МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
им. М.В. Ломоносова

Труды

ФАКУЛЬТЕТА ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ

№ 1

ПРОГРАММНЫЕ СИСТЕМЫ И ИНСТРУМЕНТЫ

Тематический сборник

Под редакцией
чл.-корр. РАН Л Н. Королева

Москва
МАКС Пресс
2000

УДК 519.6+517.958
ББК 22.18:22.19
П78

Печатается по решению Ученого Совета
факультета вычислительной математики и кибернетики
МГУ им. М.В. Ломоносова

П78 **Программные системы и инструменты:** Тематический сборник факультета ВМиК МГУ им. Ломоносова: № 1/ Под ред. Л.Н. Королева. – М.: МАКС Пресс, 2000. – 164 с.
ISBN 5-317-00116-1

В данный сборник включены научные работы по следующим темам:

- общие вопросы программирования и информатики;
- иммитационное моделирование и синтез систем обработки и передачи информации;
- прикладные программные системы.

В этих публикациях нашли отражение исследования и разработки в области создания программных систем, выполненных учеными, аспирантами и студентами факультета. Большая часть статей и тезисов доложена на Ломоносовских Чтениях 2000 года.

УДК 519.6+517.958
ББК 22.18:22.19

Напечатано с готового оригинал-макета

Издательство ООО "МАКС Пресс"
Лицензия ИД N 00510 от 01.12.99 г.
Подписано к печати 18.12.2000 г.
Усл.печ.л. 10,25. Тираж 200 экз. Заказ 732.
Тел. 939-3890, 939-3891, 928-1042. Тел./Факс 939-3891.
119899, Москва, Воробьевы горы, МГУ.

ISBN 5-317-00116-1

СОДЕРЖАНИЕ

СБОРНИК

" Программные системы и инструменты"

От редактора

В предлагаемом читателю тематическом сборнике публикуются статьи, посвященные в основном описаниям инструментальных программных систем, разработанных авторами публикаций.

В нем также публикуются статьи и заметки общего характера, касающиеся разделов информатики, тесно связанных с программированием.

Большинство статей представляют собой тексты докладов, прочитанных на Ломоносовских Чтениях 2000 на факультете ВМиК.

Перечень разделов сборника таков:

- прикладные программные системы;
- имитационное моделирование и синтез систем обработки и передачи информации;
- общие вопросы программирования;
- сообщения.

Статьи сборника будут интересны специалистам, разрабатывающим прикладные программные системы и исползующим новые информационные технологии.

Л.Н. Королев

Раздел I
Общие вопросы программирования и информатики

Королев Л.Н., Смелянский Р.Л.

Проблемы преподавания программирования в классическом университете

- **Преподавание программирования – дело почти безнадежное, а его изучение – непосильный труд.**
- *Ч.Уэзерел Этюды для программистов. М,:Мир, 1982*
- **Программист, если он хорош, то уже очень**
- **и очень хорош, но уж если он плох, то просто ужасен.**
- *Иодан Э Структурное проектирование и*
- *конструирование программ. М,:Мир,1979*

1. Характеристика дисциплины

Профессия - это люди, технологии, организации, нацеленные на решение проблем и удовлетворение нужд в определенной области деятельности. Программирование как профессия - это люди, технологии и организации, занятые решением проблем в области создания систем обработки информации, вычислений и взаимодействия посредством систем передачи данных. Мы определим область программирования, как систематическое изучение алгоритмических процессов преобразования и передачи информации: их теории, анализа, разработки, реализации, оптимальности, описания. Корневым вопросом программирования является: что может быть эффективно алгоритмизировано и как это может быть эффективно реализовано?

В докладе рассматриваются проблемы преподавания программирования при подготовке сппециалистов по прикладной математике и информатике в классическом университете. Значимость этой профессии для современного общества трудно переоценить. Вот лишь несколько фактов. В Германии при населении в 80 млн.человек на сегодня официально признанная нехватка рабочих мест в области ИТ (информационных технологий) составляет 60 тыс.рабочих мест, на переподготовку гос.служащих только в 2000 году выделено 2 млрд. марок. Практически все промышленно развитые западные страны называют близкие цифры нехватки специалистов в области ИТ. Наверняка все наслышаны о программах типа «green card», призванных восполнять эту нехватку за счет иммиграции.

Еще один пример – эффект Интернета. На сегодня успешное существование организации, не важно государственной или коммерческой, не возможно без доступа в Интернет, без возможности представлять себя в Интернете, без информационной системы, способной как поддерживать деятельность самой организации, так и взаимодействовать с такими же системами других организаций.

Другой пример. Оборот в индустрии производства программного (только программного !) обеспечения превысил оборот автомобилестроения в США! По

прогнозам производителей кристаллов в 2000 году будет произведено около 176 млрд.штук микропроцессоров. Это в несколько раз больше чем телевизоров.

И еще один факт. Обще признаны три основные движущие силы ИТ в промышленности – микропроцессоры, телекоммуникация, разработка программного обеспечения. На сегодня темпы их развития характеризуют следующие цифры:

- производительность микропроцессоров удваивается без увеличения стоимости кристалла каждые 18 месяцев, т.е. на 67% в год;
- пропускная способность каналов связи возрастает на 75% год;
- производительность труда программистов (скорость создания программ) – 4-5% в год.

Программирование, как наука, безусловно базируется на классических дисциплинах общей математики (общая математическая культура). Она также очень тесно связано со многими специальными математическими дисциплинами. Здесь характерно с одной стороны интенсивное взаимное проникновение идей, теорий и методов, с другой – очень высокий темп обновления знаний в этих областях. По существу, программисту в течение его продуктивной жизни требуется постоянно изучать, осваивать и развивать новые методы, теории и идеи. Для этого ему необходима фундаментальная математическая подготовка как в области теоретической, так и в области естественного научного моделирования.

2. Характерные виды деятельности

Любая профессиональная деятельность в области естественных наук может быть отнесена к одной из следующих категорий: теория, моделирование и инженерия.

Первая категория характеризуется следующим стилем деятельности:

1) определение объектов изучения;
2) формулировка гипотез о возможных отношениях между объектами (теоремы);
3) доказательство теорем.

Моделирование характеризуется:

1) формированием гипотез;
2) построением модели и прогноза на ее основе;
3) постановкой эксперимента и сбором данных;
4) анализом результатов.

Инженерия характеризуется:

1) формулировкой требований к изделию;
2) выработкой спецификации изделия;
3) разработкой проекта и реализацией изделия;
4) проверкой изделия.

В программировании все эти три вида деятельности столь сильно переплетены, что выделить одну из них как фундаментальную вряд ли возможно. И вместе с этим, они совершенно различны, требуют различных навыков и умений.

Теория требует умения определять, видеть отношения между объектами и доказывать их в рамках формальных систем. Моделирование – использовать эти

отношения в целях прогнозирования. Инженерия – материализовать эти отношения в целях практики.

В области программирования все эти три вида деятельности сходятся. Вычисление как процесс и вычислительная система как средство его реализации ровно поддерживают и обеспечивают разработку сложных инженерных систем, моделирование физических процессов, доказательство теорем. Следовательно, все эти три вида деятельности важны для программирования и являются его областью компетенции.

Особую роль в программировании занимает кодирование или техника использования языка программирования и инструментальных средств, которое очень часто путают с программированием. Важно подчеркнуть, что многие разделы программирования не связаны с кодированием. Примеры тому – анализ электронных систем, архитектур вычислительных систем, структур операционных систем, разработка приложений баз данных, анализ функционирования вычислительных систем и т.п. Поэтому часто встречаемая формула - программирование=кодирование - не верна!

3. Экономические и социальные факторы

Рассматривая проблемы преподавания программирования, следует учитывать новые экономические и социальные факторы, которые носят долгосрочный характер:

- существенное сокращение государственной поддержки образования, которой не достаточно для полноценной подготовки специалистов, развития и пополнения профессорско-преподавательского состава;
- отток квалифицированных, опытных кадров за рубеж и коммерческие структуры;
- резкое сокращение НИР и НИОКР в области программирования в стране (возможности зарубежных фондов пока плохо умеем использовать);
- рыночные механизмы трудоустройства вместо централизованного планирования и распределения выпускников;
- устойчивый высокий спрос на специалистов по IT как в стране, так и за рубежом;
- появление конкурентной борьбы на рынке образования в стране;
- спрос на обучение из-за рубежа (иностранные учащиеся);
- интернационализация (глобализация) образования.

4. Проблемы преподавания программирования

1. Перекос программы подготовки специалистов по прикладной математике и информатике в область «численной» подготовки. Это проявляется в чрезмерно большом объеме предметов из области «непрерывной» математики в программе.
2. Отсутствие необходимых курсов в области современной алгебры, дискретной математики, логики.
3. Слабая согласованность обще математических курсов и прикладных дисциплин между собой.
4. Катастрофическое старение профессорско-преподавательского состава.
5. Отсутствие НИР и НИОКР проектов, базы для производственной практики студентов.
6. Острая нехватка преподавателей в области инженерии программного обеспечения.

7. Острая нехватка учебно-научной литературы на русском языке.
8. Отсутствие средств, для приглашения ведущих специалистов, как лекторов.
9. Угроза утери понимания направлений развития и проблем в некоторых областях программирования.
10. Отсутствие ясного понимания, видения требований к современному выпускнику классического университета со стороны промышленности.

В этой связи предлагается организовать методическую работу по созданию целостной программы по подготовке специалистов в области ИТ.

5. Цели программы

Представляется, что такая программа должна учитывать зарубежный опыт в этой области. Так, например, в Carricula 91 (сейчас готовится Carricula 2000) выделяют девять основных подобластей программирования, поддержанные математическими дисциплинами, указанными в скобках:

1. алгоритмы и структуры данных;

(алгебра, исчисление предикатов, элементы теории алгоритмов и теории графов, комбинаторика, теория сложности вычислений, элементы теории вероятностей)

2. языки программирования;

(алгебра, теория формальных грамматик, логика, функциональный анализ, теория автоматов)

3. архитектуры вычислительных систем (включая квантовые, нейросетевые вычислители);

(алгебра, логика, теория графов, математический и функциональный анализ, теория вероятностей, теория чисел, комбинаторика, теория кодирования, численный анализ)

4. численные и символические вычисления;

(алгебра, математический и функциональный анализ, теория чисел, теория рекурсивных функций)

5. операционные системы и сети;

(алгебра, логика, теория графов, теория вероятностей, теория чисел, комбинаторика, теория кодирования, теория алгоритмов, исследование операций, целочисленное программирование, теория управления)

6. инженерия программного обеспечения;

(алгебра, теория алгоритмов, комбинаторика, логика, теория графов, теория вероятностей, оптимизация на дискретных структурах)

7. базы данных и информационный поиск;

(алгебра, комбинаторика, теория чисел, логика, теория вероятностей, теории сложности вычислений, теория графов)

8. искусственный интеллект и робототехника;

(теория формальных грамматик, алгебра, логика, математический анализ, численный анализ, теория вероятностей, дифференциальные уравнения, теория управления, исследование операций, теория оптимизации)

9. человеко-машинный интерфейс.

(алгебра, математический анализ, логика, численный анализ, теория графов, комбинаторика)

В этом списке после каждого пункта в скобках перечислены лишь некоторые математические дисциплины, знание которых необходимо для овладения

теориями и методами соответствующих подобластей программирования. Авторы не претендуют на полноту перечисления, а указали лишь неиболее важные дисциплины с их точки зрения.

Каждая подобласть разбита на, так называемые, Knowleg unit'ы. Из этих unit можно формировать программу обучения, учитывая специфику вуза и специализацию.

Требования к такой программе можно было бы сформулировать следующим образом:

1. Программа должна сформировать у студентов широкую базу знаний в области программирования, достаточную как для того, чтобы работать в этой области, как в академической, так и для того, чтобы заниматься в ней практической деятельностью. Это предполагает - умение ясно видеть проблему и ее значимость, определять, когда необходимо привлечь в качестве консультанта внешнего эксперта, выбирать надлежащую стратегию решения, изучать, описывать, разрабатывать, реализовывать, тестировать, модифицировать и документировать решения, выявлять альтернативные пути и оценивать степень их рискованности, уметь взаимодействовать в поиске решения со специалистами из других областей и непрофессионалами, умение работать в команде.
2. Программа должна учитывать опыт преподавания программирования в крупных университетов, например, таких как Беркли, Стэнфорд, Эдинбург, Оксфорд, Массачусетский технологический институт.
3. Программа должна иметь глубину изложения, достаточную для того, чтобы продемонстрировать студентам богатство теории, лежащее в основе программирования, чтобы они могли оценить интеллектуальную привлекательность и глубину процесса абстракции, с тем, чтобы в будущем заниматься научно-исследовательскими разработками в области программирования.
4. Программа должна отражать не только специальные, но и правовые, этические и социальные проблемы, связанные с областью программирования.
5. Программа должна помочь студентам найти область и вид профессиональной деятельности, наиболее соответствующие их способностям, и, в крайнем случае, как можно скорее обнаружить нецелесообразность дальнейшего обучения.
6. В этой программе должен быть учтен факт необычно высокой скорости изменений в области программирования, особенно ее технологии и теории, необходимости получения фундаментальных, долгосрочных знаний.
7. Эта программа должна быть не просто набором разрозненных, изолированных курсов. В основе принципов ее организации должен лежать метод связывания курсов в единое целое, называемое программой.

5.1. Основные принципы построения программы.

1. Все курсы программы предлагается подразделить на основные и факультативные. Основные курсы должны охватывать всю область программирования и обеспечивать широкий, согласованный базис знаний, обеспечивающий свободу выбора направления профессиональной деятельности. И, в частности, по завершении

2. обучения на первой ступени студент должен быть вполне готов к практической инженерной работе в области программирования. Глубина изучения программирования по отдельным подобластям должна достигаться за счет факультативных курсов, выбираемых студентом.
3. Все основные курсы должны быть «вытянуты» в цепочки, вдоль которых работают близкие группы понятий, что способствует закреплению материала и становлению этих понятий рабочими.
4. Курсы должны строиться так, чтобы практическая работа по ним была бы их неотъемлемой частью. Задача лектора – продумать и составить список задач, способ, порядок и сроки их выполнения. Задача ассестирующего персонала – обеспечить работоспособность оборудования, нужный состав программного обеспечения, консультации и т.п.

5.2. Связующие принципы

Ниже перечисленные принципы образуют основу метода «склеивания» отдельных курсов в единую программу, часть которых взята из Carricula 91. Предполагается, что эти принципы должны быть акцентированы особо в каждом курсе. Лектор должен сделать это осознанно и явно, сославшись по возможности на то, как эти принципы выглядят в других курсах. Тем самым предполагается обеспечить согласованность курсов. Ниже перечислены четырнадцать таких принципов.

1. Связывание логической (абстрактной) сущности с ее физической реализацией. Например: тип – имя объекта; имя объекта – адрес его физического размещения.
2. Сложность больших проблем: эффект нелинейного роста сложности с ростом размерности задачи. Следовательно, к одной и той же задаче могут быть применены различные методы в зависимости от ее сложности.
3. Концептуальные и формальные модели: различные способы формализации и описания объектов.
4. Согласованность и полнота: согласованность набора аксиом как формальной спецификации чего-либо; согласованность теорем с наблюдаемыми фактами; внутренняя согласованность языка, интерфейса. Полнота включает способность заданного набора аксиом охватывать все необходимые поведения, функциональную адекватность программ и аппаратуры.
5. Конструктивность: объект строят путем конечного применения определенных операций к базовому набору атомарных объектов.
6. Разрешимость: существование решения в области алгоритмизации.
7. Эффективность: мера цены по отношению к ресурсам, таким как время, пространство, деньги, люди и т.п.
8. Эволюция: изменения и их влияние на функционирующую систему. Примеры: способность формальных моделей отражать те аспекты систем, которые изменяются во времени, средства и методы реконфигурации систем.
9. Уровни абстракции: характер и использование абстракций в различных областях. Примеры: различные уровни описания аппаратуры, уровни спецификации объектов, понятие общности в языках программирования

10. Концепция локальности: например, физическая (как в памяти), логическая (области видимости в языках программирования), организационная, структурная (процесс – процессор и т.п.)
11. Концепция времени: время как средство синхронизации, время как мера эффективности алгоритма, время как отражение причинной упорядоченности.
12. Повторное использование (переиспользование) как возможность использования конкретных методов, концепций и компонентов системы в новых контекстах и ситуациях. Например, переносимость программ, технологии, способствующие переносимости программ, абстракции в языках программирования, способы, обеспечивающие повторное использование программ.
13. Безопасность: способность системы (программ и аппаратуры) реагировать должным образом и защищать себя от некорректных и незаконных запросов. Например, проверка типов и другие концепции в языках программирования, санкционирование доступа в базах данных, сетях. Надежные методы идентификации объектов в системах.
14. Компромиссы и их последствия: время – память, универсальность – специализация и т.п.

В качестве первого приближения к конструированию такой программы следовало бы определить:

1. Согласованную временную последовательность чтения сложившихся курсов по программированию и фундаментальной математике, а также соответствующую модификацию учебных планов.
2. Определить те разделы программирования и математики, которые следова ло бы отнести к основным, а какие к факультативным курсам.
3. Обозначить те разделы фундаментальной математики, которые следует внести в программы традиционных курсов с учетом развития современных идей в области информационных технологий.

Эта работа должна проводиться в тесном контакте специалистов, лекторов и преподавателей, математиков и программистов, непосредственно участвующих в преподавательской деятельности.

Авторы понимают всю сложность разработки такой программы, осознают невозможность резкой перестройки и ломки сложившихся традиций. Однако такой процесс должно начинать.

Брусенцов Н.П.

Блуждание в трех соснах

(Приключения диалектики в информатике)

Философы все еще не могут договориться о том, что же значит слово *информация*, но практикам исчерпывающая ясность, похоже, и не нужна: *неопределенность* ведь в каком-то смысле *свобода*. Бог с ней, с информацией, практики создают *информационные системы, информационные технологии* (также толком не определив, что это такое), и все мы уже знаем, что очередным этапом человеческой цивилизации будет *информационное общество*. Впрочем, непонятно, почему бы не *виртуальное*. Недоумения не будет, если в приведенных словосочетаниях вместо *информационные* говорить *компьютерные*. Конечно, не все *информационное* должно быть *компьютерным*, но в современном понимании информатизация - это не более чем компьютеризация, и слово *информатика* в новом смысле употребляется как синоним англоязычного *computer science*.

Однако по справедливости, информатика должна быть наукой об информации, т.е. об отображениях бытия не только компьютерных, но всевозможных - мысленных, письменных и устных, алгебраических, графических, изображающих, выражающих, подражающих, карикатурных и т.п. Аристотель называл эту науку *первой философией*, понимая ее как способ (Органон) исследования, прокладывающий "путь к началам всех учений".

Сущность этого способа в том, что, исходя из правдоподобного отображения реальной ситуации, применением его доказательно выявляются оправдывающиеся на практике заключения об этой ситуации. Слова, которыми в нашем языке отображены объекты реальности, Аристотель уподоблял счетным камешкам в том смысле, что при помощи слов можно моделировать всевозможные взаимосвязи, подобно тому как посредством камешков моделируются взаимосвязи количественные:

"... так как нельзя при рассуждениях приносить самые вещи, а вместо вещей мы пользуемся как их знаками именами, то мы полагаем, что то, что происходит с именами, происходит и с вещами, как это происходит со счетными камешками для тех, кто ведет счет."

Другими словами, у традиционного анализа реальных ситуаций методом проб и ошибок имеется "информационная альтернатива" - отобразить рассматриваемую ситуацию посредством слов и производить необходимые исследования полученного отображения методами аристотелева Органона, т.е. подменить физическое моделирование информационным. Разумеется, методы Органона должны гарантировать практическую подтверждаемость получаемых результатов в учтенных отображением условиях конкретной ситуации.

В Органоне функции счетных камешков выполняют *термины* - слова либо буквы, обозначающие *критерии*, по отношению к которым характеризуются рассматриваемые вещи. Сущности вещей (не только отдельных материальных предметов, но и всевозможных взаимосвязей, ситуаций, процессов, как реальных, так и мыслимых) отображаются совокупностями *определенностей* их

в отношении принятых критериев. Например, погоду можно охарактеризовать в таких терминах, как "ветренная", "дождливая", "холодная", "пасмурная", "промозглая" и т.п. Если характеризуемая вещь удовлетворяет данному критерию, соответствующий термин входит в ее определение непосредственно, в утвердительной форме, а если не удовлетворяет, то в отрицательной - под знаком *отрицания*. В русском языке таким знаком служит, как правило, частица *не* ("не-ветренная", "не-холодная"), а друг с другом термины, и утверждаемые, и отрицаемые, связываются выражающим совместность соответствующих определенностей союзом *и*, который в многочленных определениях обычно опускают. Вместо "ветренная, и дождливая, и холодная" говорят "ветренная, дождливая и холодная" или "ветренная, дождливая, холодная".

Подобная форма определения сущности вещи в булевой алгебре называется *элементарной конъюнкцией*. Терминами в алгебре, как и в Органоне, служат буквы, поскольку исследуются не конкретные критерии и определенности, а виды взаимосвязей, применяемых к определенностям любой природы. В роли связки *и* употребляется знак конъюнкции *(совместности)* $\wedge$, который, подобно знаку умножения в числовой алгебре, обычно умалчивают (опускают) - вместо $x \wedge y$ пишут xy. Нередко конъюнкцию называют логическим умножением, хотя как раз умножения в ней нет - в отличие от умножения $x \cdot x \equiv x^2$ она идемпотентна: $x \wedge x \equiv x$. Впрочем, следуя Булю, можно рассматривать ее как умножение чисел, допускающих только два значения: 1 - "дано", 0 - "исключено".

