На правах рукописи

Прокопенко Артем Сергеевич

СТАТИЧЕСКИЙ АНАЛИЗ УСЛОВИЙ ГОНКИ В ПАРАЛЛЕЛЬНЫХ ПРОГРАММАХ НА РАЗДЕЛЯЕМОЙ ПАМЯТИ

Специальность 05.13.18 – математическое моделирование, численные методы и комплексы программ

Автореферат диссертации на соискание
ученой степени кандидата физико-математических наук

Москва - 2010

Работа выполнена на кафедре информатики
Московского физико-технического института
(государственного университета)

Научный руководитель:
доктор физико-математических наук, доцент
ТОРМАСОВ Александр Геннадьевич

Официальные оппоненты:
доктор физико-математических наук, профессор
ДИКУСАР Василий Васильевич

кандидат физико-математических наук, доцент
ПРОХОРОВ Сергей Петрович

Ведущая организация:
Институт автоматизации проектирования РАН

Защита состоится __ декабря 2010 года в _____ часов на заседании Диссертационного совета Д 212.156.05 при Московском физико-техническом институте (государственном университете) по адресу: 141700, Московская обл., г. Долгопрудный, Институтский пер., д. 9, ауд. 903 КПМ.

С диссертацией можно ознакомиться в библиотеке Московского физико-технического института (государственного университета).

Автореферат разослан 23 ноября 2010 года.

Ученый секретарь диссертационного совета Д 212.156.05 Федько О.С.

ОБЩАЯ ХАРАКТЕРИСТИКА РАБОТЫ

Актуальность темы

Разработка и повсеместное внедрение многопроцессорных систем потребовали массового перехода к написанию параллельных программ - программ, состоящих из нескольких потоков и взаимодействующих друг с другом посредством общей памяти. Но разработка многопоточных программ сложнее последовательных. Это связано с тем, что порядок доступа к данным, в котором будут выполнены инструкции разных потоков, заранее непредсказуем, и разработчик должен предусмотреть корректную работу программы при всех возможных чередованиях инструкций.

Для упорядочивания доступа к общим данным программисты обращаются как к классическим методам синхронизации (взаимоисключения, светофоры и т.д.) так и к не блокирующим структурам и алгоритмам (non-blocking structures). Однако применение тех или иных механизмов синхронизации не является гарантией того, что не возникнет состояния неразрешенной гонки – ошибки многопоточного программирования, при которой выполнение программы нарушает ту часть спецификации (набор формализованных утверждений, описывающих требования к программе), которая касается доступа к общим данным. Поэтому постоянно необходимо проверять правильность функционирования программы.

В целом проверка соответствия между требованиями к системе и свойствами работающей системы (верификация) является нетривиальной задачей. Различают два принципиально разных подхода – динамический и статический. Динамическая верификация представляет собой эмпирическую процедуру «прогона» программы при различных входных данных. Проверить сложную систему тестированием обычно невозможно, поскольку количество тестов экспоненциально зависит от размера входа, да и само число различных комбинаций входных данных в программу может быть в целом не ограничено. В итоге, несмотря на применение различных техник, для отчетов динамических

анализаторов свойственно большое количество ложных заключений о правильности программ.

Проверка на основе моделей (model checking) и дедуктивная верификация, относящиеся к статическим формальным методам, реализуют альтернативный динамическому статический подход. Главными проблемами верификации на основе моделей являются сложность построения модели параллельных программ и «комбинаторный взрыв» в пространстве состояний системы – значительный рост объема проверок даже при небольшом усложнении комплекса программ. В этом отношении выгодно отличается дедуктивная верификация, когда посредством формальной логики устанавливается соответствие результата выполнения программы представленной спецификации. Метод R/G – Rely/Guarantee (Jones,1983) зарекомендовал себя как качественный метод верификации параллельных программ по заданным критериям. В этом методе для каждого потока выполнение параллельной программы представлено двумя отношениями: Rely и Guarantee, первое описывает изменение состояний, выполненное потоком, второе описывает результат всех других потоков - единой внешней среды (формальное описание метода Rely/Guarantee будет дано ниже). Однако, применение R/G - метода осложняется тем, что модель состояний представляется одним пулом памяти (нет разделения на локальные и общие состояния потоков). В результате необходимо целиком специфицировать состояния системы после каждого шага программы. Кроме того, метод не подходит для верификации свойства живучести (liveness properties) - т.е. свойства программ, которые утверждают, что нечто изначально задуманное произойдет при любом развитии внешних событий и любом ходе работы программы. Варианты частичного решения проблемы единого пула памяти предложены в методах RGSep (Vafeiadis, 2007) и SARG (Feng, 2007), в которых память разделена на локальную память каждого потока и общую для всех. Это позволяет не специфицировать абсолютно все локальные состояния выполнения программы. Сами методы являются синтезом R/G-метода и сепарационной логики (separation logic).

Также предложен способ использования метода RGSep для верификации свойств живучести (Gostman, 2009). Однако, модель в методах R/G, RGSep и подобных обладает следующими недостатками:

- несколько потоков могут иметь совместно используемые ячейки памяти, скрытые от других потоков. В рамках методов R/G, RGSep, SARG нет возможности скрыть эти ячейки от других потоков, что может привести к ложному отчету о корректности программы;
- необходимо специфицировать все общие ячейки, даже если доступ осуществляется только к части из них, что ограничивает применимость уже верифицированных частей программ в другой среде.

Следует отметит, что синтаксис программ в RGSep включает «крупные» команды, многие из которых могут быть представлены несколькими операциями доступа к памяти (чтение/запись).

Цель работы, задачи исследования

Целью диссертационной работы является создание метода статического анализа условий гонок многопоточных алгоритмов, основанного на R/G-подходе, для верификации параллельных программ, в которых некоторые потоки имеют совместно используемые области памяти, скрытые от других потоков, а также проведение вычислительных экспериментов для исследования прикладных аспектов этого метода.

