., Власенко А.Ю. Верификация на моделях фрагментированных программ в системе LuNA // Современное программирование: сб. ст. по материалам Междунар. науч.-практ. конф. (Нижневартовск, 2025). – Нижневартовск : НВГУ, 2025. – С. 398-406.

ПРИЛОЖЕНИЕ А

Promela-модель для поиска нарушений свойств системой SPIN

```
#define CONCAT3(a, b, c) a##b##c
#define def(name) isdef_##name = true; true
#define destroy(name) destroy_count_##name = destroy_count_##name + 1
#define enddef(name) enddef_##name: isdef_##name = false;\
   use_count_##name = 0;\
   init_count_##name = 0;\
   destroy_count_##name = 0
#define init(name) init_count_##name = init_count_##name + 1
#define use(name) use_count_##name = use_count_##name + 1
#define define_check_init(is_used, enddef_label, is_init)\
   (enddef_label -> (is_used -> is_init))
#define define_check_usage(is_used, enddef_label, is_init)\
   (enddef_label -> (is_init -> is_used))
#define check_init(name)\
   define_check_init(use_count_##name > 0,\
                     main@enddef_##name,\
                     init_count_##name > 0)
#define check_usage(name)\
   define_check_usage(use_count_##name > 0,\
                      main@enddef_##name,\
                      init_count_##name > 0)
#define depends_on(varA, varB) \
   CONCAT3(depends_on_, varA, _##varB)=true
#define inc_true(name) true_count_##name = true_count_##name + 1
#define inc_false(name) false_count_##name = false_count_##name + 1
#define init_var(name) bool isdef_##name = false;\
```

```
    int init_count_##name = 0; int use_count_##name = 0;\
    int destroy_count_##name = 0;int true_count_##name = 0;\
    int false_count_##name = 0
init_var(cond0);
init_var(var0);
bool cond0 = false;
ltl SEM3_1_var0 {[] (check_init(var0))}
ltl SEM4_var0 {[] (check_usage(var0))}
ltl SEM6_cond0 { [] (
        ( (true_count_cond0 > 0) -> (<> (false_count_cond0 > 0)) ) &&
        ( (false_count_cond0 > 0) -> (<> (true_count_cond0 > 0)) )
)}
active proctype main() {
  if
  :: cond0=0
  :: cond0=1
  fi
  def(var0);
  init(var0);
  if
  :: true -> inc_true(cond0);
    use(var0);
  fi
  if
  :: true -> inc_true(cond0);
    use(var0);
  fi
  enddef(var0);
}
```

ПРИЛОЖЕНИЕ Б

Статический анализатор LuNA-программ MC-analyzer

Руководство оператора

Листов 6

АННОТАЦИЯ

В данном программном документе приведено руководство оператора по применению модуля статического анализа LuNA-программ MC-analyzer.

Настоящий программный документ содержит руководство оператора по использованию и эксплуатации средства автоматизированного обнаружения семантических ошибок – MC-analyzer.

Раздел “Назначение программы” описывает цели разработки средства, раскрывает его функциональные возможности и предоставляет информацию, необходимую для понимания принципов работы и порядка применения.

В разделе “Условия выполнения программы” изложены технические и программные требования, при соблюдении которых возможно корректное функционирование MC-analyzer.

Раздел “Выполнение программы” содержит пошаговую инструкцию для оператора, включающую действия по загрузке, запуску, использованию и корректному завершению работы программы.

Оформление программного документа “Руководство оператора” произведено по требованиям ЕСПД: 19.101-77, 19.105-78, ГОСТ 19.505-79.

СОДЕРЖАНИЕ

1 Назначение программы

1.1 Функциональное назначение программы

Функциональное назначение заключается в автоматическом обнаружении следующих семантических ошибок на основе исходного кода LuNA-программы:

- повторная инициализация ФД;
- неинициализированный ФД используется вне цикла;
- циклическая зависимость по данным;
- использование ФД после его удаления;
- неиспользуемый ФД;
- формула в if/while тождественно истинна/ложна;
- формула в if/while истинна/ложна во всех путях выполнения.

1.2 Эксплуатационное назначение программы

Программа может выявлять ошибки в логике LuNA-программы на этапе разработки.

1.3 Требования к составу выполняемых функций

По синтаксически корректной исходной программе MC-analyzer должен создавать модель и производить на ней верификацию.

При обнаружении ошибки пользователь должен получить подробную информацию о контексте ее потенциального возникновения в исходной программе. Наполнение информации зависит от класса найденной ошибки.

MC-analyzer должен находить места потенциального возникновения ошибок, должен работать с циклами for, while, с условиями if, с вызовами атомарных и структурированных фрагментов кода (ФК).

2 Условия выполнения программы

2.1 Требования к составу и параметрам технических средств

Минимальная конфигурация технических средств:

- объем ОЗУ: 2 Гб;
- свободное место на диске: 1 Гб;

- процессор: Intel Core i3-3210 / AMD A8-7600 APU.

2.2 Минимальный состав программных средств

- Операционная система семейства Linux;
- компилятор языка C (GCC 12);
- Bash;
- Node.js 16+;
- SPIN.

2.3 Необходимое количество и квалификация персонала

Для эксплуатации требуется один человек, имеющий навыки:

- разработки на языке LuNA;
- работы с командной строкой Linux (в т.ч. умение запускать команды и интерпретировать текстовый вывод).

3 Выполнение программы

3.1 Загрузка и запуск программы

Запуск программы осуществляется вызовом команды adapt с указанием пути к анализируемому файлу, который должен содержать текст программы на языке LuNA.

Пользователь может указать следующие опциональные параметры:

- вывести информацию об использовании: --help;
- не очищать сгенерированные файлы: --no-cleanup;
- запустить конкретный анализатор: --run [ast|degsa|prolog|mc].

  ast: AST-analyzer – анализ по абстрактному синтаксическому дереву.

  degsa: DeGSA – анализ по графу зависимостей по данным.

  prolog: Prolog-analyzer – анализ логического представления программы.

  mc: MC-analyzer – верификация на моделях.

3.2 Выполнение программы

После того, как ADAPT проанализирует указанную LuNA-программу, в стандартный поток вывода будет отправлена информация обо всех найденных ошибках. В листинге Б.1 приведен пример вывода.

Листинг Б.1 – пример сообщения об ошибке средством ADAPT

```
(2) error[SEM4]: Unused DF a.
  File "/tmp/main.fa", line 12, in main
```

3.3 Завершение программы

Программа может завершиться автоматически после того, как все указанные анализаторы закончат работу и распечатается результат проверки. Также Программу можно завершить досрочно отправив сигнал SIGINT. Сделать это можно нажав комбинацию клавиш: *Ctrl+C*.