В булевой алгебре вместо не-x пишут $\neg x$, либо надчеркнутое x, либо x'. Применительно к несоставному, не детализирусмому в рамках проводимого рассмотрения, термину эти символы тождественны друг другу, синонимы. Однако в общем случае, когда буква обозначает произвольное булево выражение, их следует различать либо вводить какие-то иные знаки для представления возникающего многообразия взаимосвязей. Условимся, что постфикс $'$ обозначает *инверсию* выражения, префикс $\neg$ - *дополнение* в универсуме терминов-критериев, а надчеркивание употреблять не будем.

Применительно к двучленной конъюнкции xy это значит:

$$(xy)' \equiv x'y'$$
$$\neg(xy) \equiv x' \vee y'$$

Знак $\vee$ символизирует *дизъюнкцию* - взаимосвязь, *двойственную* конъюнкции, в русском языке представленную союзом *или*. Двойственность понимается в том смысле, что произвольное выражение булевой алгебры, если в нем заменить конъюнкции дизъюнкциями, а дизъюнкции конъюнкциями, будет представлять ту же взаимосвязь при условии, что значение 1 истолковывается как 0, а значение 0 - как 1. Так, $xy = 1$ означает совмещение двух: $x = 1$ и $y = 1$, а $x \vee y = 0$ соответственно $x = 0$ и $y = 0$, т.е. конъюнкция отображает совместность единиц, а дизъюнкция - совместность нулей. С другой стороны, при $x \vee y = 1$ термины x, y *несоисключимы*, не могут вместе принять значение 0, а при $xy = 0$ они *несовместимы*, исключена совместность 1.

Заметим , что дополнение булева выражения двойственно его инверсии: в приведенном примере дополнительное выражение $x' \vee y'$ отличается от инверсного $x' \wedge y'$ тем, что в нем заменен двойственным (перевернут,

"инвертирован") знак $\wedge$. Взаимосвязь операций инверсии, дополнения и получения двойственного ("дуалирования") δ (διπλοη - двойственное) булева выражения *e* представима тождествами:

$$e' \equiv \delta(\neg e) \equiv \neg(\delta e), \quad \neg e \equiv \delta(e') \equiv (\delta e)', \quad \delta e \equiv \neg(e') \equiv (\neg e)'$$

Странно, что это фундаментальное соотношение выявлено не логиками и не математиками, а психологом Жаном Пиаже. Впрочем, не странно, ибо логики и математики приучены к булевой алгебре с единственной одноместной операцией отрицания-дополнения, которую иногда называют также инверсией, либо полагают, что инверсия - операция не булева, а теоретико-множественная, множественное дополнение.

Джордж Буль изобрел "математику мысли", устранив из числовой математики все значения, кроме 0 и 1, интерпретируемых как "нет" и "есть", либо "исключено" и "дано", либо "ложь" и "истина". Такую систему называют *двузначной*, что не представляется верным, ибо двузначность - синоним двусмысленности. Корректней назвать ее *двухзначной* системой, двухзначной логикой. Но это только поверхностное, "косметическое" уточнение, а по существу проблема двухзначности несравненно глубже, фундаментальней. Противопоставленный стоиками аристотелеву Органону хрисиппов *принцип двухзначности* в его "классической" трактовке (либо истина, либо ложь и ничего третьего) радикально отделил формальную логику, и традиционную, и математическую, от диалектики.

Впрочем, основоположник математической логики Буль, не в пример современным представителям этой науки, сосредоточившим все внимание на проблеме двухзначного (дихотомического) вывода, считал важнейшей ее задачей решение логических уравнений, чем и оправдывалось название "математическая". Решение этой *обратной* задачи, предпринятое самим Булем, показало, что удовлетворяющим логическому уравнению значением термина может быть не только 1 либо 0, но и нечто третье - "неопределенность", которую Буль обозначал буквой u ($u \equiv 0/0$). В дальнейшем выяснилось, что в зависимости от условий, определяемых значениями прочих входящих в уравнение терминов, для искомого термина x имеется четыре альтернативы: 1) $x = 0$, 2) $x = 1$, 3) x свободно, не фиксировано (у Аристотеля - "привходяще"), 4) решение не существует.

Например, решение относительно термина y уравнения $xy = 0$, как нетрудно проверить, таково: при $x = 0$ значение y привходяще, при $x = 1$ $y = 0$. Решение уравнения $x \vee y = 0$, т.е. $x'y' = 1$, при $x = 1$ не существует, при $x = 0$ $y = 0$.

Как члены элементарной конъюнкции, которой охарактеризована рассматриваемая вещь, скажем, xy', термин x утверждается, а термин y, входящий в конъюнкцию под знаком отрицания, отрицается относительно этой вещи. В духе Аристотеля можно сказать, что определенность x необходимо присуща данной вещи, а присущность ей определенности y исключена, она антиприсуща. Но по Аристотелю определенность может быть, кроме того, *привходящей*, т.е. не присущей и не антиприсущей с необходимостью, а - то быть, то не быть, как попало. Представляющий такую определенность термин, скажем, z, рассматриваемая конъюнкция не содержит ни в утвердительной, ни в отрицательной форме, он не утверждается и не отрицается. Строго говоря,

определенность z в этом случае будет *потенциально* привходящей - возможно присущей, возможно антиприсущей, возможно привходящей. Актуально привходящее, исключающее возможность присущности и возможность антиприсущности, в "классической" двухзначной системе невыразимо.

Буль наряду с операцией отрицания применял операцию *элиминации* (устранения) термина, которая была затем усовершенствована П.С.Порецким. Выходит, что и в двухзначной булевой алгебре термин можно либо утверждать, либо отрицать в смысле антиутверждать, либо не утверждать и не отрицать, а "элиминировать", устранить из выражения, "опустить".

Обращаясь к древним грекам, нетрудно убедиться, что в логике их языка хрисиппова двухзначность не доминировала, но впоследствии привела к такому искажению смысла слов, обозначающих базисные взаимосвязи, что написанное Аристотелем стало непостижимым. Слова: καταφασις - утверждение, αποφασις - не-утверждение, αντιφασις - анти-утверждение, составляющие основу аристотелева соотнесения объектов ("быть благом", "не быть благом", "быть не благом"; "Всякое А есть Б", "Некоторое А есть не Б", "Всякое А есть не Б"), стали понимать иначе, будто αποφασις - "отрицание", αντιφασις - "противоречие". Но ведь и απο- и αντι- означают отрицание и оба порождают выражения, противоречащие утверждению. В чем же логика?

По Аристотелю конъюнкция не-утверждения и не-антиутверждения ("не быть благом *и* не быть не благом") составляет *третье*, среднее, промежуточное между утверждением и антиутверждением - συμβεβηκος (привходящее). Хрисипп же "упростил" логику, изъяв это третье, а вместе с ним адекватность реальности и здравому смыслу. У него дискретная дихотомия - "да"/"нет", поэтому αποφασις ≡ αντιφασις, "не быть благом" ≡ "быть не благом". Это мир "рыцарей" и "лжецов" из "занимательной логики": "рыцарь" никогда не лжет, "лжец" лжет всегда; если некто не "рыцарь", то он "лжец", а если не "лжец", то "рыцарь" - все четко и просто, но не так, как в действительности.

Поразительна живучесть хрисипповой "простоты". На протяжении двух с лишним тысячелетий имели место лишь единичные попытки преодолеть роковую ограниченность (Раймонд Лулий, Уильям Оккам, Ян Коменский, Лейбниц, Гегель, Льюис Кэррол).

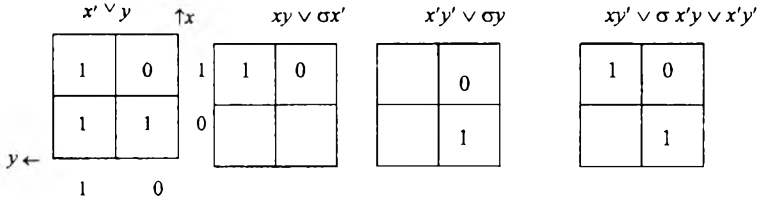
Двадцатый век ознаменован прогрессирующим нарастанием протеста против двухзначности: отвержение интуиционистской математикой закона исключенного третьего, попытки Льюиса, а затем Аккермана преодолеть "парадоксы" материальной импликации, изобретение Лукасевичем трехзначной логики, предположение Рейхенбаха о трехзначности логики микромира (квантовой механики), общее усиление активности в области многозначных логик, наконец, нечеткие множества Заде, справедливо квалифицируемые "как вызов, брошенный европейской культуре с ее дихотомическим видением мира в жестко разграничиваемой системе понятий". Однако все это пока как бы некий "модерн", не достигающий преследуемых целей, да и сами цели еще далеко не осознаны. Хрисиппова же "классика" обрела второе дыхание в исчислениях математической логики, в двоичной цифровой технике, и с позиций ее столь же непросто постичь недвухзначное, как, скажем, обитателям двухмерного мира представить себе мир трехмерный.

Показателен пример Яна Лукасевича, который, связывая создание им трехзначной логики "с борьбой за освобождение человеческого духа", затем (надо полагать, в продолжение этой борьбы) в своей неординарной книге "Аристотелева силлогистика с точки зрения современной формальной логики" устанавливает "ошибочность" положений трехзначной логики Аристотеля, формально "верифицируя" их в двухзначном исчислении высказываний. Впрочем, его попытки формализации модальностей средствами трех- и четырехзначного исчислений также не достигли цели. Он обратился к многозначности, осознав, что "модальная логика не может быть двухзначной", однако не сумел преодолеть традиции и выявить трехзначность аристотелевой силлогистики, чего ранее уже достиг поборник "эмансипации логики от влияния Аристотеля" Н.А.Васильев, в 1911 году преобразовавший логический квадрат A - I - O - E в треугольник противоположностей A - IO - E.

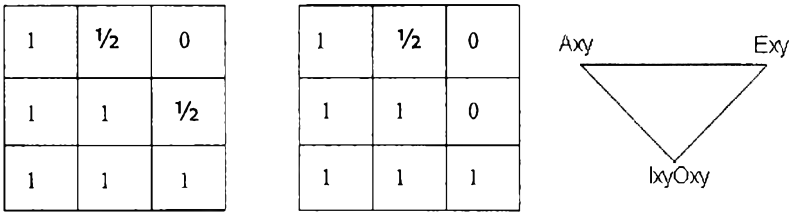
Этот треугольник и есть "три сосны", в которых заблудились логики 20-го века в попытках изобрести то, что в древности естественно и неопровержимо установил Аристотель. Изобретать вынуждала неадекватность двухзначной логики, в частности, невыразимость в ней сущности естественноязыкового (содержательного) следования. Первой, получившей значительный и все еще не угасший резонанс, была попытка Льюиса (1918 г.) устранить "парадоксы" материальной импликации, модифицировав аксиоматику классического исчисления высказываний. Но "строгая импликация" Льюиса оказалась тоже парадоксальной, да и неясно, что она такое, поскольку при помощи связок двухзначного исчисления определить конструктивно ее нельзя, а введенная в него "модальная функция самосовместимости-возможности" в свою очередь лишена определения.

Импликация Лукасевича (1920 г.) определена трехзначной таблицей истинности, но как заметил полвека спустя Слупецкий, смысл ее "довольно-таки неуловим". Сам Лукасевич впоследствии, признав трехзначное исчисление недостаточным, разработал четырехзначную модальную логику, однако именно его трехзначная импликация инициировала необыкновенную активность в области трехзначных логик и алгебр, в результате которой теперь имеется множество импликаций (интуиционистская Гейтинга, сильная и слабая Клини, внешняя и внутренняя Бочвара, Геделя, Собочинского, ...), смысл которых столь же неуловим и, увы, не тождествен содержательному следованию. Это удивительное блуждание в трех соснах обусловлено тем, что ищут, не зная что. Операции определяются не путем воплощения подразумеваемого смысла, а либо формальным обобщением соответствующих двухзначных таблиц истинности, в частности, таблицы материальной импликации, истолкование которой в свою очередь проблематично, либо модификацией системы аксиом, например, изъятием закона исключенного третьего.

Таблицы истинности



Материальная импликация ($x \leq y$) | «Если x, то y'» (буквально) | «Если y', то x'» (буквально) | Трехзначная импликация



Импликация Лукасевича | Импликация Гейтинга | Треугольник Н.Васильева

Разобраться в силлогистике удается не путем формального построения трехзначных и многозначных логик, а осмысленным последовательным развитием двухзначной логики, находящейся в ее основании. В качестве необходимых начал доказательного рассуждения ("без которых невозможно рассуждать") Аристотель принял в сущности принцип двухзначности: "... что относительно чего бы то ни было необходимо или утверждение, или отрицание и что невозможно в одно и то же время быть и не быть". По этому принципу, получившему наиболее совершенное воплощение в булевой алгебре, у Аристотеля посредством терминов определены сущности рассматриваемых вещей ("первые сущности"). "Камешки"-термины предполагаются четко различимыми, дискретными, подобно целым числам. Однозначная характеристика вещи представляет собой элементарную конъюнкцию всех уместных терминов, утверждаемых либо отрицаемых, а неоднозначная, нечеткая, характеристика - дизъюнкцию таких конъюнкций, в частности, попарно склеиваемых.

Другими словами, вещи охарактеризованы *совокупностями* терминов, четкими либо нечеткими. Это 1-я ступень силлогистики, изоморфная классической логике высказываний, булевой алгебре терминов, пополненной теоретико-множественными операциями инверсии, пересечения и объединения выражений. Она позволяет рассуждать о совокупностях терминов, характеризующих единственную и единую рассматриваемую "вещь" (в

широком смысле слова), выявляя отношения, которыми термины связаны друг с другом "в мире этой вещи". Но силлогистика исследует совокупности различных вещей, т.е. совокупности совокупностей терминов. Поэтому в Органоне необходима 2-я ступень, которую естественно и проще всего реализовать, воспользовавшись теми же булевыми связками (конъюнкцией, дизъюнкцией, отрицанием-дополнением, а также инверсией, пересечением и объединением), однако на сей раз применительно не к атрибутам вещей непосредственно, а к охарактеризованным ими вещам и к совокупностям вещей.

В отличие от представленной элементарной конъюнкцией терминов четкой совокупности 1-й ступени, означающей их *совместность* (единство) и *присущность* символизируемых ими определенностей охарактеризованной ею вещи, четкая совокупность (множество) 2-й ступени означает *сосуществование* в ней, или *сопринадлежность* ей различных вещей, сущности которых попарно несовместимы. Такая совокупность представима конъюнкцией существований, в которой существование вещи, охарактеризованной, скажем, атрибутом xy', обозначается *дизъюнктом* $\mathbf{V}xy'$. Например, множество, которому принадлежат сущности xy' и $x'y$, но не принадлежат xy и $x'y'$, обозначаемое в математике как $\{xy', x'y\}$, выражается конъюнкцией дизъюнктов $\mathbf{V}'xy\mathbf{V}xy'\mathbf{V}x'y\mathbf{V}'x'y'$.

Как видно, $\mathbf{V}x$ - конструкция, аналогичная сумме Σx, однако означает не суммирование, а дизъюнкцию значений, принимаемых термином x на элементах рассматриваемого множества (данной совокупности "вещей"). Наглядной моделью совокупности является "урна Лукасевича", в которой элементы представлены шарами, помеченными присущими им терминами. Наличие в урне хотя бы одного шара, помеченного буквой x, есть принадлежность данной совокупности элемента, которому присуще x (который удовлетворяет критерию x), а короче - *принадлежность* x, *существование* x, $\mathbf{V}x = 1$. Урна, в которой x-шаров нет, символизирует антипринадлежность x, $\mathbf{V}x = 0$, $\mathbf{V}'x = 1$. Наличие в урне шаров, не помеченных буквой x, есть принадлежность x', $\mathbf{V}x' = 1$, отсутствие - антипринадлежность x', $\mathbf{V}'x' = 1$. Наличие x-шаров при отсутствии x'-шаров представляет собой совокупность $\mathbf{V}x\mathbf{V}'x'$, которой x необходимо присуще в целом. Урна же, в которой нет ни x-, ни x'-шаров, символизирует пустую совокупность $\mathbf{V}'x\mathbf{V}'x'$.

Принадлежность вещей, охарактеризованных по нескольким критериям, представляется дизъюнктами с соответствующими элементарными конъюнкциями, например: $\mathbf{V}xyz$, $\mathbf{V}xyz'$, $\mathbf{V}xy'z$ и т.д.

В самом простом случае различения по единственному критерию x имеется два вида вещей: x, x' и четыре качественно различных четких совокупности (множества):

$\mathbf{V}x\mathbf{V}x'$ - полная совокупность, ей принадлежат (сопринадлежат) оба вида;

$\mathbf{V}x\mathbf{V}'x'$ - одноэлементная совокупность, принадлежность только вида x, присущность x всем членам совокопности;

$\mathbf{V}'x\mathbf{V}x'$ - принадлежность только вида x';

$\mathbf{V}'x\mathbf{V}'x'$ - пустая совокупность.

Дизъюнкцией этих четырех конъюнкций (они несоисключимы и попарно несовместимы) исчерпывается характеристика произвольной

однокритериальной совокупности вещей, однокритериального общего универсума (УО):

$$\mathbf{V}x\mathbf{V}x' \vee \mathbf{V}x\mathbf{V}'x' \vee \mathbf{V}'x\mathbf{V}x' \vee \mathbf{V}'x\mathbf{V}'x' \equiv 1$$

Это СДНФ-выражение 2-й ступени можно минимизировать, получив ослабленную версию законов исключенного третьего и противоречия:

$$\mathbf{V}x \vee \mathbf{V}'x \equiv 1, \quad \mathbf{V}x\mathbf{V}'x \equiv 0.$$

Исключение из УО тех или иных его частей порождает специальные универсумы, в частности, представляющие особый интерес для упорядочения логики :

непустой (интуиционистский) универсум (УИ) -
$\mathbf{V}x\mathbf{V}x' \vee \mathbf{V}x\mathbf{V}'x' \vee \mathbf{V}'x\mathbf{V}x' \equiv \mathbf{V}x \vee \mathbf{V}x' \equiv 1, \ \mathbf{V}'x\mathbf{V}'x' \equiv 0$;
трехзначный дискретный универсум Поста (УП) -
$\mathbf{V}x\mathbf{V}'x' \vee \mathbf{V}'x\mathbf{V}x' \vee \mathbf{V}'x\mathbf{V}'x' \equiv \mathbf{V}'x \vee \mathbf{V}'x' \equiv 1, \ \mathbf{V}x\mathbf{V}x' \equiv 0$;
двухзначный универсум Хрисиппа-Буля (УБ) -
$\mathbf{V}x\mathbf{V}'x' \vee \mathbf{V}'x\mathbf{V}x' \equiv 1, \ \mathbf{V}x\mathbf{V}x' \vee \mathbf{V}'x\mathbf{V}'x' \equiv 0$;
универсум Аристотеля (УА) -
$\mathbf{V}x\mathbf{V}x' \equiv 1, \ \mathbf{V}'x \vee \mathbf{V}'x' \equiv 0$;
пустой универсум -
$\mathbf{V}'x\mathbf{V}'x' \equiv 1, \ \mathbf{V}x \vee \mathbf{V}x' \equiv 0$.

В приведенном фрагменте иерархии универсумов наглядно отображена иерархия важнейших логик:

в УО имеет место четырехзначная логика, в которой термин x можно охарактеризовать как необходимо присущий - $\mathbf{V}x\mathbf{V}'x'$, необходимо антиприсущий - $\mathbf{V}'x\mathbf{V}x'$, привходящий - $\mathbf{V}x\mathbf{V}x'$, ничего не представляющий, немыслимый - $\mathbf{V}'x\mathbf{V}'x'$;

в УИ устранено пустое - предметная область непуста, логика трехзначна: присуще / привходяще / антиприсуще;

в УП исключено привходящее, осталось три четко различимых состояния - дискретная трехзначная логика;

в УБ исключены пустое и привходящее, осталось два четко различимых состояния - дискретная двухзначная логика;

в УА исключены дискретные состояния, термин определен *сосуществованием противоположностей* $\mathbf{V}x\mathbf{V}x'$, т.е. так как оно и есть в реальности.

С пониманием того, как устроена иерархия логик, открывается возможность конструктивно определить модальные функции и отношения, не изобретая их "по интуиции". Очевидно, что базисной модальной функцией должна быть простейшая и инвариантная по всем универсумам - *актуальная возможность*, или *существование*, т.е. дизъюнкт $\mathbf{V}x$, сущность которого охарактеризована выше. У Льюиса это $\Diamond x$ - *самосовместимость*, и вместе с тем $\exists x$ - *существование*, у Лукасевича: $\mathrm{M}x$ - "*простая*" *возможность* и Σx - *существование*. Ни Льюис, ни Лукасевич не обнаружили, что их модальные функторы $\Diamond$, M означают то же, что и кванторы по терминам-предикатам $\exists$, $\sum$, т.е. что

$$\Diamond x \equiv \exists x \equiv \mathrm{M}x \equiv \sum x \equiv \mathbf{V}x$$

Этой же функцией является *замыкание* $\mathrm{C}x$ в алгебрах с замыканиями. Для нее выполняются тождества:

$$\neg Mx \equiv M'x,\ \ Mx \vee \neg Mx \equiv Mx \vee M'x \equiv 1$$

Посредством нее определимы другие модальные функции и соответствующие кванторы.