Задачи исследования:

1. математическое моделирование многопоточных алгоритмов, работающих на разделяемой памяти;
2. разработка метода задания формулы частичной корректности, специфицирующей состояние гонки;
3. разработка метода анализа наличия гонок, используя расширение аксиоматического метода верификации Rely/Guarantee и сепарационной логики;
4. оценка сложности алгоритма анализа гонок в параллельных программах на общей памяти в различных случаях;

5. использование предложенного подхода и соответствующих вычислительных экспериментов для выявления состояния гонки в тестовых параллельных программах на разделяемой памяти;

6. создание комплекса программ для автоматизации выявления совместно используемых потоками данных, декомпозиции «крупных» команд на атомарные инструкции.

Научная новизна

Научная новизна работы заключается в предложенной методике формализации и анализа состояния гонки в параллельных программах на разделяемой памяти, базирующейся на аксиоматическом методе верификации R/G и сепарационной логике. По сравнению с известными формальными методами верификации Rely/Guarantee и RGSep данный подход обладает следующими достоинствами:

- Позволяет проводить анализ гонок в параллельных программах на разделяемой памяти, в которых некоторые потоки имеют области памяти, скрытые от других.
- Дает возможность не определять значения всех состояний системы после каждого шага.

Практическая ценность

Результаты исследования могут быть использованы на практике для определения наличия состояний гонки, в том числе, для анализа неблокирующихся реализаций различных алгоритмов. Предложенный метод верификации может быть автоматизирован в средах Isabella/HOL, SmallfootRG.

Положения, выносимые на защиту:

1. Математическая модель процесса выполнения параллельных программ на разделяемой памяти, в которых некоторые потоки имеют области общей памяти, скрытые от других.

2. Метод задания формулы корректности, определяющей требования к выполнению программы, при соблюдении которых считается, что неразрешенное состояние гонки не возникло.

3. Метод анализа гонок в параллельных программах на разделяемой памяти, использующий расширение аксиоматического метода верификации Rely/Guarantee и сепарационной логики.

4. Вычислительные эксперименты для комплексного исследования наличия гонок в параллельных программах на общей памяти на основе предложенной методики с использование разработанного комплекса программ.

Публикации и апробация результатов

По теме диссертации опубликовано 11 работ, в том числе две [1, 4] – в изданиях, рекомендованных ВАК РФ.

Результаты диссертационного исследования докладывались, обсуждались и получили одобрение специалистов на научных семинарах и конференциях:

- IX международная научно-практическая конференция «Исследование, разработка и применение высоких технологий в промышленности» (Санкт-Петербург, 2010 г.);
- Международная научно-техническая конференция "Многопроцессорные вычислительные и управляющие системы 2009" (Дивноморское, 2009 г.);
- Седьмая международная научная молодежная школа «Высокопроизводительные вычислительные системы ВПВС-2010», (Дивноморское, 2010 г.);
- XXXV и XXXVI международные молодежные научные конференции «Гагаринские чтения», (Москва, 2009, 2010 гг.);
- XLIX и L научные конференции МФТИ, (Москва, 2006, 2007 гг.);

- Научные семинары кафедры информатики МФТИ (Москва, 2008-2010 гг.).

Структура и объем работы

Диссертация состоит из введения, шести глав, заключения и списка использованных источников, включающего 105 наименований. Общий объем работы составляет 107 страниц текста.

СОДЕРЖАНИЕ РАБОТЫ

Во **Введении** обосновывается актуальность диссертации и формулируются ее цели, характеризуются научная новизна и практическая ценность работы.

В **главе 1** приводятся базовые понятия, используемые в работе, представлен краткий обзор современных подходов к верификации.

В **разделе 1.1** приведено базовое описание параллельных программ на разделяемой памяти.

В **разделе 1.2** введено понятие гонки данных(data race), обобщенной гонки(general race). Обозначены современные подходы к синхронизации потоков: классические методы синхронизации (взаимоисключения, семафоры и т.д.), неявные методы синхронизации(non-blocking structures), программная транзакционная память(software transactional memory). Приведена известная теорема о консенсусе.

В **разделе 1.3** приведен аналитический анализ современных подходов к проверке правильности программ. Упомянуты основные подходы: статический анализ (а именно, формальные методы верификации - дедуктивный анализ, методы проверки на основе моделей) и динамический анализ. Формальные методы требуют построения модели системы, в которых формально представлены состояния системы и переходы из одного состояния в другое. Сделан обзор исполнительных, ограничительных и аксиоматических моделей.

Глава 2 посвящена предложенной модели параллельных программ на общей памяти. В ней также описана сепарационная логика и введенная автором операционная семантика используемого языка программирования типа Си.

В **разделе 2.1** приведено описание построенной модели. При самом моделировании параллельной программы использовалась широко распространенная предпосылка – чередование (мультипрограммность, interleaving). Она позволяет значительно упростить анализ реального параллельного вычисления, которое выполняется процессорами машины, и в тоже время адекватно его представить. Чередование позволяет представить выполнение параллельной программы, как последовательность дискретных шагов. Для того чтобы в таком представлении были учтены всевозможные сценарии исполнения программы, и при этом требование адекватного представления программы не нарушалось, необходимо чтобы инструкции были максимально атомарны. Отмечено, что на платформе архитектуры x86 гарантируется атомарность операции чтения, записи, и инструкции, реализующие примитив CAS(compare and swap). Сам выбор архитектуры x86 обусловлен ее широким распространением. Также нужно отметить, что в процессе оптимизации кода компилятор может переставлять фрагменты его местами для повышения производительности.

Пусть программа - это четверка $\langle M_S, s_0, S, P^s \rangle$, где M_S - множество общих ячеек памяти, $s_0 \in S$ - начальное состояние, *S*-множество общих состояний, P^s- конечный набор подпрограмм операций. Состояние – это отображение ячеек по множество значений.