Аристотелева *актуальная необходимость* Lx определяется в виде:

$$Lx \equiv \mathbf{V}x\mathbf{V}'x' \equiv MxM'x'$$

В УИ это определение упрощается в $Lx \equiv \mathbf{V}'x' \equiv M'x'$, однако в УО выражение $M'x'$ означает лишь *потенциальную необходимость*, которой соответствует квантор общности $\forall$ по предикатам в его "математическом", не в естественноязыковом, смысле:

$$\forall x \equiv \wedge x \equiv \mathbf{V}'x' \equiv M'x'$$

Модальная функция $Qx \equiv MxMx'$ - *случайность, акцидентальность,* выявляет аристотелево *привходящее* $\mathbf{V}x\mathbf{V}x'$: термин x обладает значением σ (συμβεβηκος), если $Qx \equiv 1$. Символ σ обозначает промежуток между утверждением и антиутверждением: $0 < \sigma < 1$. В теории вероятностей 0 - невозможность, 1 - достоверность, а все прочие значения в совокупности составляют логическое σ. У Буля и у Порецкого имеется процедура *пробабилизации* - перевода булевых выражений по существу на язык нечетких множеств Заде.

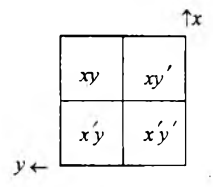
В УБ с устранением привходящего, а вместе с ним и Qx, функции Mx и Lx вырождаются, отождествляясь с их аргументом - двухзначным термином:

$$Mx \equiv \mathbf{V}x \equiv \mathbf{V}x\mathbf{V}'x' \equiv x,\ \ Lx \equiv \mathbf{V}x\mathbf{V}'x' \equiv x$$

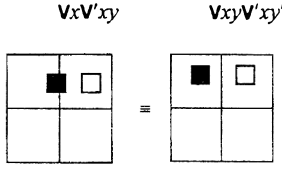
В УА, наоборот, все термины необходимо привходящи, и именно поэтому "строгая импликация" Льюиса $\sim\Diamond(x\sim y)$, т.е. $\mathbf{V}'xy'$, оказывается здесь полноценным содержательным следованием, в точности соответствующим аристотелеву определению ["Первая аналитика", 57b1], в котором Лукасевич усмотрел "ошибочное" с точки зрения современной формальной логики положение. В алгебре совокупностей 2-й ступени это определение представлено выражением $\mathbf{V}x\mathbf{V}'xy'\mathbf{V}y'$, означающим нечеткое множество, которому необходимо принадлежат x-элементы и y'-элементы, и антипринадлежат xy'-элементы, так что всякий его x-элемент непременно есть y-элемент, всякому x присуще y.

Замечательным графическим аналогом алгебры дизъюнктов является диаграмма Льюиса Кэррола, наглядно отображающая совокупности 2-й ступени с не более чем тремя терминами в формате "таблицы истинности", интерпретируемой как разбиение универсума на классы по принятым терминам-критериям.

Диаграммы Льюса Кэррола

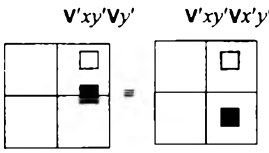


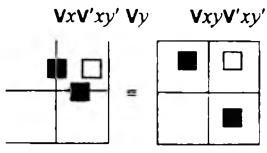
Классы xy-универсума



«Всякое x есть y»

(буквально)





«Всякое y' есть x'» (буквально) «Всякое x есть y» (следование y из x)

Следует заметить, что выражение $\mathbf{V}x\mathbf{V}'xy'\mathbf{V}y'$ представляет собой и общеутвердительное силлогистическое суждение Axy - "Всякое x есть y". Дополнением Axy в УА является частноотрицательное суждение $Oxy \equiv \neg Axy \equiv \mathbf{V}xy'\mathbf{V}x'\mathbf{V}y$, а инверсией термина y получаются общеотрицательное $Exy \equiv Axy' \equiv \mathbf{V}x\mathbf{V}'xy\mathbf{V}y$ и частноутвердительное $Ixy \equiv Oxy' \equiv \mathbf{V}xy\mathbf{V}x'\mathbf{V}y'$. Именно в УА законы подчинения частных посылок общим совместимы с силлогистическим законом исключенного третьего и имеет место треугольник противоположностей Васильева:

$$Axy \vee Exy \vee \neg Axy \neg Exy \equiv Axy \vee Exy \vee OxyIxy \equiv 1 .$$

Непарадоксальная "строгая импликация", аксиоматику которой вводил Льюис, а также Аккерман, представима в алгебре 1-й ступени как общий атрибут элементов совокупности, определяющей аристотелево следование, выражение которой предварительно преобразовано к виду $\mathbf{V}xy\mathbf{V}'xy'\mathbf{V}x'y'$. Искомая импликация оказывается трехзначной функцией двухзначных терминов: $x \to y \equiv xy \vee \sigma x'y \vee x'y'$. Но она, как и ее двухзначный прототип, не может быть характеристической функцией отношения следования, адекватно неотобразимого средствами 1-й ступени.

Ситуация в логике убедительно свидетельствует о том, что и в трех соснах, когда видят два, глядя на три, блуждать можно без конца, причем блуждать научно и изобретательно - виртуальные миры неисчерпаемы и неисчислимы. Но ведь мы живем в единственном реальном мире, и Органон должен упреждать заблуждения в познании и благоустройстве именно этого мира, нашего бытия. Так понималось назначение информатики Аристотелем и его достойными последователями. Логики уклонились от этой цели, и теперь информатика при всей ее технической мощи и "искусственном интеллекте" функции Органона не выполняет. Она не предоставляет нам безупречных методов и инструментов рассуждения, вынуждая полагаться на эмпирику и интуицию, ее положения оказываются неадекватными реальности, несовместимыми со здравым смыслом, с диалектикой. Очевидна, однако, возможность коренного исправления ситуации: ведь не извращенная двухзначниками силлогистика Аристотеля с законом сосуществования противоположностей и с привходящим в качестве третьего-среднего есть та самая *диалектическая логика*, которую днем с огнем ищут современные мудрецы.

Литература

1. Аристотель. Сочинения в четырех томах. - М.: "Мысль", т.1 - 1975, т.2 - 1978.

2. Брусенцов Н.П. Искусство достоверного рассуждения. Неформальная реконструкция аристотелевой силлогистики и булевой математики мысли. - М.: Фонд "Новое тысячелетие", 1998.

3. Брусенцов Н.П., Жоголев Е.А., Маслов С.П., Рамиль Альварес Х. Опыт создания троичных цифровых машин // Компьютеры в Европе. Прошлое, настоящее и будущее. - Киев, "Феникс", 1998. С. 67-71.

4. Brusentsov N.P., Vladimirova Yu.S., Solution of Boolean Equations // Computational mathematics and modeling, Vol.9, No 4, 1998, pp.287-295.

5. Васильев Н.И. Воображаемая логика. Избранные труды. - М.: "Наука,, 1989.

6. Карпенко А.С. Многозначные логики // Логика и компьютер. Вып.4. - М.: "Наука", 1997.

7. Кэррол Льюис. Символическая логика // Льюис Кэррол. История с узелками. - М.: "Мир", 1973.

8. Лукасевич Я. Аристотелевская силлогистика с точки зрения современной формальной логики. - М.: ИЛ, 1959.

9. Пиаже Ж. Логика и психология // Ж.Пиаже. Избранные психологические труды. - М.: "Просвещение", 1969.

10. Порецкий П.С. О способах решения логических равенств и об обратном способе математической логики. - Казань, 1884.

11. Слинин Я.А. Современная модальная логика. - Л.: Изд-во Ленингр. ун-та, 1976.

12. Стяжкин Н.И. Формирование математической логики. - М.: "Наука", 1967.

13. Отчеты по НИР "Развитие конструктно-ориентированного подхода к информатике и компьютерной дидактике" (№ гос. рег. 01.960.009505). - ВНТИЦ, инв. №№ 02.9.80 003626, 02.9.80 005206, 02.20.0000209.

Бахмуров А.Г., Захаров В.А., Смелянский Р.Л., Чистолинов М.В.

О международном проекте в области проверки правильности программного обеспечения встроенных систем

Введение

На сегодняшний день разработка распределенных встроенных систем реального времени все еще остается искусством. Причин тому несколько:

- отсутствие теории, позволяющей описывать и анализировать динамику информационных процессов;
- отсутствие методов и средств, обеспечивающих разработку и проектирование, начиная с системного уровня и кончая проектированием аппаратуры;
- отсутствие развитых методов и средств проверки правильности функционирования системы без ее прототипа.

Сложность разработки подобных систем складывается из сложностей решения следующих задач:

- выбор подходящей архитектуры встроенной системы, обеспечивающей устойчивое и надежное функционирование системы;
- распределение функций между аппаратурой и программой;
- гарантированное выполнение временных ограничений;
- проверка правильности программного обеспечения без построения аппаратного прототипа, включая обеспечение корректности коммуникационных протоколов.

В условиях роста сложности и стоимости объектов, управляемых встроенными системами, возникает необходимость строгого обоснования корректности принимаемых проектных решений. Тестирование не обеспечивает доказательство правильности функционирования, так как практически невозможно протестировать систему на всевозможных наборах данных. Кроме того, желательно обнаруживать ошибки проектирования на возможно более ранних этапах разработки, чтобы избежать бессмысленного перерасхода ресурсов и минимизировать стоимость внесения изменений. Решить все эти проблемы можно только с помощью прозрачной для пользователя технологии, единым образом поддерживающей весь цикл разработки встроенной системы.

Известный способ решения подобных задач - построение виртуальных прототипов, или, что то же самое, имитационных моделей встроенных систем и строгое, формальное доказательство корректности их функционирования. Под комплексным подходом к проектированию мы понимаем возможность проверки в рамках одной среды разработки трех групп свойств проектируемой системы: *алгоритмических свойств*, таких как обеспечение заданной функциональности, наличие тупиков, соблюдение правильных последовательностей событий в протоколах; *временных свойств*, определяемых требованиями реального времени к задержкам отклика, частотам обслуживания управляемых устройств и т.д. и *интегральных свойств*, таких как производительность и надежность всей системы или отдельных ее подсистем. Необходимым условием применимости комплексного подхода является возможность непосредственно перейти от прототипа к реализации системы без внесения изменений, которые могли бы нарушить проверенные свойства.

К сожалению, большинство современных технологий разработки встроенных систем либо не использует формальные методы вообще, либо ориентированы на методы, решающие только часть задач анализа и проектирования. В дан-

ной работе представлен опыт и результаты международного проекта (DrTesy, грант ЕС INCO-COPERNICUS 977020)[1], цели которого состояли в 1) проведении сравнительного анализа подходов и технологий, разрабатываемых партнерами из России, Голландии и Германии на примере проектирования и анализа общего реалистического комплекса из области авионики; 2) в совершенствовании методов и средств партнеров путем добавления лучшего из других подходов; и 3) в разработке концепции интегрированной среды моделирования и анализа распределенных встроенных систем реального времени, решающей большинство задач, возникающих при проектировании подобных систем.

Первый раздел статьи посвящен описанию тестового примера, который исследовался всеми партнерами. Второй раздел излагает развиваемые подходы, полученные результаты, а также сильные и слабые стороны каждого из средств. Третий раздел посвящен концепции интеграции различных подходов и практическим шагам, предпринятым для улучшения каждого из средств партнеров.

1. Бортовой навигационный вычислительный комплекс

В качестве тестовой задачи для проверки и сравнения возможностей средств проектирования и анализа был выбран многопроцессорный вычислительный комплекс бортовой навигационной системы современного самолета. Неформальное описание комплекса было предоставлено Военным Авиационно-Техническим Университетом (ВАТУ) им. Жуковского.

Основная цель навигационной системы (НС) состоит в том, чтобы вывести самолет в заданную точку пространства по заданному направлению. Для этого в навигационном комплексе выполняются три группы задач: обеспечение полета по маршруту (по заданной траектории); коррекция расчетного (предполагаемого) положения самолета по наземным ориентирам; маловысотный полет с огибанием рельефа местности. Эти задачи решаются одновременно в режиме разделения времени.

Аппаратная конфигурация для тестовой задачи состояла из четырех вычислительных узлов, построенных на базе процессоров x86, набор датчиков, общую шину и разделяемую память (со сложной системой приоритетов на входах), к которой имеют доступ два из четырех процессоров. Набор требований к комплексу был определен Государственным Научно-Исследовательским Институтом Авиационных Систем (ГосНИИАС) и включал в себя требования по логике функционирования, директивным срокам и максимально допустимым загрузкам подсистем. Так, шина должна быть загружена не более, чем на 20%, а разделяемая память - не более, чем на 50%. Были сформулированы требования к частотам обновления данных на индикаторе пилота, временам выполнения основных вычислительных функций и скорости выполнения циклограмм на шине. Алгоритмические (или логические) требования определяли понятия корректности протоколов функционирования шины, общей памяти и некоторых других компонентов.

Важно отметить, что в моделируемом примере использовались реальные, бортовые алгоритмы навигации, исследование корректности и производительности которых представляло дополнительно практический интерес.

2. Подходы и инструментальные средства участников проекта

2.1 Пакет mSZ

Пакет mSZ [2] разработан в GMD Forschungszentrum Informationstechnik GmbH (Берлин) и предназначается, в первую очередь, для ясной и однозначной графической спецификации параллельных систем на верхнем уровне абстрак-

ции. Базовым понятием mSZ является процесс, или, точнее, класс процессов. С каждым процессом связано множество данных, интерфейс для определения точек взаимодействия с другими процессами и описание поведения в случае взаимодействия через эти точки. Используемая нотация состоит из трех типов диаграмм, которые отражают структурный, функциональный и поведенческий аспекты системы.

Структурные, или конфигурационные диаграммы определяют, из каких подпроцессов состоит данный процесс и как соединены их точки взаимодействия (порты).

Функциональные диаграммы, или диаграммы данных определяют *типы данных*, множества значений этих типов и возможные операции с данными этих типов. Тип данных определяется набором констант и набором операций над этими константами. Некоторые операции могут порождать новые значения данных для типа возвращаемого значения. С типом данных может быть связан набор эквивалентностей, устанавливающих равенство различных выражений, построенных из констант, переменных и операций. Здесь используется алгебраический способ описания данных, часто встречающийся в системах спецификации и верификации программного обеспечения. Простейший пример такого типа данных - стек с операциями push, pop и эквивалентностью x ~ pop(push(a, x)) (здесь 'x' - переменная типа «стек», а 'a' - переменная типа «элемент стека»).

Диаграммы поведения описываются как диаграммы состояний (statecharts), в которых на переходах проверяются условия и выполняются операции над данными процесса. Процессы взаимодействуют между собой через общие переменные и синхронные сообщения, сопоставленные точкам взаимодействия.

В основе формальной семантики mSZ лежат диаграммы состояний Харела (в их варианте StateMate [3]) и язык спецификации данных Z [4]. Существуют средства, позволяющие проверять спецификацию на непротиворечивость, "прогонять" модель, описанную на mSZ, методами имитационного моделирования и даже строить по этой спецификации полное тестовое покрытие для реализации.

К основным недостаткам этого пакета можно отнести отсутствие средств спецификации времени, отсутствие в свободном доступе средств проверки логических свойств на mSZ, отсутствие явного перехода от спецификации к реализации, коммерческий характер многих утилит работы с пакетом.

Явные преимущества - это наглядность и простота восприятия спецификации, возможность пошаговой детализации модели за счет наследования классов процессов и декомпозиции состояний на вложенные диаграммы состояний.

В качестве других широко распространенных графических нотаций проектирования встроенных систем, ориентированных на диаграммы состояний можно упомянуть методологию ROOM [5], графический язык проектирования UML [6] и язык SDL [7]. Общий их недостаток - отсутствие средств формального доказательства свойств моделей, либо из-за отсутствия формальной семантики соответствующих языков, либо из-за слишком большой выразительной мощности.

В проекте DrTesy средствами mSZ удалось достаточно наглядно и подробно описать структуру алгоритмов навигации и основных коммуникационных компонентов, причем были задокументированы как блочная структура, так и потоки управления (диаграммы переходов) алгоритмов. Однако, совершенно не были отражены временные свойства; автоматически построенные тесты оказались весьма не полны, поскольку они строились по Z-части спецификации, а она в системах, работающих в бесконечном цикле, оказывается достаточно бед-

ной. Фактически, не были проверены даже логические свойства, поскольку программа верификации не является общедоступной.

2.2 Язык mCRL

Система mCRL, соящая из одноименного языка [8] (micro Common Representation Language) и пакет для работы с ним, разработаны в CWI Stichting Mathematisch Centrum (Амстердам). Назначение системы - спецификация и верификация (проверка свойств) поведения взаимодействующих процессов, которые работают в режиме чередования. Это направление представляет алгебраический подход в формальном анализе распределенных систем, поскольку формальная семантика mCRL базируется на Алгебре Процессов (Process Algebra) [8]. Процессы взаимодействуют с помощью попарно-"комплементарных" действий: действие и парное ему ко-действие, которые для простоты можно представлять себе как действия посылки и приема сообщения в синхронной модели взаимодействий (рандеву). Возможно также описание типов данных подобно языкам Z и mSZ, хотя и со своими особенностями.

Основные преимущества алгебраического подхода - удобно и естественно описываются протоколы событий; выразительная мощность легко варьируется за счет использования или не использования сложных типов данных. Для языка mCRL очень хорошо проработана формальная семантика, что позволяет легко развивать систему и интегрировать с другими средствами работы со спецификациями; язык чрезвычайно компактен; есть очень хорошие перспективы в области формальной проверки свойств на моделях с алгебраической семантикой. За счет интеграции с системой Aldebaran [13] удалось увеличить сложность верифицируемых систем.

Основные недостатки - отсутствие в текущей версии mCRL времени, "оторванность" от средств реализации программного обеспечения, бедность интерфейса пользователя, принципиальная невозможность задавать приоритеты процессам, недостаточный уровень редукций при проверке свойств (в текущей версии); в языке отсутствуют средства пошаговой детализации модели; инструменты пакета плохо документированы. Существуют и другие алгебраические языки спецификаций, например, Lotos [9], к которым можно отнести почти все преимущества и недостатки mCRL.

В проекте DrTesy на mCRL были специфицированы только протоколы шины и ее окружения. Одна из причин этого - невозможность моделировать схемы арбитража, связанные с различным приоритетом процессов, другая - достаточно большая дистанция между конструкциями mCRL и примитивами вычислительной системы. В ходе проверки логических свойств была найдена ошибка в неформальном описании циклограмм шины.

2.3 Среда моделирования DYANA

Среда имитационного моделирования DYANA [10] (DYnamic ANAlyzer) разработана в Лаборатории Вычислительных Комплексов факультета ВМиК МГУ им. М.В. Ломоносова (Москва). Среда создавалась специально для моделирования распределенных вычислительных систем таким образом, чтобы по одному и тому же описанию комплекса проводить как формальный, логический анализ свойств, так и оценку производительности методами имитационного моделирования. Важно отметить, что в основу системы была положена специально разработанная формальная модель функционирования распределенных вычислительных систем [11], которая обеспечила концептуальное единство среды, возможность работать с моделями программного обеспечения и аппаратуры не-

зависимо, явно учитывать время, гибко перераспределять компоненты программного обеспечения в аппаратной среде без их переделки, плавно переходить от модели программ к их реализации [10].

Для описания моделей распределенных вычислительных систем в среде DYANA был разработан язык моделирования MM (Modelling with Messages). В языке явно выделяются программная и аппаратная составляющие встроенной системы. Программа определяет логику поведения системы, а аппаратура - привязку действий программы к модельному времени и последовательность выполнения независимых процессов. Так, процессы, привязанные к одному последовательному исполнителю, обязаны выполняться в режиме чередования, а привязанные к разным - в режиме истинного параллелизма (real concurrency).

Распределенная программа представляется набором последовательных процессов и других распределенных программ, взаимодействующих друг с другом путем отправки и приема сообщений по каналам связи через буфера (почтовые ящики). Посылка сообщений асинхронна, так что процесс может приостановить свое выполнение только при запросе сообщения из пустого буфера. Набор типов сообщений в каждой модели конечен, причем типы могут образовывать иерархию, и в языке есть операторы проверки типов сообщений.

Тело MM-процесса включает в себя управляющие конструкции, типичные для алгоритмических языков, операторы межпроцессного взаимодействия и операторы продвижения модельного времени. Также, в тело процесса можно вставлять фрагменты на языке Си, что позволяет с одной стороны непосредственно переходить от модели к параллельной программе, а с другой - точно оценивать время выполнения фрагмента кода на заданном исполнителе.

Кроме MM-языка в среду DYANA входят несколько подсистем. *Подсистема логического анализа* извлекает из MM-языка информацию об обмене сообщениями и строит конечную систему переходов, которая используется для проверки выполнимости свойств на модели на основе техники Model Checking [12].

Подсистема оценки времени среды DYANA предсказывает время выполнения линейных участков Си-кода на заданном исполнителе без проведения эмуляции. Это позволяет существенно сократить время прогона имитационной модели.

Подсистемы оценки производительности и *визуализации* позволяют собирать, анализировать и просматривать данные о прогонах имитационной модели. Эти подсистемы позволяют проверить выполнение временных требований и удовлетворительность показателей производительности.

Существуют также средства, для графического описания структуры модели и отладки в терминах языка моделирования.

Среда DYANA – единственный инструмент из использованных в проекте, который позволяет эффективно описывать и проверять временные аспекты поведения исследуемой системы на разных уровнях детализации, вплоть до программной реализации. Важное достоинство среды - наличие формальной модели, описывающей совместное функционирование программы и аппаратных средств.

Основные недостатки среды - отсутствие графической нотации для наиболее содержательных элементов описания - тел MM-процессов, недостаточные возможности для практического использования пошаговой детализации и перехода к реализации, относительная бедность "наблюдаемых" для верификации типов данных.

В рамках проекта DrTesy в среде DYANA была построена полная имитационная модель навигационного комплекса, которая включала в себя численные алгоритмы на языке C и решала большинство задач комплекса. Модель работала в реальном масштабе времени на платформах Sun Ultra Enterprise 450 и Intel Pentium II 333 МГц. Часть логических свойств была доказана подсистемой логического анализа, прочие свойства проверялись на результатах прогонов.

3. Интеграция средств

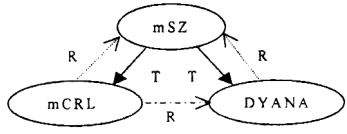
Как видно из обзора средств и полученных с их помощью результатов, ни один из подходов не имеет явного преимущества сразу по всем направлениям использования.

Так, графические средства спецификаций типа mSZ, основанные на иерархических диаграммах состояний обладают ясной и привычной для пользователя семантикой, хорошо соотносятся с принципами построения систем в некоторых предметных областях, например, в области проектирования телекоммуникационных систем, способствуют пошаговой детализации и автоматическому построению тестовых наборов.

Языки спецификаций, основанные на алгебрах взаимодействующих процессов, например mCRL и Lotos [9] могут обеспечивать высокую эффективность при проверке свойств и хорошо подходят для описания протоколов. Такие языки намного проще, чем используемые на практике языки диаграмм состояний, поэтому нет, например, никаких проблем в написании для них всевозможных конверторов и синтаксических анализаторов.

Современные языки имитационного моделирования, как правило, лучше приспособлены для работы со временем и больше приближены к конкретной предметной области, чем языки спецификаций, возникшие на базе каких-либо математических моделей. В языках моделирования гораздо проще организовать отладку модели или переход от модели к реализации. Здесь для однозначного анализа необходима формальная семантика языка.

В рамках проекта DrTesy была предложена концепция интеграции средств участников проекта [1], сохраняющая преимущества отдельных подходов:



Согласно предложенной концепции, диаграммы состояний (mSZ) используются для спецификации системы на верхнем уровне абстракции и для генерации по этой спецификации тестов (стрелки T). Прочие средства моделирования (mCRL и DYANA) должны быть способны реализовывать диаграммы состояний (отношение R). При таком подходе диаграммы состояний как бы являются спецификацией более детальных алгебраических и имитационных моделей. С другой стороны, имитационная модель полезна при отладке распределенных алгоритмов и для оценки производительности системы, но для доказательства логических свойств лучше построить реализацию этой модели на языке с алгебраической семантикой и воспользоваться средствами редукции системы переходов. В рамках проекта были описаны схемы преобразования спецификаций на mSZ и MM-языке в mCRL, а для последнего случая даже реализован транслятор.

4. Заключение

В рамках проекта DrTESY (EC INCO-Copernicus) были получены следующие основные результаты:

1. Построен пример бортового навигационного комплекса, который может быть использован для целей исследования разных инструментальных средств для построения и проверки правильности встроенных систем.
2. Исследованы разные подходы к описанию, спецификации, проверке правильности программного обеспечения программного обеспечения встроенных систем.
3. Создана первая отечественная рабочая версия системы спецификации и верификации поведения встроенных систем, позволяющая работать со встроенными системами, которые описываются диаграммами переходов с числом состояний до 10^{15}.
4. Создана формальная семантика MM-языка среды DYANA.
5. Сделан сравнительный анализ подходов к проверке правильности программного обеспечения на основе mSZ, mCRL и DYANA. Осознаны ограничения и возможности каждого подхода.
6. Разработаны подходы, позволяющие интегрировать эти инструментальные средства.
7. Разработан проект отраслевого стандарта инструментальных средств для анализа и проектирования программно-аппаратных систем реального времени.

Основной целью этого проекта был сравнительный анализ и взаимовыгодное развитие научных разработок, направленных на спецификацию, анализ и проектирование современных встроенных систем реального времени. Авторы предложили концепцию комплексного подхода к проектированию встроенных систем, основанную на объединении достижений теоретических направлений и инструментов, развиваемых различными партнерами.

ЛИТЕРАТУРА

1. DrTesy - Methods and Tools for Distributed Real Time Embedded Systems, Design and Analysis, EC INCO-Copernicus, проект 977 020 http://www.first.gmd.de/~drtesy
2. Robert Geisler, "Formal Semantics for the Integration of Statecharts and Z in a Metamodel-Based Framework", Doctoral Tesis at Technical University of Berlin, January 1999.
3. David Harel and Mihael Politi, "Modeling reactive systems with statecharts: The statemate approach", i-Logix Inc, Three Riverside Drive, Andover, MA 01810, USA, June 1996.
4. J. M. Spivey, "The Z Notation: A Reference Manual", Prentice Hall International Series in Computer Science, 2nd edition, 1992.
5. Selic B., Gullekson G. and Ward P., "Real-Time Object-Oriented Modeling", John Wiley & Sons, New York, NY, 1994.
6. Г. Буч, Д. Рамбо, А. Джекобсон, "Язык UML. Руководство пользователя", М.: МДК - 2000.
7. Карабегов А.В., Тер-Микаэлян Т.М., "Введение в Язык SDL", М.: Радио и Связь, 1993.
8. J. F. Groote and A. Ponse, "The syntax and semantics of mCRL". In A. Ponse, C. Verhoef, and S.F.M van Vlijmen, editors, Algebra of Communicating Processes, 1994, pages 26-62. Workshop in Computing Series, Springer-Verlag, 1995.
9. ISO DP 8807, "A formal description of technique based on the temporal ordering of observation behavior", 1984.
10. A.G. Bahmurov, A.P. Kapitonova, R.L. Smeliansky, "DYANA: An Environment for Embedded System Design And Analysis", Proceedings of TACAS'99, Amsterdam, March 22-26, p. 390-404.
11. Смелянский Р.Л. Модель функционирования распределенных вычислительных систем, Вестн. Моск. Ун-та. сер 15, Вычисл. Математика и Кибернетика. 1990, №. 3, стр. 3-21.
12. Захаров В.А., Царьков Д.В. Эффективные алгоритмы проверки выполнимости формул темпоральной логики CTL на модели и их применение для верификации параллельных программ //Программирование, 1998, №4, с.3-18.
13. CADP (Caesar/Aldebaran Development Package, http://www.inrialpes.fr/vasy/cadp

Раздел II
Имитационное моделирование и синтез систем обработки и передачи информации

Костенко В.А.

Задачи синтеза архитектур: формализация, особенности и возможности различных методов для их решения.

Введение.

В данной работе рассматривается проблема синтеза архитектур вычислительных систем (ВС) под заданное приложение: формальные модели описания функционирования ВС с MIMD архитектурой; сведение задачи синтеза архитектур, в рамках предложенного формализма, к классу смешанных многопараметрических и многокритериальных экстремальных задач с ограничениями; основные особенности сформулированной задачи; проводится анализ возможностей использования существующих методов решения подобных задач; рассмотрен ряд методов решения задач синтеза архитектур, учитывающих особенности рассматриваемой предметной области, и приведены результаты их исследования.

Разработанные средства и методы могут быть использованы на этапах концептуального, структурного и функционального проектирования многопроцессорных вычислительных систем с MIMD архитектурой.

1. Формализация задачи синтеза архитектур и ее особенности.

Под архитектурой ВС будем понимать совокупность аппаратных средств ВС (структура ВС) и системного программного обеспечения (логическая среда ВС), обеспечивающего функционирование аппаратных средств. Под прикладной программой будем понимать программу, выполняемую на ВС с использованием требуемых аппаратных средств и системного программного обеспечения ВС. Ограничим класс рассматриваемых архитектур MIMD системами. То есть, в ВС имеются процессоры способные работать по индивидуальной перезагружаемой программе. Какие-либо ограничения на коммутационную среду, память и способ взаимодействия не накладываются.

Задача синтеза архитектур ВС под заданное приложение в самом общем виде может быть поставлена следующим образом. Для заданного поведения программы требуется синтезировать архитектуру ВС и построить расписание выполнения программы на полученной архитектуре. При этом должны оптимизироваться критерии оценки качества решения и выполняться ограничения на допустимые решения. Поведение программы задается в инвариантной форме относительно архитектуры ВС. В качестве параметров оптимизации (управляемых переменных) выступают варьируемые параметры архитектуры ВС и расписания выполнения программы. Критерии оценки качества решения, ограничения на допустимые решения и варьируемые параметры определяются при конкретизации задачи синтеза архитектур.

Формальные модели описания функционирования ВС, предложенные в [1], позволяют определить структуру ВС и программы как конструктивные объекты, композируемые/декомпозируемые из элементарных параметризованных компонентов, атомарных для выбранного уровня детализации. Задача синтеза

архитектур ВС при использовании разработанного в [1] формализма может быть сформулирована как смешанная многопараметрическая и многокритериальная экстремальная задача с ограничениями:

$f_i(HP,HW,\varphi) \rightarrow min/max,\ i \in I_1,I_2;$

$g_i(HP,HW,\varphi) \le 0,\ i \in I_3,I_4;$

$g_i(HP,HW,\varphi) = 0,\ i \in I_5,I_6;$

$HP \in HP^*,\ HW \in HW^*;$

где HP - модель расписания выполнения программы, HW - модель архитектуры ВС, φ - оператор интерпретации (задает взаимодействие между компонентами из HP,HW), f_i - критерии оценки качества решения, g_i - ограничения на допустимые решения, I_1, I_3, I_5 - множества динамических критериев и ограничений, I_2, I_4, I_6 - множества статических критериев и ограничений, (HP^*,HW^*) - пространство решений. Оценки значений динамических критериев и ограничений из множеств I_1, I_3, I_5 для конкретного варианта решения задачи (HP_i,HW_i) могут быть получены лишь после выполнения оператора интерпретации φ; оценки значений статических критериев и ограничений из множеств I_2, I_4, I_6 могут быть вычислены непосредственно по моделям HP и HW без выполнения интерпретации.

Модель поведения программы задается историей выполнения программы H и набором временных ограничений. H представляет собой ациклический ориентированный размеченный граф: $H=(P,\prec)$. Вершинам $P = \{p_i\}_{i=1}^{N_1} \cup \{p_i\}_{i=1}^{N_2} \cup \ldots \cup \{p_i\}_{i=1}^{N_K}$ соответствуют рабочие интервалы процессов [1], дугам $\prec = \{\prec_{ik} = (p_i,p_k)\}_{(i,k) \in (1\ldots N)}$ - связи, определяющие взаимодействия между рабочими интервалами из множества P. Где N_i - число рабочих интервалов в процессе p_i, K - число процессов в программе PR, $N=N_1+N_2+\ldots+N_K$ - мощность множества P. Рабочие интервалы процесса определяются точками взаимодействия процесса с другими процессами. Все рабочие интервалы одного процесса должны быть назначены на один и тот же процессор. Чередование рабочих интервалов различных процессов, назначенных на один и тот же процессор, допустимо, если не нарушается частичный порядок, заданный $\prec$. Отношение $\prec_{ik}$ представляется следующим образом: если $p_i \prec_{ik} p_k$, то рабочий интервал p_i, необходимо выполнить до начала выполнения рабочего интервала p_k. Каждая вершина имеет свой уникальный номер и метки: принадлежности рабочего интервала к процессу и вычислительной сложности рабочего интервала. Вычислительная сложность рабочего интервала позволяет оценить время выполнения рабочего интервала на процессоре. Дуга определяется номерами смежных вершин и имеет метку, соответствующую объему данных обмена. Объем данных обмена для каждой связи из $\prec$ позволяет оценить затраты времени на выполнение внешнего взаимодействия при условии, что разделяемые ресурсы, требуемые для выполнения взаимодействия, в момент обращения свободны.

Модель расписания выполнения программы определим набором простых цепей и секущих ребер: $HP = (\{SP_i\}_{i=(1\ldots M)}, \prec_c)$, где $\{SP_i\}_{i=(1\ldots M)}$ -набор простых цепей, $\prec_c$ - набор секущих ребер. Простые цепи (ветви параллельной программы) образуются рабочими интервалами процессов, распределенными на один и тот же процессор, секущие ребра определяются связями процессов, распределенными на разные процессы. Если рабочие интервалы p_i и p_j распределены на разные процессоры и в $\prec$ существует связь $\prec_{ij}$, то она

определяет секущее ребро в *HP*. В модели *HP* сохраняются нумерация вершин, дуг и их метки заданные в модели поведения прикладной программы *H*.

Модель аппаратных средств ВС [1] задается ориентированным размеченным графом $HW=(E,L)$. Вершинам $e_i \in E$ графа *HW* соответствуют исполнители, дугам $l_{ij} \in L$ - каналы связи между исполнителями. Исполнитель обладает некоторым ресурсом: способностью выполнять заданный набор действий с определенным временем выполнения каждого действия. Канал связи может лишь обеспечивать передачу данных с некоторой задержкой. Вершины и дуги графа *HW* размечаются в соответствии с характеристиками соответствующих им исполнителей и каналов связи. В качестве меток вершин могут также использоваться функции/операторы, описывающие механизм функционирования исполнителя. Множество исполнителей разобьем на два класса: $E = EP \cup EAS = \{ep_i\} \cup \{eac_i\}$, где $\{ep_i\}$ - множество процессоров в ВС, $\{eac_i\}$ - множество разделяемых ресурсов в ВС. Такое разделение исполнителей на два класса связано с особенностями оценки времени выполнения рабочих интервалов и внешних взаимодействий рабочих интервалов. Наличие в ВС разделяемых исполнителей делает невозможным локальное получение точной оценки времени выполнения внешних взаимодействий для отдельных фрагментов расписания, т.к. время получения ресурса рабочим интервалом может зависить как от ранее поступивших заявок от других рабочих интервалов, так и от заявок, поступивших позже.

Основные подклассы данной задачи будем различать по следующим характеристикам: 1)способу задания фиксированных численных параметров моделей (скалярное число, интервал, случайное число); 2)типу модели прикладной программы (информационный граф программы, информационный граф истории поведения программы); 3)типу модели аппаратных средств (в архитектуре ВС допустимы и не допустимы разделяемые ресурсы); 4)типу оптимизируемых параметров (оптимизируемые параметры принадлежат: подмножеству множества действительных чисел, множества натуральных чисел, множества функций/операторов, множества комбинаторных структур).

Данная задача независимо от варианта постановки обладает следующими основными особенностями:

1. Пространство возможных решений компонентно разнородное, т.е. различные оптимизируемые параметры могут принадлежать подмножествам множества: действительных чисел, целых чисел, функций и комбинаторных структур. Разнородность пространства не позволяет выбрать единую метрику.
2. f_i, g_i, φ - операторы, заданные правилами/алгоритмами их вычисления, т.е. их аналитическая структура не может быть использована для организации поиска оптимального решения *(HP', HW')*.
3. Негладкий характер функций f,g_i.
4. Отсутствие информации о производных функций f,g_i или их производные не являются непрерывными, что не позволяет найти вектор характеризующий их убывание.
5. Значения динамических критериев и ограничений для конкретного варианта решения задачи могут быть определены лишь после выполнения оператора интерпретации φ. Для выполнения интерпретации независимо от конкретного варианта постановки задачи всегда должна быть задана модель расписания выполнения программы *HP*.

Задача синтеза архитектур ВС является сложной задачей оптимизации с ограничениями. Под сложной задачей оптимизации с ограничениями будем по-

нимать задачи для которых отсутствует априорная информация о функциях f_i, g_i и пространстве решений (HP^*, HW^*), которая может быть использована для организации поиска оптимального решения, или сложность получения этой информации неприемлема. Данные особенности задачи делают невозможным эффективное применение классических методов оптимизации, использующих некоторые априорно известные свойства для организации поиска оптимального решения.

В следующих разделах рассмотрим следующие методы решения задачи синтеза архитектур ВС:

- жадные алгоритмы для решения задачи построения расписания, подзадачи, присутствующей в любом варианте постановки задачи синтеза архитектур;
- методы слияния для решения практически важного подкласса задач синтеза архитектур: минимизации аппаратных ресурсов ВС;
- генетические и эволюционные алгоритмы, как универсальные методы настраиваемые на любой конкретный вариант постановки задачи.

2. Жадные алгоритмы построения расписания.

Будем рассматривать задачу составления расписаний в следующей постановке: для *заданных* архитектуры ВС (*HW*) и поведения программы (*H*), *требуется* построить расписание (*HP*) таким образом, чтобы *обеспечить* минимально возможное время выполнения программы. Данная задача относится к классу задач комбинаторной оптимизации и является *NP*-полной.

Введем следующие ограничения: 1)любой процесс из множества *P* содержит лишь один рабочий интервал (в дальнейшем в данном разделе вместо термина рабочий интервал будем использовать термин процесс); 2)система однородна по типу процессоров и каналов связи, то есть выполнение определенного процесса на любом процессоре ВС занимает одно и тоже время и для любого канала связи время передачи одного слова одно и тоже.

Предложенные в работах [2,3] принципы построения жадных алгоритмов составления расписаний, обеспечивают их линейную сложность относительно числа планируемых процессов:

1. Набор процессов, подлежащих планированию, обходится однократно. После выбора очередного процесса, алгоритм распределяет его на один из процессоров ВС и больше к этому процессу не возвращается.

2. Процесс помещается в расписание таким образом, чтобы не изменить времена инициализации уже распределенных процессов.

3. Решение о распределении очередного процесса принимается исходя из принципа получения оптимального расписания на текущем шаге при условии, что расписание, полученное на предыдущем шаге, не может быть измененно. То есть, на каждом шаге алгоритм делает выбор, оптимальный с точки зрения получения частичных расписаний, предполагая, что эти локально-оптимальные решения приведут к приемлемому решению задачи.

Схематично алгоритм можно представить следующим образом [2,3]:

1. Присвоение длинам всех ветвей (времена выполнения) нулевого значения: $T_i=0$, $i=(1,\ldots,M)$.

2. Выбор очередного процесса для распределения: j-й процесс, $j\in(1,\ldots,N)$.

3. Выбор множества мест возможного размещения j-го процесса: M0.

4. Выбор, из множества возможных мест M0, одного места для размещения j-го процесса: i-ый процессор i∈(1,...,M).

5. Коррекция T_i и SP_i.

6. Из графа H выбраны все процессы - завершение работы алгоритма, в противном - переход к п.2.

Выбор очередного процесса для распределения осуществляется в порядке возрастания их номеров в графе *H*. Отметим, что нумерация вершин в графе *H* удовлетворяет условиям полной топологической сортировки. Полная топологическая сортировка всегда существует, т.к. граф *H* ациклический.

Выбор множества мест возможного размещения очередного процесса может осуществляется следующими способами:

1. Процесс может быть размещен в конец одного из списков SP_i, в которых находится один из непосредственных его предшественников *(SIMPLE)*.

2. Процесс может быть размещен в конец любого из списков SP_i , $i∈(1,...,M)$ *(COMPLEX)*.

3. Процесс может быть размещен в конец любого из списков SP_i , $i∈(1,...,M)$ или перед любым процессом этих списков, для которого время ожидания данных превышает время выполнен очередного процесса (то есть в места, где возникают простои процессоров из-за ожидания данных от других процессоров) *(COMPLEX_HOL)*.

В соответствии со способом выбора мест возможного размещения очередного процесса в дальнейшем будем выделять три варианта построения алгоритмов: *SIMPLE, COMPLEX* и *COMPLEX_HOL*.

Выбор, из множества возможных мест *M0*, одного места для размещения текущего процесса заключается в последовательном сужении множества возможных мест. На каждом шаге применяется один из критериев [2,3] для оценки возможных мест постановки процесса и множество *M0* сужается до множества мест, на котором достигается минимальное значение критерия. Если после последовательного применения всех локальных критериев множество *M0* не удалось сузить до одного места, то из оставшихся мест выбирается место с наименьшим порядковым номером. При размещении процессов, не имеющих предшественников, выбор места постановки осуществляется лишь по критерию "равенство длин ветвей".

Алгоритмы могут настраиваться на тип графа *H*, класс архитектуры ВС и критерии оценки качества расписания:

- выбором набора критериев и порядком их использования;
- допустимыми местами постановки очередной анализируемой вершины.

Исследование алгоритмов проводилось для полносвязных архитектур и архитектур с общей шиной на различных типах графов *H* (с различной структурой связи между вершинами [9]).

Получаемое алгоритмами расписание для полносвязных архитектур, по времени его выполнения, никогда не превосходит получаемое расписание для архитектур с общей шиной при одинаковом числе процессоров. Причем, разница времен выполнения расписаний имеет тенденцию к увеличению с ростом числа процессоров. То есть, данные алгоритмы не подвержены аномалиям по типу архитектуры: получение более хороших по времени выполнения расписаний для архитектур с меньшей пропускной способностью коммутационной среды.

Алгоритм *SIMPLE* более эффективен для графов, представляющих собой набор цепочек процессов или близких по структуре связей к ним. Для других типов графов эффективней алгоритмы *COMPLEX* и *COMPLEX_HOL*. Разница между временем выполнения расписаний получаемых алгоритмами *COMPLEX_HOL* и *COMPLEX* может достигать 8%. Особенно ярко выражено преимущество алгоритма *COMPLEX_HOL* для графов являющихся деревьями.

Набор и порядок используемых критериев для выбора одного из возможных мест размещения процесса не оказывает существенного влияния в случае полносвязных архитектур: время выполнения получаемых расписаний при использовании в алгоритме различных наборов критериев отличается не более чем на 1%. Однако, в случае архитектур с общей шиной отличие времен выполнения получаемых расписаний может достигать 40%. Для архитектур с общей шиной оптимального набора и порядка использования критериев единого для всех тестов обнаружить не удалось.

3. Методы слияния для решения задач минимизации аппаратных ресурсов ВС.