Каждая подпрограмма P^s – это семерка $\langle M_L, M_{LS}, l_0, L, v, T, <\textit{имя подпрограммы}> \rangle$, где M_L - множество локальных ячеек памяти, $M_{SL} \subseteq M_S$ - множество общих ячеек, доступных для подпрограммы ячеек, l_0 - начальное локальное состояние, L - локальные состояния, $T = \{t_1, .. t_z\}$ - последовательность ассемблерных инструкций (команд) t_j на архитектуре x86,

$v : t_j \to \Sigma_{метки}$ - метки, $\Sigma_{метки} = \{q_0, ..., q_{фин}\}$. Локальные состояния подпрограммы не доступны для других подпрограмм.

Для того чтобы, исполнить программу, надо задать потоки пользователя $\Psi = \varphi_1, ..., \varphi_n$. Каждый поток φ_i представляет собой последовательность подпрограмм $p_1^s, ..., p_{n_j}^s$. Поток стартует в начальном состоянии подпрограммы p_1^s, когда заканчивает ее выполнение, переходит в начальное состояние следующей подпрограммы и так далее.

Для заданных пользовательских потоков Ψ состояние выполнения характеризуется текущими значениями программных переменных $\sigma=(s,l^1,...,l^n)$, где s - разделяемое состояние, l^i – локальное состояние потока φ_i и текущими метками v^j всех потоков. Начальное состояние параллельной программы обозначим $\sigma_0 = (s_0, l_0^1, ...l_0^n)$. Множество состояний выполнения обозначим *St*. Само выполнение параллельной программы определяется как последовательность программных состояний $\sigma_0,\sigma_1,\sigma_2,\ldots$, где каждое последующее состояние получено из предшествующего в результате соответствующего перехода (представляющее выполнение команды) одного из параллельных потоков: переход $\sigma \xrightarrow{\mu(\alpha)} \sigma'$, $\sigma=(s,l^1,...l^\alpha,...l^n)$, где $\mu(\alpha)$ обозначает, что выполняется операция с меткой μ в потоке $\alpha \in \Psi$.

Трасса выполнения программы - последовательность $\mu_1(\alpha),\mu_2(\alpha),\ldots\mu_N(\alpha)$ такая, что $\sigma_0 \xrightarrow{\mu_1(\alpha_1)} \sigma_1 \xrightarrow{\mu_2(\alpha_2)} .... \xrightarrow{\mu_N(\alpha_N)} \sigma_N$.

Раздел 2.2 посвящен описанию базового языка программирования типа Си (определен синтаксис команд, описана модель программных состояний, операционная семантика, сепарационная логика).

С той целью чтобы инструкции потоков были максимально атомарны, при этом верификация программ не была чрезмерно затруднена, в качестве команд в модели используются ассемблерные инструкции на архитектуре x86. Но чтобы облегчить применимость предложенного метода верификации по заданной спецификации делается следующее: «укрупняется» входной алфавит

инструкции до языка типа Си, а в самом методе вносятся соответствующие поправки.

Определен синтаксис команд программ следующим образом:

- выражения: $\Phi ::= \Phi \,||\, \Phi \mid P; P \mid P\,;\, P ::= C \mid E \mid B\,;$
- команды: $C ::= skip \mid x := E \mid x := [E] \mid [E] := E \mid C; C \mid if\ B\ then\ \{C\}\ else\ \{C\} \mid$

$\mid while\ B\ do\ \{C\} \mid x := new() \mid delete(E)$,

- целочисленные выражения: $E ::= x \mid n \mid E\ iop\ E\,;$
- логические выражения: $B ::= b \mid E\ bop\ E\,.$

Вспомогательные определения: $n \in Z$ - множество целых чисел; $b \in B = \{true, false\}$ - множество логических значений; $iop \in Iop = \{+,-,[],...\}$ - постоянное конечное множество целочисленных бинарных операций; $bop \in Bop = \{=,<,>,\ldots\}$ - постоянное конечное множество логических бинарных операций. Запись [x] означает содержание ячейки по адресу x. Оператор skip (пустой оператор) не меняет состояния программы.

Определено базовое понятие операционной семантики: состояние. В классических работах оно определяется как отображение программных переменных во множество их значений. Для того чтобы работать с программами, реализованными с помощью динамически выделяемых структур в работе используется сепарационная логика (separation logic). В этой логике множество программных состояний Σ задаётся, например, следующим образом:

$$Values = \{...,-2,-1,0,1,...\};\quad Vars = \{x, y,...,\&x,\&y,...\};\quad Locations = \{1,2,....\}$$

$$Heaps = Locations \rightarrow_{fin} Values;\quad Stores = Vars \rightarrow Values;\quad \Sigma = Stores \times Heaps$$

Поскольку в предложенной модели состояния могут быть одного из двух типов: локальные $l \in \Sigma$ или общие $s \in \Sigma$, то программные состояния – это пары состояний (s,l) $\in \Sigma^2$.

Для привязки состояния к конкретной точке программы используется понятие *конфигурации*. Это пара <*S, σ*> — программный фрагмент *S* и программное состояние *σ*. После определены аксиомы и правила вывода

операционной семантики, описывающие бинарное отношение перехода во множестве конфигурации.

В сепарационной логике программные состояния характеризуются через утверждения о состоянии памяти (формула состояния памяти, ФСП).

Пусть Σ - пространство программных состояний, B - булево множество {0,1}, тогда *утверждением p* над пространством состояний называем предикат вида $p = \{\alpha \mid \alpha : \Sigma \to B\}$.