Принципы построения методов слияния рассмотрим для следующего варианта постановки задачи минимизации аппаратных ресурсов ВС: для *заданных* поведения программы (*H*) и директивного срока ее выполнения (T^{dir}), *требуется* выбрать минимально достаточное число процессов в ВС (*M*) и получить расписание выполнения программы (*HP*), чтобы *обеспечить* время выполнения программы за директивный срок. Задача синтеза архитектур ВС в данной постановке относится к классу смешанных задач комбинаторной и целочисленной оптимизации: оптимизируемый параметр "число процессоров" принадлежит множеству целых чисел, параметр "расписание выполнения программы" - множеству комбинаторных структур.

Ограничим класс архитектур полносвязными однородными архитектурами, т. е. *HW* представляет собой полносвязный граф, вершинами которого являются процессоры, дуги – связи между ними. Данные ограничения позволяют задавать вычислительную сложность рабочего интервала временем его выполнения на процессоре и включать в это время затраты на взаимодействие с рабочими интервалами других процессов.

Известен качественный закон изменения времени выполнения *T* программы от числа процессоров *M* в ВС (рис.1)- с увеличением числа процессоров время выполнения уменьшается. Когда число процессоров, превышает некоторый предел M^* время выполнения не изменяется или даже увеличивается.

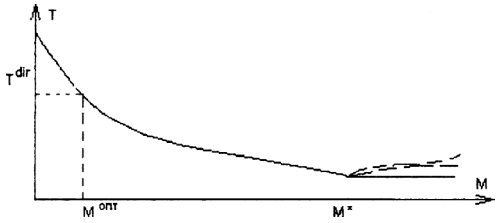


Рис.1. Зависимость времени выполнения программы от числа процессоров.

Функцию $T = f(M)$ на интервале $[1, M^*]$ мы можем считать невозрастающей при условии, что для каждого значения M расписание оптимально. Полагая, что время выполнения программы равное директивному сроку ($T = T^{dir}$) достигается в интервале $[1, M^*]$, можно построить алгоритм, основанный на пошаговой минимизации числа процессоров.

Принципы построения алгоритма слияния для данного варианта постановки задачи синтеза архитектур при данных ограничениях были предложены в работе [4]. Алгоритм, основан на пошаговом переходе от максимальной по числу процессоров ВС к минимальной. На первом шаге строится ВС с максимально возможным числом процессоров - каждый процесс программы назначается на свой процессор. Поскольку архитектура полносвязная, то в дальнейшем вместо процессоров будем рассматривать ассоциации процессов. Ассоциация процессов (SP_j) - упорядоченное множество рабочих интервалов процессов, назначенных на один и тот же процессор. На каждом последующем шаге происходит просмотр всех возможных пар ассоциаций, среди них выбирается наилучшая пара для объединения, проверяется выполнение временных ограничений и выполняется коррекция числа ассоциаций. При пересечении времен выполнения рабочих интервалов ассоциации осуществляется их сдвиг таким образом, чтобы времена выполнения рабочих интервалов не пересекались. После объединения пары ассоциаций, количество ассоциаций будет уменьшено на единицу. На некотором шаге будет невозможно произвести объединение какой-либо пары ассоциаций без нарушения директивного срока или останется лишь одна ассоциация. На этом алгоритм заканчивает свою работу. Схематично данный алгоритм можно представить:

1. Начальная инициализация.

2. Выбор пары ассоциаций - кандидатов на объединение:

- *оценка значений пошаговых критериев оптимизации,*
- *проверка выполнения временных ограничений.*

3. Коррекция числа ассоциаций.

4. Нет ассоциаций, для которых возможно обьединение: завершение алгоритма.

Операция начальной инициализации. Результатом выполнения операции является конфигурация исходной ВС: число ассоциаций равно числу процессов (в каждой ассоциации определены порядок выполнения и времена инициализации рабочих интервалов).

Операция выбора кандидатов на объединение. Результатом операции являются номера ассоциаций - кандидатов на обьединение. Для оценки кандидатов на обьединение введен набор пошаговых (локальных) критериев оценки качества решения [5], получаемого при обьединении двух ассоциаций. Выбор из множества возможных кандидатов на объединение одной пары заключается в последовательном сужении множества возможных пар. На каждом шаге применяется один из критериев для оценки возможных кандидатов и множество возможных пар сужается до множества пар, на котором достигается минимальное/максимальное значение критерия. Если после последовательного применения всех критериев, множество возможных пар не удалось сузить до одной, то пара для объединения выбирается из них случайным образом.

Операция слияния. Результатом выполнения операции слияния является новый набор ассоциаций (на 1 меньший прежнего) и оценка времени его выполнения. Данная операция применяется при оценке значений динамических пошаговых критериев и проверке временных ограничений. Операция слияния для выбранных двух ассоциаций $SP_a=\{p_{a1}, p_{a2}, \dots p_{ak}\}$ и $SP_b=\{p_{b1}, p_{b2}, \dots p_{bm}\}$ строит множество $SP_c=\{p_{c1}, p_{c2}, \dots p_{c\,k+m}\}$ для новой ассоциации SP_c. При этом требуется скорректировать времена инициализации всех рабочих интервалов программы так, чтобы не было пересечения времен выполнения рабочих интервалов в каждой ассоциации. Различные варианты построения множества SP_c рассмотрены в работе [5].

Операция коррекции числа ассоциаций. Количество ассоциаций уменьшается на единицу, рабочие интервалы двух ассоциаций объединяются в одну (используется операция слияния).

Различные модификации алгоритма могут быть получены: 1)использованием различных операций слияния; 2)изменением набора и/или порядка применения используемых критериев пошаговой оптимизации.

Вычислительные эксперименты по синтезу полносвязных архитектур показали высокое качество решений, получаемых алгоритмом, и возможность уменьшения вычислительной сложности алгоритма при сохранении качества получаемых решений следующим способом. На заданном числе начальных шагов пара для обьединения выбирается случайным образом, что исключает просмотр всех возможных пар ассоциаций на первых шагах, где их количество велико.

Тип операции слияния оказывает влияние на качество получаемого решения. Однако, это влияние не столь значительно по сравнению с влиянием набора и порядка применения пошаговых критериев оптимизации. Недостатком данного подхода является сильная зависимость качества получаемых решений от набора и порядка использования пошаговых критериев оптимизации: получаемое число процессоров в ВС может отличаться в 2.5 раза. Случайный выбор пар для слияния на начальных шагах позволяет уменьшить вычислительную сложность, при сохранении качества получаемого решения. Для плохого набора и порядка критериев, случайный выбор пар на начальных шагах позволяет даже значительно улучшить решение.

Предложенный подход (без каких либо существенных модификаций) может быть использован для минимизации любого ресурса ВС, если функция зависимости динамических критериев оптимизации от количества ресурса не возрастающая в области ожидаемого решения. Алгоритм может быть настроен на

тип графа H, класс архитектуры ВС, критерии оценки качества решения и ограничения:

- выбором набора пошаговых критериев оптимизации и способом их использования;
- выбором варианта выполнения операции слияния, а именно способом организации сдвигов рабочих интервалов.

4. Генетические алгоритмы для решения задач синтеза архитектур ВС.

Идея о целесообразности случайного поведения, при наличии неопределенности, т.е. отсутствии достаточной информации, которая может быть использована для организации поиска оптимального решения/поведения, впервые в четкой форме была сформулирована У.Р. Эшби в работе [6] и реализована в известном гомеостате (гомеостат Эшби). Применительно к сложным задачам оптимизации, алгоритм поиска оптимального решения должен опираться на метод проб и ошибок. Только такой процесс позволяет извлечь информацию, необходимую для организации поиска решения. Метод проб и ошибок основан на понятии случайного эксперимента: случайного выбора значений оптимизируемых параметров, что дает возможность получить информацию о функциях $f,g,$ и пространстве решений (HP^*, HW^*), которая может быть использована для организации поиска решения сложной задачи оптимизации с ограничениями. Генетические алгоритмы [7] относятся к алгоритмам поиска, опирающимся на метод проб и ошибок, что позволяет использовать их как универсальные методы, настраиваемые на любой конкретный вариант постановки задачи синтеза архитектур ВС.

Используемый для решения задач синтеза архитектур ВС генетический алгоритм схематично можно представить следующим образом:

1. *Сгенерировать случайным образом начальную популяцию.*
2. *Вычислить целевую функцию для каждой строки популяции.*
3. *Выполнить операцию селекции.*
4. *Выполнить операцию скрещивания.*
5. *Выполнить операцию мутации.*
6. *Если критерий останова не достигнут перейти к п.2, иначе завершить работу.*

Популяция - это множество битовых строк. Каждая битовая строка представляет в закодированном виде одно из возможных решений задачи. По битовой строке может быть вычислена целевая функция, которая характеризует качество решения. В качестве начальной популяции может быть использован произвольный набор битовых строк. Основные операции алгоритма - селекция, скрещивание и мутация выполняются над элементами популяции. Результатом их выполнения является очередная популяция. Данный процесс продолжается итерационно до тех пор, пока не будет, достигнут критерий останова. Операция селекции обеспечивает формирование на очередной итерации алгоритма новой популяции из битовых строк, полученных на шагах 4, 5. Операция скрещивания обеспечивает обмен фрагментов различных битовых строк. Операция мутации обеспечивает случайное изменение отдельных битов строк популяции.

Рассмотрим основные решения, используемые в разработанных ГА для решения задач синтеза архитектур ВС.

Кодирование решения. Выбранный способ кодирования решений кратко может быть представлен следующим образом:

$B = B^{HP} \cup \{B^{HW,L}\}_L$,

где $\cup$- операция конкатенации (склейки) битовых строк, B^{HP} -поля кодирующие расписание выполнения программы, $\{B^{HW,L}\}_L$ - поля кодирующие варьируемые параметры архитектуры ВС. Варьируемые параметры, принадлежащие множествам действительных и натуральных чисел, кодируются в форме битового представления числа, множеству функций/операторов - в форме битового представления номера функции/оператора. Комбинаторные параметры задаются путем параметрического представления соответствующих комбинаторных структур, не допускающего получения недопустимых вариантов решения в результате выполнения операций генетического алгоритма. Комбинаторная структура восстанавливается по ее параметрическому описанию с использованием быстрых алгоритмов восстановления. Например, комбинаторный параметр расписание выполнения программы *HP* задается склейкой битовых полей процессов:

<поле процесса> $\equiv$ *(<номер процессора>* $\cup$ *<приоритеты интервалов>),*

<приоритеты интервалов> $\equiv \cup_i$ *(<приоритет интервала>*$_i$*).*

Поле «номер процессора» однозначно определяет распределение процессов по процессорам. Поле «приоритеты интервалов» используется для определения строгого порядка реализации рабочих интервалов на процессорах [8,9]. Одновременно с определением порядка осуществляется оценка динамических критериев и ограничений. Выбранный способ кодирования *HP* позволяет исключить появление недопустимых конфигураций из-за нарушения частичного порядка на множестве рабочих интервалов в результате выполнения операций скрещивания и мутации [8,9].

Операция мутации. Для выполнения операции мутации был предложен и реализован следующий алгоритм [8,9]: вместо того, чтобы вычислять мутацию для каждого бита, вычислялось расстояние от текущего бита. Бит, отстоящий на вычисленное расстояние, обязательно инвертировался.

Операция скрещивания: одноточечное скрещивание со случайным выбором битовых строк для скрещивания и точки скрещивания.

Способы применения операций скрещивания и мутации. Данные операции могут применяться: 1)для всей битовой строки; 2)к указанным полям (это дает возможность декомпозиции задачи на подзадачи в целях сочетания ГА с эвристическими методами или ускорения работы/акцентирования усилий ГА на определенных этапах работы по выделенным параметрам); 3)операция скрещивания манипулирует полями, не изменяя содержимое отдельных полей (это дает возможность на определенных этапах работы ГА комбинировать в различных вариантах полученные значения оптимизируемых параметров, например при выборе производительностей процессоров в разнородных ВС).

Селекция. Используются комбинация схемы пропорциональной селекции и схемы рулетки [9]. Для вычисления целого числа потомков используется схема пропорциональной селекции, а для распределения остатка – схема рулетки.

Целевая функция задается как функция/функционал от критериев оценки решений и ограничений на решения, например, как линейная функция от числа процессоров и времени выполнения программы в полученном решении [8,9].

Критерий останова. Останов ГА происходит, если значение критериев оценки качества решений попадает в некоторую заданную окрестность верхней/нижней границы критериев (данная граница может быть в принципе недостижимой) [9]. В противном случае, алгоритм останавливается, если за заданное число итераций, максимальное значение целевой функции улучшено не было.

Генетический алгоритм был апробироан для постановки задачи синтеза архитектур приведенной в разделе 3. В процессе экспериментов удалось

подобрать параметры генетического алгоритма (верятность скрещивания, вероятность мутации, весовые коэффициенты в целевой функции, размер популяции), при которых достигаются приемлемые сложность нахождения и качество решения [9] (никакой из эвристик не удалось найти лучшего решения, чем с помощью генетического алгоритма). Запуск алгоритма с другими значениями параметров значительно ухудшает качество получаемых результатов. Подходящее решение находилось за 50÷1500 итераций алгоритма. Зависимость числа итераций от качества получаемых решений и параметров алгоритма не прослеживается.

Литература

1. Смелянский Р.Л. Модель функционирования распределенных вычислительных систем// Вестн. МГУ, сер. 15, Вычислительная математика и Кибернетика, 1990., No. 3, С. 3-21.
2. Костенко В.А. Метод автоматизированного синтеза параллельных программ для вычислительных систем с MIMD архитектурой// Программирование, 1993., No. 1, стр. 43-57.
3. Костенко В.А. Автоматизация составления параллельных программ для вычислительных систем с MIMD архитектурой// Кибернетика и системный анализ, 1995., No. 5, стр. 170-179.
4. Подгорный С.А., Смелянский Р.Л. К вопросу синтеза структуры вычислительных систем на основе поведения программ// Вопросы организации программного обеспечения и принятия решений. МГУ, 1993.
5. Костенко В.А., Романов В.Г., Смелянский Р.Л. Алгоритмы минимизации аппаратных ресурсов ВС// Искусственный интеллект (Донецк). 2000, №2, С. 383-388.
6. У. Росс Эшби. Конструкция мозга. ИЛ, М.,1962.
7. Holland J.N. Adaptation in Natural and Artificial Systems. Ann Arbor, Michigan: Univ. Michigan Press, 1975.
8. Костенко В.А., Смелянский Р.Л., Трекин А.Г. Генетические алгоритмы: синтез структур вычислительных систем// Тезисы докладов Всероссийской научной конференции "Фундаментальные и прикладные аспекты разработки больших распределенных программных комплексов". М.: Изд-во МГУ, 1998. - С. 35-41.
9. Костенко В.А., Смелянский Р.Л., Трекин А.Г. Синтез структур вычислительных систем реального времени с использованием генетических алгоритмов// Программирование, 2000, № 5, С. 63-72.

Чистолинов М.В. Бахмуров А.Г.

Среда моделирования многопроцессорных вычислительных систем[1]

Введение

Термин *моделирование* чрезвычайно перегружен. Появился и в настоящее время стандартизуется унифицированный язык моделирования UML [1], с которым, обычно, и связывают представление о моделях информационных систем. Можно сказать, что UML представляет из себя набор графических нотаций, позволяющих описывать структуру, функциональность и поведение сложной системы. В данной работе мы будем говорить об *имитационном моделировании*. Этот способ исследования динамических систем позволяет не только описать поведение распределенного объекта, но и конструктивно построить его как набор историй функционирования [2], что существенно при использовании моделирования для анализа производительности.

Дальнейший материал организован следующим образом: раздел 1 посвящен особенностям организации и функционирования распределенных многопроцессорных вычислительных систем (РМВС), в нем также объясняется необходимость разработки новых методов анализа и инструментальных средств; в разделе 2 определяется круг задач, стоящих перед средой моделирования РМВС; в разделе 3 предлагается некоторая обобщенная архитектура среды автоматизированного имитационного моделирования и требования к ней; в разделе 4 описывается среда анализа и проектирования РМВС DYANA, в развитии которой авторы принимают непосредственное участие; в разделах 5 и 6 приводится опыт применения среды DYANA в области встроенных РМВС и излагаются перспективы развития данного проекта.

1. Особенности РМВС

Приведем некоторые особенности организации, функционирования и разработки распределенных вычислительных систем. Более подробный анализ можно найти в [2], [3].

- существенная зависимость производительности как от прикладной нагрузки, так и от алгоритмов арбитража разделяемых ресурсов (в том числе и аппаратных); показатели производительности приобретают статистический характер за счет наличия недетерминизма в поведении распределенных программ;
- сложность разработки и реализации параллельных алгоритмов; эта особенность обусловлена наличием разделяемых ресурсов и необходимостью синхронизации;
- сложность организации отладки параллельных программ; обусловлена отсутствием централизованного управления в системе и проблемой мгновенного останова вычислений;
- сложность интерпретации поведения РМВС: существование двух видов параллелизма - чередования и истинной паралельности, независимость поведения программы от времени, отсутствие единого времени;

[1] Работа частично поддержена грантами РФФИ № 98-01-00151 и EC INCO-COPERNICUS № 977020

Перечисленные особенности организации создают дополнительные сложности анализа и проектирования РМВС. Для этого, вообще говоря, требуется создание новых подходов и инструментальных средств. Имитационное моделирование хорошо зарекомендовало себя как способ исследования динамики функционирования систем практически неограниченной сложности. Результаты имитационного моделирования позволяют осуществлять анализ производительности, надежности и даже проверять некоторые логические свойства [4], [5].

2. Задачи среды моделирования РМВС

Нас будет интересовать среда имитационного моделирования, позволяющая описывать РМВС с учетом упомянутых выше особеностей, и решающая следующие задачи:

1. Описание встроенных систем широкого класса на системном уровне без использования прототипа аппаратуры.

2. Прогнозирование и анализ производительности систем при различном распределении функций между аппаратурой и программным обеспечением на стадии проектирования.

3. Разработка и отладка распределенных программ через моделирование и пошаговую детализацию.

4. Проверка и обоснование корректности реализации параллельных алгоритмов на основе одного и того же описания системы.

5. Оптимизация структуры вычислительной системы по заданному критерию. При этом необходимо предоставить возможность независимого эксперимента с моделями программного обеспечения и аппаратуры.

3. Архитектура среды автоматизированного имитационного моделирования

В любой среде имитационного моделирования можно выделить три группы инструментов, используемых для

- описания модели
- прогона модели (имитации)
- анализа и интерпретации результатов экспериментов

Остановимся подробно на первой группе. Основное требование - «естественность» описания модели, т.е. описание должно производиться в терминах исследуемой проблемной области. В случае моделирования РМВС, в языке моделирования или графической нотации должны присутствовать понятия процессов, вычислителей, примитивов синхронизации, коммуникационных каналов и т.д.

Другое важное свойство языка моделирования - это наличие формальной семантики. Только в этом случае можно обеспечить строгое, в математическом смысле, обоснование свойств модели.

К началу 80-х годов существовало более 500 языков имитационного моделирования [6] и их число постоянно росло. Тем не менее, можно указать общие проблемы их использования:

- Сложность или невозможность отладки имитационных моделей в терминах предметной области (ПО)
- Узкая направленность языка на конкретную ПО и/или схему моделирования

• Низкая эффективность моделирующих программ (в случае отказа от специализации)

• Необходимость навыков, выходящих за предметную область исследователя, например, навыков программирования

• Большой объем программирования и частный характер получаемых результатов

• Сугубо индивидуальный характер транслятора, необходимость его сопровождения

Можно предположить, что большая часть этих недостатков будет устранена за счет использования графических нотаций и объектно-ориентированного подхода. Другое перспективное направление - поддержка компонентной сборки имитационных моделей. Здесь можно говорить о независимой повторной используемости интерфейса, реализации и архитектуры модели или ее компонента.

4. Среда DYANA

DYANA (DYnamic ANAliser) - среда для анализа и проектирования PMBC [7] - построена в соответствии с требованиями и задачами, упомянутыми выше. Поддерживаются как языковые, так и графические средства описания модели. Описание модели PMBC состоит из:

- *описания распределенного исполнителя* (модель аппаратуры)
- *описания распределенной программы* (модель логической среды и прикладной программы)
- *описания привязки* (распределения процессов по исполнителям)

Распределенная программа состоит из последовательных процессов и других распределенных программ. Последовательные процессы взаимодействуют друг с другом посредством передачи сообщений. Распределенный исполнитель состоит из последовательных исполнителей и распределенных исполнителей. Последовательный исполнитель включает описание архитектуры, содержащей такие характеристики как множество команд, стековую и регистровую структуру процессора. Привязка определяет распределение последовательных процессов по последовательным исполнителям, а также наложение межпрограммных взаимдействий на каналы исполнителей.

Схема моделирования может быть определена следующим образом:

• Модель - набор параллельных процессов, взаимодействующих посредством передачи сообщений.

• Каждый процесс имеет конечное множество неограниченных буферов для приема и хранения сообщений.

• Попытка чтения из пустого буфера задерживает процесс до появления в этом буфере очередного сообщения.

Модель приводится к следующим алгоритмическим классам:

• Класс машин Тьюринга (TM);

• Класс сетей Петри (PN), если буфера неограничены и неупорядочены;

• Класс P/V систем (PV), если буфера неограничены, неупорядочены и в модели используется только один тип сообщения;

- Класс конечных автоматов (FS), если в модели все буфера ограничены.

Описание модели может автоматически преобразовываться в форму, пригодную как для количественного, так и для алгоритмического анализа. Алгоритмический анализ проводится подсистемой верификации на основе CTL-семантики языка моделирования.

Существенными возможостями среды DYANA являются отладка имитационной модели в терминах предметной области и оценка времени выполнения кода на языке высокого уровня для целевой архитектуры. Подсистема оценки времени позволяет по описанию стеково-регистровой архитектуры процессора получить время работы кода на нем без проведения эмуляции на уровне команд ассемблера.

Архитектура среды DYANA

5. Опыт моделирования встроенных систем

За десять лет существования, среда DYANA использовалась как инструментальное средство в нескольких десятках учебных и примерно в 10 крупных промышленных проектах. Основная область применения - анализ производительности бортовых вычислительных систем и сетевых протоколов. Ниже приводятся данные наиболее крупных работ:

НИИСИ РАН, 1997-1998 гг.

Построение модели распределенной вычислительной системы цифровой обработки сигналов БАГЕТ на базе процессоров DSP96002. Моделировалось более 120 процессоров. Задача оценки времени выполнения программы решалась штатными средствами среды DYANA примерно в 1000 раз быстрее, чем эмулятором фирмы Motorolla.

НИЦ «Контур» ФАПСИ, 1995-1999 гг.

Создание среды оценки производительности конфигураций телекоммуникационных сетей. В рамках проекта построены имитационные модели FDDI, IEEE 802.3, X.25, FrameRelay и стека TCP/IP.

INCO-COPERNICUS 977020 - DR TESY, 1998 - 2000 гг.

Цель этого международного проекта [5] - сравнительное испытание методов и инструментальных средств моделирования и логического анализа программного и аппаратного обеспечения встроенных систем реального времени на практическом примере многопроцессорной бортовой системы навигации. При моделировании использовались реальные алгоритмы. При этом замедление имитационной модели по сравнению с реальной системой составляет от 2 до 60 раз в зависимости от параметров комплекса. В ходе моделирования удалось автоматически проверить большую часть логических и временных свойств, сформулированных экспертами.

6. Перспективы развития среды DYANA

В качестве заключения, приведем направления развития среды DYANA в ближайшем будущем. Как нам кажется, они достаточно хорошо согласуются с общими тенденциями в области имитационного моделирования и проектирования встроенных систем.

- Организация компонентной сборки моделей из подмоделей
- Реализация объектно ориентированного подхода в средствах описания моделей
- Настройка средств отображения и анализа результатов моделирования на семантику конкретной модели
- Унификация языков спецификации логических и временных свойств
- Сопряжение DYANA с High Level Architecture (HLA) - стандартом для распределенного, полунатурного и кроссплатформенного моделирования [8]
- Обеспечение семантической интероперабельности моделей PMBC
- Сопряжение с UML либо реализация поддержки альтернативного жизненного цикла для имитационного моделирования (включая этапы калибровки и обоснования адекватности модели)
- Реализация в DYANA распределенного имитационного моделирования
- Поддержка подсистемой оценки времени библиотеки современных CPU, в том числе заданных своими VHDL-описаниями
- Борьба со сложностью системы переходов формальной модели, построенной по описанию на языке моделирования

Литература

1. Фаулер, Скотт UML в кратком изложении// М. Мир 1999
2. Смелянский Р.Л. Анализ функционирования многопроцессорных распределенных вычислительных систем на основе инварианта поведения программ. Диссертация на соискание степени д.ф.-м.н., МГУ 1990
3. Смелянский Р.Л. Об инварианте поведения программ// Вестн. МГУ, сер. 15, Вычислительная математика и Кибернетика, 1990., No. 4, С. 54-60.
4. Anders Ek Automatic Debugging of Communicating Systems Using the SDT Validator Telelogic AB
5. Домашняя страница проекта DrTesy, отчет по второму этапу http://www.first.gmd.de/~drtesy/
6. Киндлер Е. Языки моделирования// М. Энергоатомиздат 1987

7. DYANA: An Environment for Embedded System Design and Analysis Bahmurov A.G., Kapitonova A.P., Smeliansy R.L. (TACAS 99)
8. Учебный курс McLeod Institute по HLA http://www.ecst.csuchico.edu/~hla

9. **Шчепанович Стеван [1], Леонтьев Дмитрий [2], Мратов Александр[2]**

Вопросы определения калибровочных параметров имитационных моделей серверных приложений

[1] - ф-т естественно-математический, Университет Черногории, Подгорица, Югославия,
[2] -ф-т ВМиК, МГУ, Москва,

1.Введение

Массовое использование и стремительный рост вычислительных сетей в современной жизни привело к необходимости оптимизации существующих сетей и определения производительности проектируемых. Одним из эффективных и дешевых способов исследования системы и проведения экспериментов с ней является моделирование. В силу сложности вычислительной сети и недетерменированности поведения использование аналитических методов исследования является затруднительным. На помощь приходит имитационное моделирование, которое с высокой степенью точности [1] позволяет исследовать моделируемую систему. Одним из этапов построения модели, влияющих на точность, является калибровка построенных моделей. В данной работе рассмотрены общие принципы калибровки имитационных моделей сетевых компонентов и предложена методика калибровки моделей файл-сервера и сервера базы данных.

2.Основные понятия

Опишем базовые понятия, необходимые для рассмотрения процесса моделирования.

Система - группа или совокупность объектов, объединенных некоторой формой взаимодействия или взаимосвязи с целью выполнения определенной функции [2]. Функционирование системы представляет собой совокупность координированных действий, необходимых для выполнения определенной задачи.

Моделью в общем случае является представление объекта или системы в некоторой форме, отличной от формы их реального существования [3].

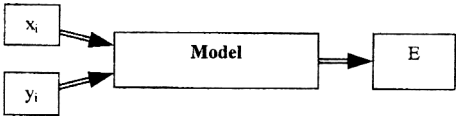
Имитационное моделирование - есть процесс конструирования модели реальной системы и постановки экспериментов на этой модели с целью исследования поведения системы, либо оценки (в рамках ограничений, накладываемых некоторым критерием или их совокупностью) различных стратегий, обеспечивающих функционирование данной системы [2].

Внутренняя структура имитационной модели, согласно [3], представляет собой некоторую комбинацию следующих составляющих:

- компоненты;
- переменные;
- параметры;
- функциональные зависимости;

- целевые функции.

Перед построением имитационной модели необходимо определить, что представляют собой ее структурные элементы. Хотя математическая или физическая структура модели может быть сложной, основная схема ее построения весьма проста. Модель можно представить в виде "черного ящика", на входе которого мы имеем некоторый набор переменных и параметров, а на выходе - результат действия системы. Связь между ними можно выразить следующим образом:



где x_i - переменные системы;

y_i - параметры системы;

E - результат действия системы (целевая функция), причем может выполняться соотношение $E=f(x_i,y_i)$, где f - функциональная зависимость между x_i,и y_i, определяющая величину E.

3.Имитационные модели сетевых компонент

Имитационная модель представляет собой программу, написанную на одном из языков высокого уровня (C/C++), в которой вводится дискретно-событийная модель функционирования и единое модельное время для всех компонент модели. В данном случае для построения моделей использовалась система имитационного моделирования DYANA [4].

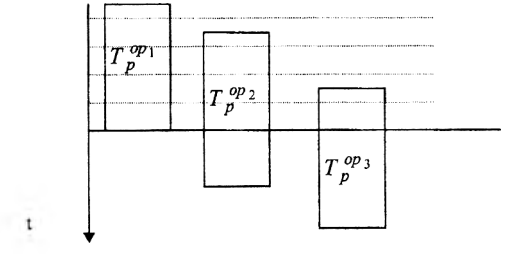
Моделируемой системой в данном случае является вычислительная сеть, которая состоит из каналов передачи данных, промежуточного (маршрутизаторы, шлюзы, мосты и т.д.) сетевого оборудования, оконечного (сервера приложений, машины конечных пользователей) сетевого оборудования [5]. В свою очередь модели объектов содержат в качестве компоненты модели программного обеспечения (операционная система, стек протоколов, приложения). Все построенные модели образуют библиотеку настраиваемых моделей [6]. Настройка моделей осуществляется путем определения значений соответствующих параметров. Процесс калибровки заключается в выделении параметров, однозначно характеризующих компонент модели, и определения их значений. Часть значений калибровочных параметров можно получить из документации, но зачастую калибровочные параметры связаны со временем выполнения определенных операций (задержек) по обработке информации на конкретном оборудовании. Определить значения калибровочных параметров в данном случае можно с помощью эксперимента с использованием программных наблюдателей [1]. Далее показана методика определения временных задержек при выполнении запросов к файл-серверу и серверу базы данных и описаны проведенные эксперименты по этой методике для ОС Novell NetWare 4.1.

4. Методика измерения производительности серверов

Начальным этапом измерения производительности любой вычислительной системы является выбор меры или мер производительности, т.е. тех параметров, которые ее характеризуют. В качестве меры производительности серверов в данной работе было выбрано время выполнения операций (запросов к серверам).

Такой выбор соответствует концепции построения модели сервера. Упрощенно модель функционирования выглядит так: на вход серверу поступает запрос, сервер делает задержку на его обработку и посылает ответ.

Рассмотрим более подробно механизм обработки сервером нескольких запросов. Пусть T_p^{op} - удельное время выполнения сервером операции *op*, т.е. время необходимое серверу для выполнения данной операции при отсутствии любых других запросов. Если же серверу надо выполнить одновременно несколько операций, то он выполняет их в режиме разделения времени.



Перейдем к рассмотрению собственно методики измерения.

Основная проблема состоит в том, как ввести измерительный (наблюдательный) инструмент в систему, чтобы он не вносил существенных возмущений в ее работу. Наблюдатель должен следить за системой и с помощью имеющихся у него измерительных инструментов оценивать количественно происходящие изменения. Однако, наблюдение - это взаимодействие с системой, т.е. наблюдатель должен быть хотя бы пассивным посредником при передаче определенной информации в системе.

В данном исследовании все программные наблюдатели размещались на рабочих станциях - клиентах и сочетали в себе функции задания рабочей нагрузки системы и проведения измерений. Таким образом, они практически не оказывали влияния на работу серверов.

Точность измерений зависит от "чистоты" экспериментов и от разрешающей способности наблюдателя.

Под "чистотой" экспериментов в случае измерения производительности серверов понимается зависимость результатов от:

- загруженности КЭШ жесткого диска;

- размещения файлов на диске (разброса секторов одного файла);
- особенностей работы операционной системы.

Разрешающая способность определяется максимальным интервалом между "тиками" часов наблюдателя.

В качестве часов измерительной системы был использован системный таймер персонального компьютера. Он "тикает" примерно 18,2 раза в секунду, или каждых 0,054945 секунд.

Для измерения производительности серверов средствами языков Си и Ассемблер были разработаны два вида программных наблюдателей:

1. для измерения общего времени прохождения запроса по сети, обработки его сервером и прохождения ответа по сети.
2. для измерения времени обработки запроса на сервере.

Наблюдатели первого вида работают на рабочей станции - клиенте. Они измеряют рабочую нагрузку (поток запросов) и проводят замеры времени от ухода запроса в сеть до прихода ответа от сервера (т.е. время обработки запроса в сети). Измерения для каждой операции проводились отдельно. Каждый поток состоял из 200 одинаковых запросов для файл-сервера и 10000 одинаковых запросов для сервера базы данных. Число запросов в каждом потоке выбиралось так, чтобы значение ошибки от таймера не превосходило 10%. Для уменьшения ошибки из-за условий проведения экспериментов каждый эксперимент состоял из 20 - 50 испытаний (повторов). Число повторов выбиралось так, чтобы значение дисперсии не превосходило 10%.

Наблюдатели второго вида работают на сервере. Они измеряют рабочую нагрузку (поток запросов) напрямую на сервер и производят замеры времени от посылки запроса на сервер до прихода ответа с сервера.

Поток запросов, создаваемый в качестве рабочей нагрузки, для наблюдателей обоих видов был одинаков и равномерно распределен.

При проведении экспериментов число, длина принятых и переданных пакетов определялась с помощью программы Monitor службы диагностики файл-сервера Novell NetWare.

Интерфейс между наблюдателями и сетью осуществлялся через драйвер IPXODI.COM.

Ошибка измерений вычислялась как дисперсия:

$$S_n^2 = \left[1/n - 1\right] \sum_{i=1}^{n} \left(x_i - \bar{x}_n\right)^2 , \text{где,}$$

n - количество запросов в потоке,

$\bar{x}_n = \left(1/n\right) \sum_{i=1}^{n} x_i$ - математическое ожидание времени выполнения одного запроса.

Ошибка от таймера вычислялась по следующей формуле:

$S_t = (T_r / T_{op}) * 100\%$,

Где,

T_r - разрешающая способность таймера (55 мс),

T_{op} - время выполнения операции.

5.Описание экспериментов

Было проведено измерение производительности файл-серверов и серверов баз данных (с СУБД Btrieve 6.10), работающих под управлением операционных систем Novell NetWare 4.1. Для проведения экспериментов была собрана сеть из одного сервера и одной рабочей станции - клиента следующих конфигураций:

Сервер:

- центральный процессор i486 DX2 с частотой 66 МГц,
- оперативная память объемом 12 Мбайт с временем доступа 80 нс,
- накопитель на жестких магнитных дисках WESTERN DIGITAL Caviar 2120 емкости 125 Мбайт,
- 16-ти разрядная сетевая плата типа Novell Ethernet NE2000.

Клиент:

- центральный процессор i386 с частотой 40МГц ,
- оперативная память объемом 4Мбайт,
- накопитель на жестких магнитных дисках емкости 40 Мбайт,
- 16-ти разрядная сетевая плата типа Novell Ethernet NE2000.

На сервере помещалась сетевая операционная система Novell NetWare 4.1, на рабочую станцию, работающую под управлением операционной системы MS-DOS 6.22, - программное обеспечение клиента.

Проведено исследование следующих типов запросов:

к файл-серверу:

- создание (*create*) файла,
- открытие (*open*) файла,
- закрытие (*close*) файла,
- удаление (*delete*) файла,
- чтение (*read*) из файла,
- запись (*write*) в файл.

к серверу базы данных:

- вставка (*insert*) записи в базу данных,
- выбор (*select*) записи из базы данных,
- обновление (*update*) записи в базе данных,
- удаление (*delete*) записи из базы данных.

Для исследования запросов к серверу базы данных была создана экспериментальная база данных со следующими характеристиками:

- длина записи - 100 байт,
- длина блока - 512 байт.
- один ключ длиной 4 байта.

Обращение к базе данных всегда осуществлялось последовательно.

Результаты экспериментов перечислены в нижеследующих таблицах.

Таблица 1. Время обработки одной операции на файл-сервере Novell NetWare 4.1.

Novell NetWare 4.1	среднее время обработки запроса сервером [мс]	среднее время обработки запроса в сети [мс]	количество испытаний	среднее число прин./пер. пакетов на файл-сервере	средняя длина прин./пер. пакетов на файл-сервере	ошибка измерений обусловленная дисперсией [%]	ошибка измерений обусловленная таймером [%]
create	8,02	8,97	50	1/1	71,7/91,6	3,68	3,06
open	5,82	6,91	24	1/1	72,8/90,1	3,72	3,97
close	4,86	5,96	50	1/1	59,3/54,0	9,29	4,61
delete	1,66	2,76	50	1/1	71,1/54,0	8,94	9,97

Таблица 2. Скорость чтения и записи в файл, размещенный на файл-сервере Novell NetWare 4.1.

Novell NetWare 4.1	средняя скорость обработки запроса сервером [Кбайт/с]	средняя скорость обработки запроса в сети [Кбайт/с]	количество испытаний	ср. число прин./пер пакетов для файла 1000 Кбайт	средняя длина прин./пер. пакетов файл-сервером	ошибка измерений обусловленная дисперсией [%]	ошибка измерений обусловленная таймером [%]
write	184,74	150,03	28	751/750	1491,3/52,4	0,64	0,82
read	376,05	255,65	28	745/747	64,1/1464,5	0,81	1,41

Таблица 3. Время обработки операции над одной записью базы данных.(СУБД - Btrieve 6.10, ОС - Novell NetWare 4.1.)

Novell NetWare 4.1	среднее время обработки запроса сервером [мс]	среднее время обработки запроса в сети [мс]	количество испытаний	среднее число прин./пер. пакетов на сервере БД	средняя длина прин./пер. пакетов на сервере БД	ошибка измерений обусловленная дисперсией [%]	ошибка измерений обусловленная таймером [%]
insert	3,751	4,856	21	2/2	136/95	2,65	0,11
select	2,073	3,173	23	2/2	88/145	0,66	0,17
update	5,070	7,230	22	4/4	113/120	1,17	0,07
delete	5,898	8,108	21	4/4	87/119	1,61	0,06

6. Заключение

В данной работе была предложена методика измерения, позволяющая легко и быстро измерять производительность серверов. Разработана система программных наблюдателей. Приведен пример их применения для измерения производительности файл-серверов и серверов баз данных (с СУБД Btrieve 6.10), работающих под управлением операционной системы Novell NetWare 4.1.

Полученные значения довольно ценны - они были использованы в качестве калибровочных параметров при построении имитационных моделей данных приложений. Кроме того, на основании результатов работы можно сделать выводы о производительности приложений, работающих под управлением операционной системы Novell NetWare 4.1.

В дальнейшем предполагается определить место калибровки моделей в процессе определения адекватности имитационных моделей и расширить методику калибровки моделей на более широкий класс имитационных моделей сетевых устройств и приложений.

Литература

1.Смелянский Р.Л. "Методы анализа и оценки производительности последовательных вычислительных систем". // Методическая разработка. М.: Ротапринт НИВЦ МГУ, 1990

2.Y.Monsef "Modelling and Simulation of Complex Systems. Concepts, Methods and Tools", Budapest University of Economic Science, 1997

3.Шеннон Р.Ю. "Имитационное моделирование систем - искусство и наука", М.: Мир, 1978

4.Бахмуров А.Г., Костенко В.А., Смелянский Р.Л. "Среда моделирования DYANA: синтез анализ и оптимизация вычислительных систем" // Тезисы докладов Всероссийской научной конференции "Фундаментальные и прикладные аспекты разработки больших распределенных программных комплексов" (сентябрь 1998 г., г. Новороссийск) М.: Изд-во МГУ, 1998

5.Макаров-Землянский Н.В., Шчепанович С. "Имитационная модель многовходового адаптера межмашинной связи сети ЭВМ "Нерпа" // Сборник "Информационное моделирование и проектирование подсистем АСУ", М.: МГУ, 1988

6.Леонтьев Д.В. Средства анализа функционирования клиент/серверных систем // Сборник докладов Научной конференции, посвященной 70-летию со дня рождения академика В. А. Мельникова, Москва, М.: Научный Фонд "Первая Исследовательская Лаборатория имени академика В. А. Мельникова", 1999

Вознесенская Т. В.

Математическая модель алгоритмов синхронизации времени для распределенного имитационного моделирования

Введение.

Под имитационной моделью будем понимать программу, которая явно воспроизводит поведение интересующего нас объекта с целью его исследования. Будем считать, что в такой программе поведение каждой самостоятельной сущности исследуемого объекта представляется отдельным процессом. Отличительной особенностью программ имитационного моделирования является то, что поведение всех процессов должно быть согласовано в едином модельном времени, то есть в том времени, в котором "живет" объект моделирования. Каждое событие в модели имеет вычисляемую в процессе работы (прогона) метку – модельное время его наступления. Событие – это изменение состояния системы.

Будем рассматривать имитационные модели, которые

1. состоят из множества процессов, взаимодействующих между собой через прием и посылку сообщений;
2. имеют частично упорядоченное в модельном времени множество событий: если модельное время события a меньше модельного времени события b, значит, событие a наступило раньше события b; одновременные события не упорядочены.

Последовательное имитационное моделирование для таких моделей происходит следующим образом. Вначале вычисляются метки готовых к обработке событий. Такие события заносятся в список (календарь) событий и упорядочиваются по неубыванию модельного времени. Затем модельное время устанавливается равным метке первого события, а само событие обрабатывается. В результате появляются новые готовые к обработке события. Они в правильном порядке заносятся в календарь. Причем события со временем, меньшим текущего появиться не могут. И так далее.

При распределении имитационной модели на несколько процессоров появляется распределенное модельное время и распределенный календарь событий. Для поддержания правильной логики развития модели в этом случае необходимо специальное средство синхронизации – алгоритм синхронизации (АС).

Существует много различных АС (в разделе 1 приведена их классификация). Нет однозначного ответа на вопрос, какой АС лучше. Все алгоритмы имеют свои плюсы и минусы, и их эффективность зависит от свойств имитационных моделей.

Целью данной работы является разработка математической модели развития пары (АС, имитационная модель) для исследования свойств, сравнения и оценки различных таких пар.

Имитационную модель будем называть в дальнейшем приложением.

Структура этой статьи такова. В разделе 1 на примере рассмотрена проблема синхронизации времени в распределенном имитационном моделировании. Раздел 2 содержит основные принципы построения математической модели функционирования пары (АС, приложение). В разделе 3 приводится математическая модель последовательного случая. В разделах 4, 5, 6 строятся математические модели функционирования приложения под управлением строго син-

хронного, консервативного и оптимистических АС соответственно. В разделе 7 дан метод сравнения различных АС на основе этих моделей.

1. Проблема единого модельного времени в распределенном имитационном моделировании.

Проиллюстрируем основные проблемы, возникающие при распределенном имитационном моделировании на примере.

Рассмотрим взаимоотношения клиента, банка и магазина. Клиент делает в магазине покупки, расплачиваясь карточкой, и ходит в банк за деньгами. Магазин сообщает банку о покупках клиента. На основании этих сообщений банк решает, можно клиенту выдавать деньги, или нужно взять с него сумму, чтобы покрыть покупки (если произошел перерасход средств).

Опишем эти взаимодействия в виде приложения, состоящего из трех процессов:

P1 – магазин;

P2 - банк;

P3 – клиент.