Пусть *p* и q являются утверждениями в сепарационной логике:

$$p, q ::= false \mid emp \mid e = e' \mid e \to e' \mid p \Rightarrow q \mid p * q \mid p -\!\!\otimes q \mid ...$$

Здесь *emp* введено для обозначения пустой кучи (Heap), $e \to e'$ описывает состояние, в котором куча состоит из одной выделенной ячейки по адресу e_1 с содержанием e_2. *Оператор раздельной конъюнкции* (separating conjunction) * является ключевым элементом сепарационной логики. Куча *h* удовлетворяет ФСП *p*q*, если ее можно разбить на две части, одна удовлетворяет утверждению *p*, другая *q*. Определение утверждения *p*q* вводиться через операцию • над Σ такую, что для всех $(t_1,h_1),(t_2,h_2) \in \Sigma$:

$(t_1, h_1) \bullet (t_2, h_2) = (t_1, h_1 \perp h_2)$, так что если $t_1=t_2$ и $h_1 \perp h_2$ определено, иначе $(t_1,h_1) \bullet (t_2,h_2)$ не определено. И так, утверждение *p*q*: $(t,h) \models (p * q) \Leftrightarrow \exists h_1, h_2 . h_1 \perp h_2 = h \wedge (h_1 \models p) \wedge (h_2 \models q)$.

Если для всех $h_1, h_1', h_2, h_2' \subseteq h$, таких что $(h_1 \perp h_2 = h_1' \perp h_2') \wedge (t, h_1 \models p) \wedge (t, h_1' \models p)$ следует $h_1 = h_1'$, то утверждение $p = \{\alpha \mid \alpha : \Sigma \to B\}$ называется *однозначным утверждением precise(p)*.

Поскольку в предложенной модели состояния могут быть одного из двух типов: локальные $l \in \Sigma$ или общие $s \in \Sigma$, то ФСП характеризуют сразу и общие, и локальные состояние. Конечно, можно специфицировать каждое состояние посредством двух утверждений, но вместо этого воспользуемся грамматикой утверждений из RGSep. Тогда, пусть запись $\underline{\underline{p}}$ - утверждение относительно общих частей, а *p* – локальных.

Синтаксис: $p, q ::= P \mid \underline{\underline{P}} \mid p * q \mid p \wedge q \mid p \vee q \mid p -\!\!\otimes q$.

Отношение |= выполнимости ФСП в состоянии (s,l) определяется по индукции:

$$(s,l) \models P \Leftrightarrow l \models_{SepL} P$$

$$(s,l) \models \underline{\underline{P}} \Leftrightarrow (l =) \wedge (s \models_{SepL} P)$$
$$(s,l) \models p_1 * p_2 \Leftrightarrow \exists l_1, l_2 (l = l_1 \perp l_2) \wedge (l_1, s \models_{SepL} p_1) \wedge (l_2, s \models_{SepL} p_2)$$
$$(s,l) \models p_1 \vee p_2 \Leftrightarrow (l_1, s \models_{SepL} p_1) \vee (l_2, s \models_{SepL} p_2)$$
$$(s,l) \models p_1 \wedge p_2 \Leftrightarrow (l_1, s \models_{SepL} p_1) \wedge (l_2, s \models_{SepL} p_2)$$
$$(s,l) \models p -\!\otimes q \Leftrightarrow \exists l_1, l_2 (l_2 = l_1 \perp l) \wedge (l_1, s \models_{SepL} p) \wedge (l_2, s \models_{SepL} q)$$

Здесь операторы *,-⊗ определены на Σ^2.

В главе 3 описаны метод верификации Rely/Guarantee, сделаны обзоры синтезированных методов: RGsep, SARG, LocalRG; описан предложенный автором метод анализа гонок на разделяемой памяти; введена формула корректности; доказана непротиворечивость предложенной логики.

Раздел 3.1 посвящен методу формальной верификации Rely/Guarantee: спецификация, система правил, приводиться теорема о непротиворечивости.

Суть метод формальной верификации Rely/Guarantee заключается в следующем: пусть заданы потоки $\Psi = \varphi_1, ..., \varphi_n$, для всех $\alpha \subseteq \Psi$ рассматривается выполнение программы. Другие потоки $\Psi_E = \Psi \setminus \{\alpha\}$ называются *средой* потока α. Спецификация выполнение потока α определяют четверкой *(pre,R,G,post)* где *p*-предусловие (утверждения, которые должны быть истинны в начале выполнения потока), *q*-постусловие (утверждения, которые должны быть истинны после выполнения), *G* - спецификация шагов потока α, *R* - спецификация шагов среды по изменению данных в течение выполнения потока α. Далее посредством применения правил вывода проверяется, что выполнение потока удовлетворяет заданной спецификации.

Раздел 3.2 посвящен формализации переходов системы из одних состояний в другие.

В методе, предложенном автором, Rely и Guarantee условия определяют характер изменений общих состояний. Введено понятие *действий(actions)*

$p \sim> q$, которые определяют переходы между состояниями, специфицированными утверждениями p и q:

$s_1 * s_0, s_2 * s_0 \models p \sim> q \quad \Leftrightarrow \quad s_1 \models p \wedge s_1 \models q$; $\qquad s_1, s_2 \models [p] \quad \Leftrightarrow \quad s_1 = s_2 \wedge s_1 \models p$;

$s, s' \models a * a' \quad \Leftrightarrow \quad (s = s_1 \perp s_2) \wedge (s' = s_1' \perp s_2') \wedge (s_1, s_1' \models a_1) \wedge (s_2, s_2' \models a')$;

$s_1, s_2 \models a \Rightarrow a' \quad \Leftrightarrow \quad$ *если из* $s_1, s_2 \models a$, *следует* $s_1, s_2 \models a'$.

Первая формула специально записано в расцепленном на две части виде в целях уменьшения работы при верификации, аналогично можно записать и другие выражения, если присутствует часть общего состояния, которая не затрагивается действием.

Поскольку некоторые операции меняют состояния ячеек, то и в работе введено понятие устойчивости утверждения *p* под воздействием *a*.

Утверждение p устойчиво под действием *a Stab(p,a)* тогда и только тогда, когда $\forall s, s'$ таких, что $s \models p \wedge (s, s' \models a)$ следует $s' \models p$.