Совершение покупки клиентом в магазине моделируется посылкой сообщения процессом *P3* процессу *P1* (событие *a*). Уведомление магазином банка отражает передача сообщения от процесса *P1* к процессу *P2* (событие *b*). Визит клиента в банк описывает передача сообщения от процесса *P3* к процессу *P2* (событие *c*). Информирование банком клиента о возможной операции осуществляется посылкой сообщения процессом *P2* процессу *P3* (событие *d*). В зависимости от информации, содержащейся в этом сообщении, клиент снимает деньги (событие *e*), либо пополняет счет (событие *f*).

Поведение этой системы показано в виде диаграммы на рис.1 (а), где *T*-ось модельного времени. Модельное время события *x* будем обозначать *T(x)*. В приведенном фрагменте $T(a)<T(b)<T(c)<T(d)<T(e)<T(f)$.

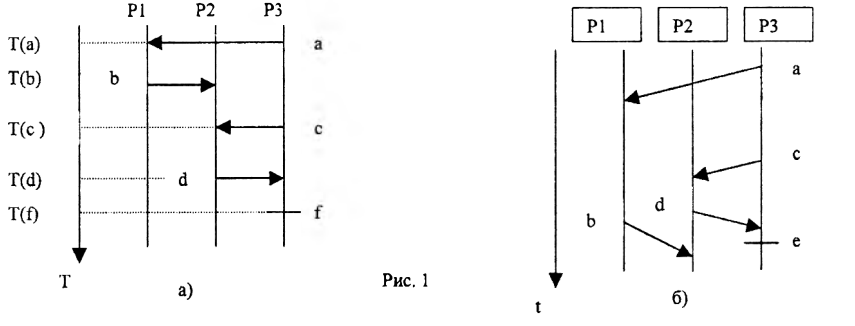


Рис. 1

Здесь предполагается, что:

1. сообщения передаются мгновенно в модельном времени, следовательно, ранее отправленное сообщение должно быть ранее принято;
2. события всех процессов частично упорядочены в едином модельном времени (частично, так как порядок одновременных событий, не связанных логикой программы, не важен).

При распределенном моделировании однородность времени, описанная свойством 2, может нарушаться. Разместим все процессы из нашего примера на отдельных процессорах. На рис. 1 (б) показано возможное развитие приложения

в физическом времени (ось *t*). На этой диаграмме процесс *P2* успевает получить сообщение от процесса *P3* и отправить ему ответ прежде, чем он получит сообщение от *P1*. Это может произойти по одной из следующих причин:

- процессор 1 медленнее остальных;
- линия связи между процессорами 1 и 2 работает медленнее, чем между процессорами 2 и 3;
- для продвижения на единицу модельного времени процессу *P1* надо выполнить больше операций, в этом случае даже при размещении всех процессов на одинаковых процессорах, процесс *P1* будет продвигать модельное время медленнее.

В результате, к моменту прихода клиента банк не знает о его покупке в магазине, которая привела к перерасходу, и разрешает ему снять деньги. Ошибка здесь в том, что вместо события *f* происходит событие *e*.

Это искажение поведения приложения произошло из-за пространственно-временной неоднородности среды моделирования, то есть из-за отсутствия единого модельного времени у разных процессов. Действительно, с точки зрения свойства 2 событие *d* при прогоне приложения не выполнимо до тех пор, пока не придет сообщение, посланное событием *b*. Однако, если модельное время процессов не согласовано, то у процесса *P2* нет причин не выполнять событие *d*.

Избежать подобных ситуаций призваны как раз алгоритмы синхронизации.

Условимся далее подразумевать модельное время, употребляя слово "время" без прилагательного. Физическое время будет всегда упоминаться явно.

Все АС можно разделить на четыре основных класса [1]: строго синхронные, с синхронизацией взаимодействий, консервативные и оптимистические.

Строго синхронные АС поддерживают единые часы для всего приложения, то есть любое событие продвижения модельного времени подлежит синхронизации.

Консервативные АС и АС с синхронизацией взаимодействий позволяют процессам развиваться независимо до тех пор, пока не встретится событие получения сообщения. Обработка этого события разрешается только в случае, если известно, что процесс не может получить "отстающее" сообщение. (Для АС с синхронизацией взаимодействий: если во всем приложении не могут появиться сообщения с меньшим модельным временем.) Иначе процесс блокируется и ждет, когда можно будет обработать событие получения сообщения.

Оптимистические АС (ОАС) позволяют приложению развиваться естественным образом, без каких-либо ограничений, но при возникновении некорректных ситуаций (получении "отстающего " сообщения) выполняют откат назад.

2. Основные принципы построения математической модели функционирования пары (АС, приложение).

N взаимодействующих процессов исполняются на *P* процессорах $P=\{1,N\}$. Вариант $1<P<N$ не рассматриваем, так как в таком случае скорость развития приложения будет зависеть еще и от алгоритма распределения процессов по процессорам. Выбор такого алгоритма – тема отдельного исследования. За системой ведется наблюдение через равные промежутки физического времени. Моменты наблюдения назовем шагами и занумеруем 1,2,...,. Каждый процесс имеет свои локальные часы, либо в системе единое модельное время, то есть значения всех локальных часов одинаковы. На *i*-м шаге состояние системы оп-

ределяется значением вектора $X_i = (x_i^1, x_i^2, \ldots, x_i^N)$ текущих времен процессов. В случае единого времени X_i - скаляр. Время всей системы на i-м шаге $T_i = \min\limits_{n=\overline{1,N}} x_i^n$. Значения X_i меняются на каждом шаге по определенному закону. Этот закон описывается случайным процессом, вид которого зависит от конкретного АС, то есть X_i – случайный процесс. На любом шаге процессы могут обмениваться сообщениями. k-й процесс на i-ом шаге посылает сообщение l-му процессу с вероятностью $\alpha_i^{kl}(k,l=\overline{1,N})$, $0 \le \alpha_i^{kl} \le 1$. Предполагаем, что сообщения передаются мгновенно. Затраты времени на внутреннюю работу процессов между двумя соседними шагами (приращения локальных часов) характеризуются случайными величинами $\{\xi_i^k\}$, $k=\overline{1,N}, \xi_i^k > 0$.

Таким образом, входными параметрами модели являются вероятности посылки сообщений между процессами $\{\alpha_i^{kl}\}$ и приращения локальных часов процессов $\{\xi_i^k\}$. Эти параметры характеризуют приложение. АС характеризует закон продвижения модельного времени.

Выбирая значение P и закон продвижения модельного времени, можно построить модели функционирования как последовательных приложений, так и распределенных с синхронными, консервативными или оптимистическими АС.

Целью построения таких моделей является анализ и сравнение различных АС. Критерий эффективности при сравнении: лучше тот АС, который способен за данное число шагов (аналог физического времени) развить приложение до большего значения времени. Поскольку оптимистические АС допускают откаты, а другие АС – блокировку процессов, то одно и то же приложение под управлением разных АС за одинаковое число шагов достигает, вообще говоря, различных значений времени.

3. Последовательное имитационное моделирование

$P=1$, то есть все процессы исполняются на одном процессоре.

X_i скаляр, то есть в системе единое время. Для каждого процесса определим величину l_j^k, которая показывает до какого значения k –й процесс хотел бы продвинуть часы на j-м шаге. Тогда продвижение времени в системе можно описать следующим образом:

$$\begin{cases} l_{j+1}^n = \begin{cases} l_j^n + \xi_j^n, \text{если } l_j^n = X_j \\ l_j^n, \text{иначе} \end{cases} \\ l_0^n = 0 \\ n = \overline{1,N} \\ X_{j+1} = \min\limits_{n=\overline{1,N}}(l_{j+1}^n) \end{cases}$$

На каждом шаге процессы, у которых $l_j^n > X_j$ остаются заблокированными. Только те процессы, у которых $l_j^n = X_j$ разблокируются (то есть могут обработать события, назначенные на это время) и пересчитывают l_{j+1}^n.

Для наглядности рассмотрим пример, в котором процессы двигаются с постоянными шагами $\{t^1 = 2, t^2 = 3, t^3 = 6, t^4 = 10\}$:

j	X_j	l_j^1	l_j^2	l_j^3	l_j^4
0	0	0	0	0	0
1	2	2	3	6	10
2	3	4	3	6	10
3	4	4	6	6	10
4	6	6	6	6	10
5	8	8	9	12	10
6	9	10	9	12	10
7	10	10	12	12	10
8	12	12	12	12	20
9	14	14	15	18	20
10	15	16	15	18	20
11	16	16	18	18	20
12	18	18	18	18	20
13	20	20	21	24	20

Поскольку все процессы исполняются на одном процессоре, то они реально работают в режиме чередования и выполняют свои вычисления (внутреннюю работу), соответствующие продвижению времени, по очереди. Поэтому, чтобы продвинуть время на некоторую величину, необходимо для каждого процесса по очереди (всего N раз) выполнить соответствующие действия по продвижению времени. То есть увеличение времени происходит каждые N шагов. Поэтому в формуле (3.1) $j=kN$, $k=0,1,2,\ldots$. Тогда

$$\begin{cases} l^n_{(k+1)N} = \begin{cases} l^n_{kN} + \xi^n_{kN}, если\, l^n_{kN} = X_{kN} \\ l^n_{kN}, иначе \end{cases} \\ l^n_0 = 0 \\ n = \overline{1,N} \\ X_{(k+1)N} = \min\limits_{n=1,N}(l^n_{(k+1)N}) \end{cases} \qquad (3.1)$$

4. Строго синхронные АС

$P=N$. X_i скаляр. От последовательной версии такое моделирование отличается только тем, что ведется на N процессорах, а не на одном, поэтому подвигать время могут все процессы одновременно. Закон продвижения времени в этом случае описывается следующей системой:

$$\begin{cases} l^n_{i+1} = \begin{cases} l^n_i + \xi^n_i, если\, l^n_i = X_i \\ l^n_i, иначе \end{cases} \\ l^n_0 = 0 \\ n = \overline{1,N} \\ X_{i+1} = \min\limits_{n=1,N}(l^n_{i+1}) \end{cases} \qquad (4.1)$$

Время всей системы обязательно будет двигаться вперед. Поскольку модель последовательного имитационного моделирования отличается от модели строго синхронного АС только коэффициентом N, то все, доказанное ниже, справедливо и для последовательного случая.

Обратимся к уравнениям (4.1). $X_i = \min\limits_{n=1,N} l_i^n$. На $i+1$-м шаге увеличивается значение той величины l_i^n, которая равна X_i, то есть $\min\limits_{n=1,N} l_i^n$ с увеличением шага растет: $l_{i+1}^n = l_i^n + \xi_i^n$, *если* $l_i^n = X_i = \min\limits_{n=1,N} l_i^n \Rightarrow X_{i+1} > X_i$. Так как в данном случае $T_i = X_i$, то отсюда следует $T_{i+1} > T_i$.

Формула (4.1) показывает, что быстродействие пары (строго синхронный АС, приложение) определяется самым медленным процессом. За один шаг продвижение часов возможно на величину $\le \min\limits_{n=1,N}(\xi_i^n)$. Это означает, что данный АС не выгодно использовать при большом разбросе значений приращений локальных часов процессов.

5. Консервативные АС и АС с синхронизацией взаимодействий.

$P=N$. X_i - вектор $(x_i^1, x_i^2, \ldots, x_i^N)$, где x_i^k - значение локальных часов k-го процесса. Для любого n-го процесса определим множество $M^n \subset \{1,2,\ldots,N\}$ процессов, которые могут послать ему сообщение.

Для любого процесса n и любого шага $i+1$ продвижение локальных часов происходит только в том случае, если

- (для консервативных АС) невозможно получение сообщения от процесса с меньшим временем, то есть $\forall m \in M^n : x_i^n + \xi_i^n \le x_i^m + \xi_i^m$. Однако в системе могут появиться сообщения с меньшим временем, лишь бы их отправители не принадлежали множеству M^n;
- (для АС с синхронизацией взаимодействий) в системе уже не могут появиться сообщения, отправитель которых имест меньшее значение локальных часов (т. е. $M^n = N$).

$$\begin{cases} x_{i+1}^n = x_i^n + \xi_i^n F^n(i) \\ F^n(i) = \begin{cases} 1, & \text{если } \forall m \in M^n : x_i^n + \xi_i^n \le x_i^m + \xi_i^m \\ 0, & \text{иначе} \end{cases} \\ x_0^n = 0 \\ n = \overline{1,N} \end{cases} \qquad (5.1)$$

Уравнения (5.1) соответствуют консервативному АС Chandy-Misra с передачей нулевых сообщений [3]. Для случая $M^n = N$ они соответствуют АС с синхронизацией взаимодействий.

Рассмотрим процесс развития системы (5.1). На первом шаге локальные часы любого процесса n принимают значение либо $x_1^n = \xi_1^n$, либо $x_1^n = x_0^n = 0$. На втором шаге любой процесс может либо продвинуть свои локальные часы на соответствующее ξ_2^n, либо оставить их значение прежним, и так далее.

Время всей системы обязательно будет двигаться вперед. Из формул (5.1) очевидно, что $\forall n : x_{i+1}^n \ge x_i^n \Rightarrow T_{i+1} \ge T_i$. На каждом шаге хотя бы для одного n выполнено условие $\forall m \in M^n : x_i^n + \xi_i^n \le x_i^m + \xi_i^m$ просто потому, что в последовательности чисел всегда существует минимальное. Отсюда следует, что на ка-

ждом шаге хотя бы одному процессу разрешено продвижение часов и, следовательно, хотя бы для одного n выполнено условие $x_{i+1}^n > x_i^n$.

Рассмотрим i-й шаг. Обозначим $x_i^{\min} = \min_{n=\overline{1,N}} x_i^n$. Если на $i+1$ шаге разрешено продвижение для процесса с номером *min*, то очевидно, что $T_{i+1} > T_i$. В противном случае, пусть $L < \infty$ сколь угодно большое число такое, что заведомо $L > (x_i^{\min} + \xi_i^{\min})$. Зная на i-м шаге значение $x_i^{\min}$ и закон распределения $\xi_i^{\min}$, можно выбрать такое L с вероятностью близкой к единице. Тогда, учитывая, что число процессов N конечно и на каждом шаге хотя бы один процесс увеличивает значения своих часов, можно сделать вывод, что за конечное число шагов k сложится ситуация, при которой $\forall n \neq \min : x_k^n \geq L$. (Для вещественных ξ_i^n можно их точность ограничить каким-то конечным числом знаков после запятой.) Такая ситуация означает, что будет разрешено продвижение локальных часов процессу с номером *min*, то есть $x_k^{\min} > x_i^{\min} \Rightarrow T_k > T_i$. Отсюда следует, что выполнено условие: $\forall i \exists k : k > i,\ k - i < \infty : T_k > T_i$.

Из формул (5.1) видно, что $\forall i$ максимальное значение T_i достигается в случае пустых множеств $\{M^n\}$ или, когда процессы могут получать сообщения только от процессов с большим временем. В этом наилучшем случае, развитие системы описывается следующими уравнениями:

$$\begin{cases} x_{i+1}^n = x_i^n + \xi_i^n \\ x_0^n = 0 \\ n = \overline{1,N} \end{cases} \qquad (5.1')$$

Количество элементов в множестве M^n - это количество процессов, от которых зависит возможность продвижения времени n-го процесса. В общем случае, чем меньше элементов содержат множества $\{M^n\}$, тем лучше. Для АС с синхронизацией взаимодействий $\{M^n\}$ содержат наибольшее число элементов, поэтому этот АС всегда будет давать результаты, не лучшие, чем консервативный.

6. Оптимистические алгоритмы синхронизации

Данная математическая модель является обобщением по числу процессов математической модели взаимодействия двух процессов под управлением ОАС, разработанной D.Mitra и I.Mitrani [2].

6.1. Классический ОАС

P=N. X_i - вектор$(x_i^1, x_i^2, \ldots, x_i^N)$, где x_i^k - значение локальных часов k-го процесса.

Рассмотрим закон продвижения времени. Если на i-м шаге k-й процесс получил отстающее сообщение от l-го $(x_i^l < x_i^k)$, то он делает откат до времени $x_i^l (x_{i-1}^k = x_i^l)$. Поскольку он может получить, вообще говоря, несколько отстающих сообщений, то он делает откат до минимального из них.

Вне зависимости от того, произошел откат, или нет, k-й процесс продвигает свои часы на случайную величину ξ^k.

Обозначим $k \to l$ событие "k-й процесс послал сообщение l-му". Обозначим I_i^B - индикатор события B

$$I_i^B = \begin{cases} 1, & \textit{событие } B \textit{ произошло на } i-\textit{м шаге} \\ 0, & \textit{событие } B \textit{ не произошло на } i-\textit{м шаге} \end{cases}$$

Тогда $I_i^{m \to n}$ - индикатор события посылки сообщения m-м процессом n-у на i-м шаге, а $I_i^{x_i^m < x_i^n}$ - индикатор события, заключающегося в том, что на i-м шаге значение локальных часов m-го процесса меньше значения локальных часов n-го.

Развитие модели может быть описано следующей системой уравнений:

$$\begin{cases} x_{i+1}^n = x_i^n + \xi_i^n - \max\limits_{m=\overline{1,N}}((x_i^n - x_i^m) I_i^{m \to n} I_i^{x_i^m < x_i^n}) \\ x_0^n = 0 \\ n = \overline{1,N} \end{cases} \qquad (6.1.1)$$

Действительно, на каждом шаге для любого процесса n к предыдущему значению его локальных часов добавляется случайная величина ξ^n, отражающая затраты на внутреннюю работу процесса на шаге. Кроме того, если существуют процессы (хотя бы один), приславшие отстающие сообщения, то необходимо сделать откат до $\min\limits_m x_i^m$, где $I_i^{m \to n} = 1$ *и* $I_i^{x_i^m < x_i^n} = 1$. Откату соответствует вычитание величины $\max\limits_m (x_i^n - x_i^m)$.

Уравнения (6.1.1) соответствуют классической схеме оптимистического алгоритма, известной в литературе под названием Time Warp[4]. Данная модель легко модифицируется для схем с ограничением оптимизма и с периодическим сохранением состояний (6.2.1), (6.3.1).

Покажем, что время системы обязательно движется вперед. Обратимся к уравнениям (6.1.1). В случае отсутствия откатов все процессы на каждом шаге увеличивают значения своих локальных часов. Следовательно, $\min\limits_{n=\overline{1,N}} x_i^n$ тоже растет. Очевидно, что откатиться на $i+1$-м шаге любой процесс может до времени, нет меньшего, чем $\min\limits_{n=\overline{1,N}} x_i^n$. При этом, поскольку даже в случае отката любой процесс прибавляет положительную случайную величину ξ_i^n, а процесс со временем $\min\limits_{n=\overline{1,N}} x_i^n$ откатиться не может (так как не существует процесса с меньшим временем, который мог бы прислать отстающее сообщение), то $T_{i+1} > T_i$.

В случае отсутствия откатов развитие системы описывается уравнениями (5.1').

6.2 ОАС с временным окном

Является некоторым ограничением оптимизма. Не позволяет процессам "убегать" далеко вперед с целью уменьшения времени некорректного развития.

К входным параметрам добавляется размер временного окна, а для варианта (б) еще и алгоритм его пересчета.

$$\begin{cases} x_{i+1}^n = x_i^n + F^n(i)\xi_i^n - \max\limits_{m=\overline{1,N}}((x_i^n - x_i^m)I_i^{m\to n} I_i^{x_i^m < x_i^n}) \\ F^n(i) = \begin{cases} 1, \forall m = \overline{1,N} : x_i^n - x_i^m \le w_i^n \\ 0, \textit{иначе} \end{cases} \\ x_0^n = 0 \\ n = \overline{1,N} \end{cases} \qquad (6.2.1)$$

(а) Постоянное окно. $W_i = (w_i^1, w_i^2, \ldots, w_i^N) = W = (w^1, w^2, \ldots, w^N), \forall i \forall n : w_i^n > 0$.

(б) Перемещаемые окна. Размер окна W_i пересчитывается на каждом шаге.

Докажем, что время системы всегда будет продвигаться вперед. В отличие от классического ОАС не все процессы на каждом шаге могут продвигать свои часы. Однако процесс со временем $x_i^{\min} = \min\limits_{n=\overline{1,N}} x_i^n$ может это делать всегда. Действительно, поскольку $\forall i \forall n : w_i^n > 0$ и $\forall m \in \{1, \ldots, N\} : x_i^{\min} - x_i^m \le 0$, то $x_i^{\min} - x_i^m \le w_i^{\min}$ Отсюда следует, $T_{i+1} > T_i$.

6.3 ОАС с периодическим сохранением состояний

Для возможности организации отката назад необходимо сохранять состояния процессов. В целях экономии памяти и времени сохранение состояний происходит не на каждом шаге, а с некоторым периодом p_i^n. При этом значение периода 1 соответствует классическому ОАС. Пусть $\overline{X^n} = \{\overline{x_\tau^n}\}, n = \overline{1,N}, \tau = p_i^n j$ - последовательности сохраненных состояний процессов. Чтобы отразить в модели временные затраты на сохранение состояний, будем дополнительно на каждом шаге вычитать постоянные величины $\{t^n\}$ - затраты на сохранение одного состояния n-м процессом, выраженные в модельном времени. В случае возникновения отката для отражения временных затрат на восстановление состояния будем вычитать постоянные величины $\{\overline{t^n}\}$. Эти величины не учитывались при составлении уравнений для вышеописанных ОАС, чтобы не усложнять модель. При необходимости их можно ввести в любые уравнения для ОАС, поскольку это константы, то они никак не повлияют на вышеизложенные качественные выводы.