Доказана **лемма 1.** $Stab(p, r -> q) \quad \Leftrightarrow \quad \models (r -\otimes p) * q \Rightarrow p$

Первая лемма утверждает что, если из состояния, удовлетворяющего утверждению *p*, вычистить часть состояния, удовлетворяющею *r* , и заменить на часть, удовлетворяющую утверждению *q*, то результат по-прежнему будет удовлетворять *p*.

Доказана **лемма 2.** $Stab(p, (R_1 \cup R_2)^*) \quad \Leftrightarrow \quad Stab(p, R_1) \wedge Stab(p, R_2)$

Вторая лемма о том, что утверждение p устойчиво под множеством действии R тогда, когда устойчиво под действием каждого.

Доказана лемма 3. $Stab(p_1, a) \wedge Stab(p_2, a) \quad \Leftrightarrow \quad Stab(p_1 \wedge p_2, a)$**;**

$Stab(p_1, a_1) \vee Stab(p_2, a_2) \quad \Leftrightarrow \quad Stab(p_1 \vee p_2, a_1 \wedge a_2)$**;**

$Stab(p, a_1) \wedge Stab(p, a_2) \quad \Leftrightarrow \quad Stab(p, a_1 \wedge a_2)$**.**

Казалось бы сепарационная конъюнкция над действиями *a*a'* позволит скомбинировать различные переходы в одну запись, однако это не так.

Пример. Пусть $L_1 \mapsto 1, a_1 = \{L_2 \mapsto 2, L_2 \mapsto 3\}$ и $L_2 \mapsto 4, a_2 = \{L_1 \mapsto 5, L_1 \mapsto 6\}$ причем $L_1 \neq L_2$, т.е. имеем $Stab(p_1, a), Stab(p_2, a)$ и $Stab(p_1 * p_2, a_1 * a_2)$ ложь.

Это вызвано тем, что в предложенной модели есть стремление снять требование спецификации всех возможных состояний. Поэтому типичного определения инварианта как утверждения истинного до и после выполнения нам недостаточно, еще необходимо требование точного инварианта I.

Говорят, что *действие a защищено точным инвариантом* $I \triangleright a$ тогда и только тогда, когда инвариант имеет место в начальном и конечном переходах, удовлетворяющих a, и является точным инвариантом precise(I): $a \Rightarrow (I \sim> I) \wedge precise(I)$.

Доказана **лемма 4**. $(Stab(p_1, a_1) \wedge Stab(p_2, a_2) \wedge p_1 \Rightarrow I) \wedge (I \triangleright a) \Leftrightarrow$
$\Leftrightarrow Stab(p_1 * p_2, a_1 * a_2)$

В разделе 3.4 представлена аксиоматическая система.

Выводимость модифицированной тройки Хоара обозначена как $R; G; I \vdash \{pre\} C \{post\}$, что означает, что если предусловия удовлетворяют *pre*, изменения общих состояний выполняются в соответствии с R, выполнение кода потока удовлетворяет G при защите точным инвариантом I, и если подпрограмма завершает работу, то постусловия удовлетворяют *post*.

В работе введено понятие сепарационной конъюнкции для спецификации $R_1 * R_2$, $G_1 * G_2$, инварианта $I_1 * I_2$; введено два новых правила вывода для скрытия части общих данных и «разбивки» локальных:

$$\frac{R; G; I \vdash \{P * \underline{Y}\} C \{Q * \underline{Y'}\} \quad Stab(\underline{M}, R) \quad I \triangleright \{R', G'\} \quad \underline{M} \Rightarrow I' * true}{R, G, I \vdash \{P * \underline{M} * \underline{Y}\} C \{Q * \underline{M} * \underline{Y'}\}} \text{ (Frame)}$$

$$\frac{R * R'; G * G'; I * I' \vdash \{p\} C \{q\} \quad I \triangleright \{R, G\}}{R, G, I \vdash \{p\} C \{q\}} \text{ (Hide)}$$

В качестве примеров приведены следующие правила, типичные для R/G методов, но записанные в рамках предложенной системы:

$$\frac{R, G, I \vdash \{pre\} C \{post\} \quad Stab(p, R \cup G))}{R, G, I \vdash \{pre * p\} C \{post * p\}} \text{ (Stab)}$$

$$\frac{R', G', I \vdash \{pre'\} C \{post'\} \quad R \subseteq R' \quad G' \subseteq G \quad pre \Rightarrow pre' \quad post' \Rightarrow post}{R, G, I \vdash \{pre\} C \{post\}} \text{ (Weaken)}$$

$$\frac{R;G;I\vert\!-\{pre_1\}C\{post\}\quad R;G;I\vert\!-\{pre_2\}C\{post\}}{R;G;I\vert\!-\{pre_1\cup pre_2\}C\{post\}}\quad\text{(Disj)}$$

$$\frac{R;G;I\vert\!-\{pre\}C\{post_1\}\quad R;G;I\vert\!-\{pre\}C\{post_2\}}{R;G;I\vert\!-\{pre\}C\{post_1\cap post_2\}}\quad\text{(Conj)}$$

Отмечено, что в правиле параллельного исполнения явно указываются утверждения о локальных и общих частях, при этом общие части должны удовлетворять точному инварианту:

$$\frac{\begin{array}{ll}R\cup G_2;G_1;I\vert\!-\{P_1*\underline{\underline{T}}\}C_1\{Q_1*\underline{\underline{T_1}}\} & I\triangleright\{R,G\}\\ R\cup G_1;G_2;I\vert\!-\{P_2*\underline{\underline{T}}\}C_2\{Q_2*\underline{\underline{T_2}}\} & T\cup T_1\cup T_2\Rightarrow I\end{array}}{R;G_1\cup G_2;I\vert\!-\{P_1*P_2*\underline{\underline{T}}\}\,C_1\parallel C_2\,\{P_1*P_2*(R_1\wedge R_2)\}}\quad\text{(Par)}$$

Еще для примера приведено правило для исполнения команд во взаимоисключающем режиме для простого случая:

$$\frac{P\sim>Q\subseteq G\quad P\cup Q\Rightarrow I\quad I\triangleright\{R,G\}\quad Sta(\{\underline{\underline{P}},\underline{\underline{Q}}\},R)\quad P_1\sim>Q_1\subseteq G}{R;G;I\vert\!-\{P_1*\underline{\underline{P}}\}atomic\{C\}\{Q_1*\underline{\underline{Q}}\}}\quad\text{(Atom)}$$

В **разделе 3.5** определяется взаимоотношение между формализованным исчислением высказываний и алгеброй высказываний, между синтаксисом и семантикой, Для этого доказывается теорема о непротиворечивости для свойства частичной корректности.

В начале дается по индукции формализованное определение выполнимости спецификации *G* на пути исполнения программы *Guar(G,C,σ,R)*. Неформально, это звучит как: пусть программный код *C* начинает выполняться из состояния σ, действия среды удовлетворяет спецификации R, тогда говорят, что выполнение потока удовлетворяет спецификации G, тогда и только тогда, программа не завершается аварийно и изменение всех общих состояний выполняется в соответствии с *G*.

Далее вводится определение истинности формулы частичной корректности: $R;G;I\models\{p\}C\{q\}\Leftrightarrow l,s\models p,Guar(G,C,(l,s),R)\,,(C,(l,s))\xrightarrow{R}*(skip,(l',s'))\,,$ $l',s'\models q$.

Доказана **лемма 5**. Если программный код C является примитивом, то возможен один из трех сценариев: $(C,\sigma)\xrightarrow{R}*(C',\sigma')\Leftrightarrow(C,\sigma)\xrightarrow{R}_e*(C,\sigma')$;

$(C,\sigma) \xrightarrow{R} * fault \Leftrightarrow \exists \sigma''.(C,\sigma) \xrightarrow{R}_e (C,\sigma'') \xrightarrow{R}_p fault$;

$(C,\sigma) \xrightarrow{R} *(skip,\sigma') \Leftrightarrow \exists \sigma''\sigma'''.(C,\sigma) \xrightarrow{R}_e (C,\sigma'') \xrightarrow{R}_p (skip,\sigma''') \xrightarrow{R}_e (skip,\sigma')$.

Доказана **лемма 6** об условии G при параллельном исполнении. Если $Guar(G_1, C_1, \sigma, R \cup G_2)$, $Guar(G_2, C_2, \sigma, R \cup G_1)$ и $\sigma_1 \circ \sigma_2 = \sigma$, то $Guar(G_1 \cup G_2, C_1 \parallel C_2, \sigma, R)$; если $(C_1 \parallel C_2, \sigma) \xrightarrow{R} *(C_1' \parallel C_2', \sigma')$, то $\exists \sigma_1', \sigma_2'.(C_1, \sigma_1) \xrightarrow{R \cup G_2} *(C_1', \sigma_1')$, $(C_2, \sigma_2) \xrightarrow{R \cup G_1} *(C_2', \sigma_2')$, $\sigma_1' \circ \sigma_2' = \sigma'$.

Доказана **лемма 7** об условии G при последовательном исполнении. Если $Guar(G, C_1, \sigma, R)$, $\forall \sigma' : (C_1, \sigma) \xrightarrow{R} *(skip, \sigma')$, $Guar(G, C_2, \sigma', R)$, то $Guar(G, C_1; C_2, \sigma, R)$.

Доказана **лемма 8**. Если $(C_1; C_2, \sigma) \xrightarrow{R} *(skip, \sigma'')$, тогда $\exists \sigma' : (C_1, \sigma) \xrightarrow{R} *(skip, \sigma')$ и $(C_2, \sigma') \xrightarrow{R} *(skip, \sigma'')$.

Для интерпретации выводимости как доказательства в смысле частичной корректности необходимо, чтобы система вывода обладала свойством непротиворечивости, т.е. чтобы при выполнении общезначимых формул получались общезначимые, что отражает следующая теорема о том, что предложенная система вывода непротиворечива для свойства частичной корректности: если $R; G; I \vdash \{p\} C \{p\}$ то $R; G; I \models \{p\} C \{p\}$. Программа считается прошедшей проверку, если для каждого потока, с помощью применение аксиом и правил, показан вывод формулы корректности.

Глава 4 посвящена способам задания формул корректности, приведены примеры задания формул, сделаны оценки сложности алгоритма анализа гонок.

В дедуктивных методах на основе Rely/Guarantee спецификация (R, G, пред/постусловия) выполнения команд потока выражается посредством задания формулами состояния памяти, для каждой логически законченной части подпрограммы. Но как известно, такой подход менее нагляден и обладает меньшей выразительной способность, чем спецификации посредством временных логик. В работе предложен способ связи спецификации, выраженной посредством временной логики LTL (Linear Temporal Logic) и формулами R и G в методе R/G. Применение временной логики, изначально приспособленной к рассуждению о последовательностях, к анализу вычислительного поведения параллельных программ, обосновано, поскольку

автором представлено вычисление программы как выполнимая последовательность программных состояний.

Синтаксис LTL включает в себя стандартные темпоральные операторы (X,F,G,U,R), булевы связки ($\neg, \wedge, \vee$) и пропозициональные переменные типа atl, которые пробегают по выражениям «поток φ_i находиться в метке l».

Синтаксис *LTL*-формул таково:

- пропозициональные переменные atl;
- True, False;
- φ и ψ – формулы, то: $\neg\varphi$, $\varphi\Lambda\psi$, $\varphi V\psi$ – формулы;

  $X\varphi$, $F\varphi$, $G\varphi$, $\varphi U\psi$, $\varphi R\psi$ – формулы.

В LTL логике различают формулы трассы τ (способные обращаться в истину на протяжении некоторой трассе) и состояния ψ (способные обраться в истину в некоторых состояния). Отношение выполнимости формул в состоянии или на пути $\models$ определяется стандартным образом по индукции.

Пример. Общий вид формулы частичной корректности в логике LTL имеет следующий вид: пусть φ – предусловие, а ψ - постусловие. Пусть l^0_1, l^0_2... l^0_n– исходные метки потоков параллельной программы $T_1||\ldots||T_n$, $l^в_1$, $l^в_2$... $l^в_n$- конечные метки этих же процессов. Тогда частичная корректность данной программы относительно φ и ψ может быть выражена следующей формулой: (atl^0_1& … atl^0_n & φ) $\supset$ $G(atl^в_1$& ... $atl^в_n$ & $\psi)$.. Что преобразуется в формулу частичной корректности в терминах троек Хоара $\{\varphi\}$ $T_1||\ldots||T_n$ $\{\psi\}$. А общий вид формула частичной корректности одного потока в предложенном методе имеет вид $R;G;I|-\{\varphi_\alpha\}T_\alpha\{\psi_\alpha\}$.

Пример. LTL вид формулы корректности, которая описывает свойство исключения критической секции: $\xi \supset G\neg(atl_1,...,atl_n)$, где ξ-формула, выражающая исходные предусловия, l_1,...l_n-метки входа в критическую секцию. В методе $G\neg(atl_1,...,atl_n);I|-\{\xi\}T_\alpha\{true\}$, где G-оператор темпоральной логики LTL.

Получены оценки сложности алгоритма анализа. В частности, оценка числа правил и аксиом для одного потока - O(n), где n - число операторов

потока; оценка общего числа правил вывода для анализа всех k различных потоков программы: $O(n^2kv)$, где v- число задействованных общих ячеек.

В **главе 5** представлены примеры анализа гонок в программах на разделяемой памяти. В качестве примеров рассмотрены операции добавления, удаления элементов из неблокирующегося стека и очереди, верифицирован «алгоритм булочной».

Рассмотрена неблокирующуяся реализация алгоритма стека.

Пусть:

```
class Node { Node * next; int data;};
void push(int t) {
        Node* node = new Node(t);
        do {
                node->next = head;
        } while (!cas(&head, node, node->next));
};
```

Пусть каждый поток представляет из себя последовательность подпрограмм void push() и в работе не рассматривается ABA-проблема, поэтому формула частичной корректности $R, G, I| = \{pre\}push\{\text{post}\}$ имеет следующие составляющие:

- Инвариант I: $\underline{\exists x. \&head \to x * lseg(x, NULL) * true}$, где $lseg(x, y) \Leftrightarrow (x = y \cap emp) \cup (\exists z. x \neq y \cap x \to_{*(x+1)} \to y * lseg(z, y))$.
- Предусловие *pre:* $\underline{[head] \to y}$
- Постусловие *post:* $\underline{[head] \to x * x \to t * (x + 1) \to y}$
- Для определения R,G заметим, что запись происходит посредством инструкции CAS. При условии равенства первых двух аргументов:

$$\{\underline{[head] \to y}\}CAS_{head}\{\underline{[head] \to x * x \to t * (x + 1) \to y}\}$$

Поскольку операции записи общих данных в функции push осуществляется только с помощью одного CAS, то сразу получаем выполнимость формулы частичной корректности, то есть отсутствие гонок.

Рассмотрена неблокирующаяся реализация алгоритма очереди.

Пусть определен массив int q[N] и две переменные int tail и int head. Здесь N – некоторое заранее заданное число, а массив и переменные расположены в разделяемой памяти. В массиве хранятся числа, добавляемые в очередь, переменная head содержит номер первого элемента в очереди, а tail – номер свободной ячейки, следующей за последним элементом очереди.

Добавление элемента в очередь выглядят следующим образом:

```
bool Enqueue( int x )
{     int t;
      while (1) {
A1:       t=tail;
A2:       if  (t == N) return 0;
A3:       if  (q[t] != NULL) {
A4:                 CAS(tail, t, t+1);
                    continue;}
A5:        if (CAS(q[t], NULL, x)) {
A6:                 CAS(tail, t, t+1);
A7:                 return 1;}
       } }
```

В работе не рассматривается ABA-проблема, поэтому формула частичной корректности $R, G, I| = \{pre\}enqueue\{\text{post}\}$ имеет следующие составляющие:

- Инвариант: $\underline{\underline{lseg(head, tail) * tail \rightarrow Null}}$
- Предусловие: $\underline{\underline{lseg(head, n, tail) * tail \rightarrow Null}}$
- Постусловие: $(\underline{\underline{lseg(head, n+1, tail) * tail \rightarrow Null * [n] \rightarrow x)}} \cup$

  $\underline{\underline{\cup\ (lseg(head, n, tail) * [tail] \rightarrow N)}}$

- Для определения R, G заметим, что запись происходит посредством инструкции CAS. При условии равенства первых двух аргументов CAS: $\{\underline{true}\}CAS_{tail}\{\underline{tail = tail + 1}\};\ lseg(head, n, tail) * [q + t] \rightarrow$

  $\underline{\underline{\rightarrow Null}}\}\ CAS_{q[t]}\{\underline{\underline{lseg(head, n, tail)}}\ \underline{\underline{* [tail] \rightarrow x \cup lseg(head, n, tail) \cap ([q + t] = x)}}\}$

Для наглядности выполнение разбито на типы проходов по циклу:

Типы проходов по циклу while(1)	**Guarantee**	**Rely**	**Rely/Guarantee**
Терминация в A7	$CAS_{q[t]}$ *CAS_{tail}	Inv	Inv
Терминация в A7	$CAS_{q[t]}$	CAS_{tail}	Inv
Терминация в A2	Inv	$Rely_0$	Inv *$Rely_0$
Прогон по циклу без терминации	Inv*CAS_{tail}	$Rely_{холостой}$	Inv*CAS_{tail} *$Rely_1$

Тогда выполнение программы удовлетворяет формуле корректности.

В **главе 6** представлен комплекс программ для автоматизации предложенного метода анализа гонок на общей памяти.

Для предложенного метода, как и для других аксиоматических методов, сложность автоматизации является одним из главных препятствий для широко применения. В целях автоматизации верификации программ в среде Isabella/HOL на базе генераторов Flex и Bison написан комплекс программа по автоматическому извлечению инструкции доступа к памяти, разложению операторов на атомарные, переводу выражений в постфиксную запись и т.п.

В **Заключении** приведены основные результаты работы.

ОСНОВНЫЕ РЕЗУЛЬТАТЫ РАБОТЫ

1. Разработана математическая модель выполнения многопоточных программ на разделяемой памяти.
2. Предложена методика задания формулы частичной корректности, которая специфицирует состояние гонки.
3. Предложен метод анализа параллельных программ на наличие гонок на основе расширения аксиоматического метода верификации Rely/Guarantee и сепарационной логики.
4. Оценена сложность алгоритма анализа гонок в параллельных программах предложенным методом.
5. Разработан комплекс программ для автоматизации предложенного метода анализа гонок.
6. Комплексно исследован ряд задач параллельного программирования на основе анализа их математических моделей с помощью предложенной методики и вычислительных экспериментов.

СПИСОК ПУБЛИКАЦИЙ ПО ТЕМЕ ДИССЕРТАЦИИ

1. Прокопенко А.С., Тормасов А.Г. Аксиоматический метод верификации на основе декомпозиции состояний в методе RGSep. // Научно-технические ведомости СПбГПУ, Серия “Информатика, телекоммуникации, управление.” – СПб.: Изд-во Политехн. ун-та, 2010. – № 6 (113). – С. 112-121.
2. Кудрин М.Ю., Прокопенко А.С., Тормасов А.Г. Метод нахождения состояний гонки в потоках, работающих на разделяемой памяти // Труды МФТИ. – М.: МФТИ, 2009. – № 4. – Том 1. – С. 181-201.
3. Прокопенко А.С., Тормасов А.Г. Синтез метода Джонса и сепарационной логики для дедуктивной верификации параллельных программ на наличие условий гонки // Высокопроизводительные вычислительные системы ВПВС-2010. Материалы Седьмой Международной научной молодежной школы. – Ростов-на-Дону – Таганрог, 2010. – С. 240-244.
4. Прокопенко А.С., Тормасов А.Г. Дедуктивный метод анализа логических гонок с использованием сепарационной логики. // Труды МФТИ. — М., 2010. – Т.2. № 3. – С. 175-184.
5. Кудрин М.Ю., Прокопенко А.С., Тормасов А.Г. Детектор логических гонок для программ, работающих на разделяемой памяти // Высокие технологии, исследования, промышленность. Сборник трудов девятой международной научно-практической конференции «Исследование, разработка и применение высоких технологий в промышленности». – С.-Пб, 2010. – С. 67-71.
6. Заборовский Н.В., Кудрин М.Ю., Прокопенко А.С., Тормасов А.Г. Автоматизация алгоритма поиска логических гонок в программах на разделяемой памяти // XXXVI Гагаринские чтения. Международная молодежная научная конференция. Научные труды. Т. 4. – Москва: МАТИ, 2010. – С. 91-92.
7. Кудрин М.Ю., Петров В.Н., Прокопенко А.С., Тормасов А.Г. Математическое моделирование структур, работающих на разделяемой

памяти // Материалы международной научно-технической конференции "Многопроцессорные вычислительные и управляющие системы 2009". – Таганрог: Изд-во ТТИ ЮФУ, 2009. – Том 1. – С. 73-74.

8. Кудрин М.Ю., Петров В.Н., Прокопенко А.С. Обнаружение состояний гонки в потоках, работающих на разделяемой памяти // Модели и методы обработки информации, сборник научных трудов. - М.: МФТИ, 2009. - С. 93-98.
9. Кудрин М.Ю., Петров В.Н., Прокопенко А.С. Математическое моделирование структур, работающих на разделяемой памяти // XXXVI Гагаринские чтения. Международная молодежная научная конференция. Научные труды. Т. 4. – Москва: МАТИ, 2009. – С. 24-25.
10. Прокопенко А.С. Математическое моделирование структур, не требующих блокирования при параллельном доступе. // Современные проблемы фундаментальных и прикладных наук. Часть VII. Управление и прикладная математика: Труды 50-й научной конференции. – М.– Долгопрудный: МФТИ, 2007. – С. 62-63.
11. Прокопенко А.С., Тарасов В.Н. Перспективы применения структур, не требующих синхронизации при параллельном доступе // Современные проблемы фундаментальных и прикладных наук. Часть VII. Управление и прикладная математика: Труды 49-й научной конференции. – М.– Долгопрудный: МФТИ, 2006. – С. 50.

В работах с соавторами лично соискателем выполнено следующее: построение математической модели исполнения многопоточных алгоритмов, разработка формулы корректности, проведение вычислительных экспериментов для комплексного исследования задач параллельного программирования, анализ гонок параллельного программирования с помощью данного метода.

Прокопенко Артем Сергеевич

СТАТИЧЕСКИЙ АНАЛИЗ УСЛОВИЙ ГОНКИ В ПАРАЛЛЕЛЬНЫХ ПРОГРАММАХ НА РАЗДЕЛЯЕМОЙ ПАМЯТИ

Автореферат

Подписано в печать 12.11.2010 Формат 60x90/16.
Усл печ л 1.0. Тираж 90 экз. Заказ No 770.
Московский физико-технический институт
(государственный университет)

Печать на аппарате Rex-Rotary Copy Printer 1280. НИЧ МФТИ.

141700, г Долгопрудный Московской обл, Институтский пер, 9,
тел.: (095) 4088430, факс (095) 5766582