К входным параметрам добавляются времена сохранения/ восстановления состояний, период $(P_i = (p_i^1, p_i^2, \ldots, p_i^N))$, а для варианта (б) еще и алгоритм его пересчета.

$$\begin{cases} x_{i+1}^n = x_i^n + \xi_i^n - \max\limits_{m=\overline{1,N}}((x_i^n - \max\limits_{x_\tau^n \le x_i^n} \overline{x_\tau^n} + \overline{t^n})I_i^{m\to n} I_i^{x_i^m < x_i^n}) - Fp^n(i)t^n \\ Fp^n(i) = \begin{cases} 1, \; i = p_i^n j \\ 0, \; \textit{иначе} \end{cases} \\ \overline{x}_{\tau+1}^n = \begin{cases} x_i^n, \; \textit{если } Fp^n(i) = 1 \\ \overline{x}_\tau^n, \; \textit{иначе} \end{cases} \\ x_0^n = 0 \\ \overline{x}_0^n = 0 \\ n = \overline{1,N} \end{cases} \qquad (6.3.1)$$

Период сохранения состояний может:

(а) фиксироваться перед началом моделирования;

(б) пересчитываться после каждого сохраненного состояния.

В общем случае доказать обязательное продвижение модельного времени всей системы невозможно, так как в крайнем случае $\forall n \in \{1,\dots,N\}\,\forall i : p_i^n = \infty$ (то есть когда состояния не сохраняют) при откате каждый раз будет происходить возврат в ноль.

7. Метод сравнения эффективности различных АС

С помощью построенных выше моделей можно оценить время, до которого способна дойти пара (АС, приложение) за любое фиксированное число шагов. Это позволяет оценивать эффективность различных АС для различных приложений. Для этого необходимо:

1. Выбрать модели интересующих АС.
2. Задать параметры приложений.
3. Оценить значения T_l для всех пар. Чем больше T_l, тем эффективнее данный АС для данного приложения.
4. Выбрать наиболее эффективный для данного типа приложений АС.

Выделим основные параметры приложений, влияющие на эффективность АС:

- Разброс значений приращений локальных часов:

абсолютный - $\Delta T_{абс} = \max\limits_{n=1,N,i} M(\xi_i^n) - \min\limits_{n=1,N,i} M(\xi_i^n)$ и

относительный - $\Delta T_{отн} = \dfrac{\Delta T_{абс}}{\min\limits_{n=1,N,i} M(\xi_i^n)} 100\%$

- Интенсивность обмена сообщениями.
- Топология модели.
- Законы распределений приращений локальных часов.

Для получения значений T_l для различных пар необходимо промоделировать соответствующие случайные процессы методом статистических испытаний [4]. Для приложений с регулярными взаимодействиями были также подобраны простые для подсчета рекуррентные оценки для классического ОАС. Шаг I выбирается достаточно большим, таким, чтобы приложения "успели" продемонстрировать все режимы поведения.

Заключение.

Таким образом, предложенная математическая модель развития пары (АС, приложение) позволяет формализовать и исследовать характеристики алгоритмов синхронизации, выбрать оптимальный АС для заданного типа приложений. Представление развития таких пар в виде случайных процессов улучшает понимание их природы, упрощает анализ, сравнение и выбор оптимального алгоритма синхронизации.

Литература.

[1] Ю.Н. Казаков, Р.Л. Смелянский. Об организации распределенного имитационного моделирования. // Программирование №2 , 1994, с. 45-63.

[2] D.Mitra, I.Mitrani. Analysis and optimum performance of two message-passing parallel processors synchronized by rollback.//Performance'84, pp.35-50, 1984.

[3] Р. Райтер, Ж. Вальран "Распределенное имитационное моделирование дискретно-событийных систем."// М. - Мир, ТИИЭР, т. 77, № 1, янв. 1989, с. 245-262.

[4] В.Е. Гмурман. Теория вероятностей и математическая статистика.// М. – "Высшая школа", 1999, 480 страниц.

Раздел III

Прикладные программные системы

Новиков М.Д., Павлов Б.М.

Программный инструментарий идентификации динамических режимов нелинейных систем уравнений

1. В статье кратко описываются структура и возможности интерактивного пакета прикладных программ «Нелинейные динамические системы» (ППП «НДС»), предназначенного для численного исследования математических моделей (ММ) по технологии вычислительного эксперимента (ВЭ). Пакет реализован на ПК типа IBM-PC. Рассматриваются детерминированные динамические системы в форме обыкновенных дифференциальных уравнений и разностных уравнений (ОДУ и РУ), которые могут иметь запаздывающие аргументы.

Сформулируем математически строго задачу Коши для системы ОДУ размерности n с s параметрами и с m запаздывающими аргументами (для системы РУ задача Коши формулируется аналогично). Для краткости будем использовать векторную форму записи: $Y=(Y_1,Y_2,\ldots,Y_n)$ – вектор состояний (решений) системы, $F=(F_1,F_2,\ldots,F_n)$ – вектор правых частей, $G=(G_1,G_2,\ldots,G_n)$ – вектор начальных условий, $A=(A_1,A_2,\ldots,A_s)$- вектор параметров. Без ограничения общности положим начальное время t_0 равным 0.

$$Y'=F(t,A,Y,Y_{j_1}(t-\tau_1),\ldots,Y_{j_m}(t-\tau_m)),\ t\geq 0$$

начальные условия: $Y_i(t)=G_i(t)$ при $t\in[-\sigma_i,0]$.

Здесь t – непрерывное время, $Y'\equiv dY/dt$, $Y\equiv Y(t)$, $j_1,\ldots j_m\in\{1\ldots n\}$, Y_{j_i} – переменная j_i с запаздыванием τ_i, причём все τ_i – неотрицательные вещественные числа (т.е. производная Y′ явно зависит от m состояний в моменты времени $t-\tau_1,\ldots,t-\tau_m$). Числа σ_i определяются как max τ_L по всем L, для которых $j_L=i$; если такие τ_L отсутствуют (нет запаздываний по переменной Y_i), то $\sigma_i=0$. В частном случае m=0 система не имеет запаздывающих аргументов, тогда $Y(0)=G(0)$ – числовой вектор.

Наибольший интерес представляет поведение ММ на больших промежутках времени, когда при $t\to\infty$ решение системы «устанавливается», т.е. асимптотически выходит на определённый устойчивый режим, или, иначе говоря, после некоторых переходных состояний «самоорганизуется» во времени.

Интеллектуальное ядро пакета «Нелинейные динамические системы» составляют вычислительные модули, позволяющие визуализировать особенности численного решения и на основе заложенных теоретических знаний и эвристик уверенно идентифицировать установившийся во времени тип динамического режима изучаемой модели. Пакет может быть использован в научных исследованиях нелинейных систем невысокой размерности ($n\leq 4$); однако главное

его назначение – применение в университетском образовании: в компьютерных практикумах по математическому моделированию нелинейных процессов, изучаемых в естественных и гуманитарных науках.

Для учебных целей из научной литературы (по физике, химии, биологии, экологии, экономике, медицине и т.д.) нами отобраны и тщательно апробированы в различных динамических режимах математические модели, обладающие регулярным и хаотическим поведением (около 100 моделей); для студентов составлены задания практикума (исследовательские мини-проекты). Модели, задания и необходимые теоретические сведения из нелинейной динамики и синергетики описаны в учебных пособиях [1-3]. Пакет «НДС» с загруженным в него набором математических моделей и эталонных решений («ответов») и с комплектом учебно-методических материалов назван нами Автоматизированным Практикумом (АП) для предметной области «нелинейная динамика» и «синергетика».

Целями обучения с помощью АП на факультете ВМК МГУ являются:

ознакомление студентов с особенностями численных решений для нелинейных систем ОДУ и РУ путем наблюдений на экране ПК за развитием неустойчивостей и возникновением бифуркаций (ветвлений решений) при изменении управляющих параметров модели, за переходами по времени от порядка к хаосу и обратно с образованием аттракторов эволюции и фрактальных объектов (приобретение знания «know about»);

обучение студентов технологии осуществления вычислительного эксперимента на компьютере с акцентом на разработку дружественного интерфейса и модулей анализа численных решений (приобретение знания «know how»);

привитие вкуса к исследовательской, творческой работе на ПК;

развитие «нелинейного» (синергетического) стиля мышления, непривычного для классической (линейной) науки, на основе опыта, приобретаемого студентом в ходе выполнения индивидуального задания.

Предлагаемый метод обучения можно назвать «познавательными компьютерными играми с математическими моделями».

Разработанный АП не является автоматизированной обучающей системой (АОС), понимаемой в обычном смысле (см., например, [4]). Пакет «Нелинейные динамические системы» в составе АП следует отнести к классу компьютерных средств поддержки обучения (КСПО) в определённой области (учебной дисциплине), т.к. предполагается, что необходимые теоретические сведения студенты уже имеют (через лекции, семинары, учебники и учебные пособия). Известно, что КСПО в отличие от обычных АОС и, особенно, от интеллектуальных обучающих систем являются пассивными средствами, т.к. в диалоге «программная среда – обучаемый» инициатива полностью передана последнему: в разумных пределах он имеет свободу действий для самостоятельного развития своих навыков и творчества. Наиболее интересны и актуальны КСПО в форме исследовательской среды решения задач, ориентированной на определённую предметную (проблемную) область. В англоязычной литературе по компьютерной науке и

проектированию (CSE – Computer Science and Engineering) такая среда называется Problem Solving Environment (PSE) и считается конечным программным продуктом профессиональной деятельности. Разработка различных PSE должна находиться в фокусе CSE – так считают многие специалисты этой науки [5].

Пакет «НДС» является типичным PSE с языком и исследовательскими инструментами, характерными для предметной области. Другими особенностями пакета «НДС» (кроме качества КСПО и PSE) являются: а)возможность создания на его базе электронного задачника благодаря средствам добавления, модификации и удаления новых моделей и эталонных примеров (ответов) в базе данных пакета; б)благодаря средствам автоматического показа и анимации накопленных графических образов пакет обретает качество электронного демонстратора математических образов в статике и динамике. Таким образом, пакет «НДС» – это КСПО типа PSE с дополнительными свойствами электронного задачника и автодемонстратора.

Разработанные задания для рассматриваемого практикума можно выполнять без программирования и без специализированного пакета «НДС», если воспользоваться одним из коммерческих пакетов общего назначения типа PSE: DERIVE, Matlab, Mathematica, MathCAD, Maple и др. Однако эти пакеты достаточно громоздки, требуют от ПК больших ресурсов, недёшевы и сложны в использовании. Их применимость в учебном процессе ограничена и по другой причине: в таких пакетах, как правило, не предусмотрена возможность анализа промежуточных и итоговых результатов, отсутствует контроль за ходом решения задачи, нет хорошей системы помощи и диагностики ошибок пользователя, слабо используется трёхмерная графика, анимация и средства мультимедиа, пользовательский интерфейс далеко не дружествен. Из-за указанных недостатков затруднительно осуществлять полноценные ВЭ с математическими моделями двух рассматриваемых классов.

С развитием сети Интернет и информационной среды WWW на ряде сайтов появились полезные для учебных целей в предметной области «Нелинейная динамика и синергетика» программные продукты, распространяемые платно и бесплатно. Отметим несколько таких продуктов, созданных для ПК типа IBM-PC.

Пакет FRACTINT, созданный коллективом разработчиков в США, ориентирован на эффектную демонстрацию различных фрактальных объектов, в том числе странных аттракторов. Распространяется бесплатно.

Программная коллекция «Chaos» для ПК (авторы Х.Корш и Х.Йодль, университет в г.Кайзерслаутерн, Германия, 1999 г.) представляет на CD-ROM набор исполнимых программ для проведения вычислительных экспериментов и содержит вводные тексты по теории хаоса и его компьютерного моделирования.

Пакет DYNAMICS (авторы Х. Нюссе, университет в г. Гронинген, Нидерланды, и Дж. Йорке, университет в Мэриленде, США, 1998 г.) предназначен для проведения вычислительных экспериментов в области нелинейной динамики и хаоса; имеется справочное руководство. Пакет удобен для исследовательских целей и обучения благодаря дружественной системе меню, системе помощи пользователю и использованию иллюстративной графики. Имеется возможность добавлять и анализировать собственные модели пользователя (кроме моделей, встроенных в пакет).

В учебном руководстве Дж. Корна (университет Аризоны, США, 1998 г.). «Численное проникновение в суть динамических систем» рассматриваются интерактивные имитационные эксперименты. Они применяются для решения дифференциальных уравнений, возникающих при исследовании аэрокосмических аппаратов, силовых агрегатов, химических процессов, а также в популяционной динамике и физиологии.

2. Программный комплекс в составе АП по нелинсйной динамике (синергетике) является, по существу, пакетом прикладных программ (ППП) со своим входным языком, пользовательским диалоговым интерфейсом, системным и функциональным наполнением и базой данных. Входной язык служит для ввода в пакет любой математической модели допустимых классов, задания начальных условий и значений параметров модели, для генерации и вывода решений и демонстрационных примеров. Диалог пользователя (студента, преподавателя) ведется на русском языке в понятиях, характерных для данной предметной области и технологии вычислительного эксперимента. К системным компонентам программного комплекса относятся: работа с базой данных (ввод, коррекция и удаление модели или эталонного решения, создание мультипликации, галереи графических образов и др.), организация навигации по пунктам иерархических меню, обработка унифицированных форм экранных шаблонов для ввода-вывода данных и др. К функциональным компонентам относятся вычислительные алгоритмы - методы Рунге-Кутта, интерполяция для запаздывающих аргументов, построение графиков и множеств точек на экране компьютера, обработка временных рядов, распознавание финальных образов и построение различных бифуркационных диаграмм.

Основу дружественного интерфейса «пользователь – пакет» составляет унифицированная, практически общая для ОДУ и разностных уравнений (итерируемых отображений) система меню. которая имеет несколько уровней.

МЕНЮ-1 позволяет выбрать для исследования один из двух классов систем (ОДУ или разностные уравнения), в том числе и с запаздывающими аргументами; последние автоматически распознаются пакетом.

МЕНЮ-2 позволяет задать один из трёх режимов работ для выбранного класса систем уравнений: 1)выбрать модель, введённую ранее, 2)ввести новую модель в банк моделей, 3)продемонстрировать в автоматическом режиме все накопленные графические образы для ряда моделей (автодемонстрация). В последнем режиме допускается вращение трехмерного объекта по заданному закону (например, странного аттрактора - для выяснения его детальной структуры).

МЕНЮ-3 предназначено для задания начальных работ с конкретной моделью – вызванной из базы данных или только что введённой; отсюда же происходит обращение к построению различных бифуркационных диаграмм.

МЕНЮ-4 дает возможность либо начать численное исследование выбранной модели, либо осуществить показ эталонных решений, генерируемых пакетом автоматически по заранее заданным шаблонам.

МЕНЮ-5 составляют пункты основных работ при численном исследовании ММ. Это «рабочее» меню возникает сразу после окончания очередного кванта расчета (интегрирования системы ОДУ или итерирования системы РУ) и содержит обращение к инструментальным средствам анализа полученного отрезка

временного ряда $Y_1(t),Y_2(t),\ldots$, где t – непрерывное время для ОДУ и дискретное (t=N -- номер итерации) для разностных уравнений. Многие пункты МЕНЮ-5 имеют собственные меню (подменю) для задания конкретных характеристик обработки данных (например, построение диаграммы Ламерея для одномерных РУ или множества точек в сечении Пуанкаре для трёхмерных ОДУ).

Компонентой входного языка является набор стандартных (для данного пакета) форм – шаблонов размещения и форматов данных: шаблона для ввода конкретной модели, шаблона для создания эталонного примера, шаблонов для создания различных бифуркационных диаграмм.

Дружественность интерфейса обеспечивается также легко понятной системой помощи (HELP) пользователю при его некорректных действиях или затруднениях.

Пакет написан на языке Паскаль для среды DOS, требует около 800 кб дисковой памяти; может работать и в среде Windows 3.1x или Windows-95. В среде Windows-NT пакет работает только при запуске из оболочки Norton (Volkov) Commander. Для работы пакета необходим TPC-компилятор (файл tpc.exe и turbo.tpl). Поддерживается работа с мышью.

Пакет прост в установке и эксплуатации. Требования к компьютеру минимальные: 1)достаточно 5 - 10 мб свободной памяти на жестком диске для хранения результатов интегрирования и данных для бифуркационных диаграмм; 2)тип процессора - 286 или выше; 3)объем оперативной памяти - от 1 мб. Пакет может быть размещен как на отдельном компьютере, так и на файл-сервере локальной вычислительной сети в исполнимом виде; при наличии в вузе выхода в Интернет может быть использован в дистанционном обучении (самообучении). Пакет запускается на выполнение командой "DIFF".

Подробное описание архитектуры пакета и его возможности в составе АП по нелинейной динамике (синергетике) дано в [3].

3. В программный инструментарий анализа и идентификации различных динамических режимов (равновесие, неограниченный рост решения, периодические колебания, квазипериодические колебания, детерминированный хаос) входят следующие модули, которые используют данные $Y_i(t)$, i=1,2,3,4 на любом расчетном отрезке $[t_{нач.}, t_{кон.}]$:

- построение двумерных и трехмерных фазовых портретов;
- «бегущие» осциллограммы решений $Y_i(t)$;
- бифуркационные диаграммы при вариации одного из параметров ММ;
- вычисление и построение точек – следов фазовых траекторий в заданных сечениях (карты Пуанкаре) для трехмерных систем;
- построение таблиц и графиков экстремумов $Y_i(t)$ с определением периода;
- построение множества точек $\{Y_1,Y_2\},\{Y_2,Y_3\},\{Y_3,Y_4\},\ldots$, где Y_k – последовательные экстремумы $Y_i(t)$;
- построение «лестницы» Ламерея для одномерных разностных уравнений;
- вычисление показателей Ляпунова для трехмерных систем ОДУ;
- построение области характерных режимов поведения ММ при вариации двух параметров или двух начальных условий (карта режимов);
- вычисление автокорреляционной и спектральной функций с помощью алгоритма быстрого преобразования Фурье и построение соответствующих графиков;

• «анимация» - вращение 3D-графика по заданному закону и эволюция аттракторов при изменении одного из параметров.

Некоторые возможности пакета «НДС» по идентификации хаотической динамики проиллюстрированы на рис. 1-4 для широко известной системы Э.Лоренца (см., например, [6]):

$Y_1'=S*(Y_2-Y_1)$, $Y_2'=R*Y_1-Y_2-Y_1*Y_3$, $Y_3'=Y_1*Y_2-B*Y_3$,

имеющей при $1<R<\infty$, S=10, B=8/3 регулярные и хаотические решения. Для R=28 на рис. 1-3, соответственно, изображены: «бабочка» Лоренца (странный аттрактор), характерное множество точек в сечении Пуанкаре и спектральная функция. На рис. 4 изображена бифуркационная диаграмма – по оси ординат отложены все экстремумы решения $Y_1(t)$ для значений параметра R от 20 до 180.

Литература

1. Павлов Б.М. Вычислительный практикум по исследованию нелинейных динамических систем (самоорганизация во времени). М.: МГУ, 1996.
2. Павлов Б.М., Новиков М.Д. Вычислительный практикум по исследованию нелинейных разностных систем (итерируемых отображений). - М.: МГУ, 1998.
3. Павлов Б.М., Новиков М.Д. Автоматизированный практикум по нелинейной динамике (синергетике). – М.: МГУ, 2000.
4. Довгялло А.М., Ющенко Е.Л. Обучающие системы нового поколения // Управляющие системы и машины, 1988, №1, с.83-86.
5. Gallopoules E., Sameh A. CSE: Content and Product. //IEEE Computational Science & Engineering, 1997, v.4, №2, pp.39-43.
6. Лоскутов А.Ю., Михайлов А.С. Введение в синергетику. –М.: Наука, 1990.

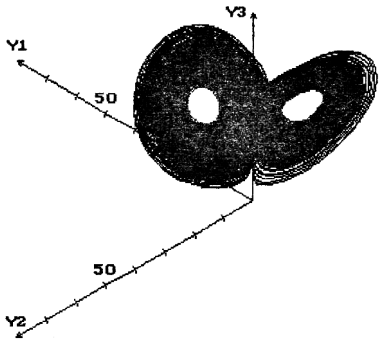


Рис. 1.

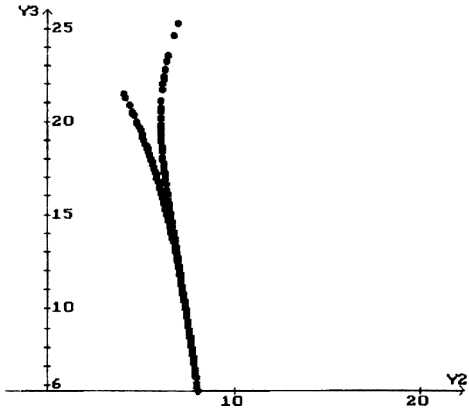


Рис 2.

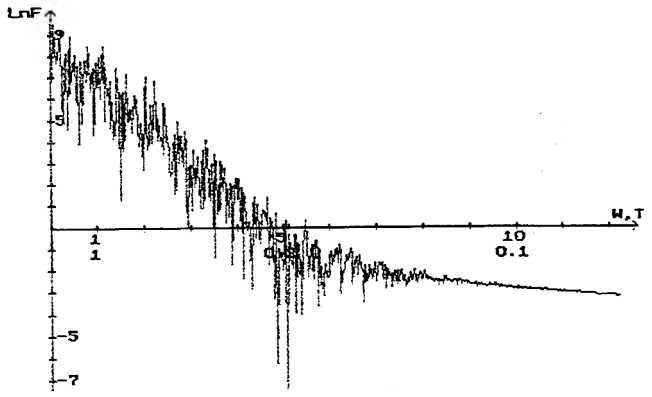


Рис. 3.

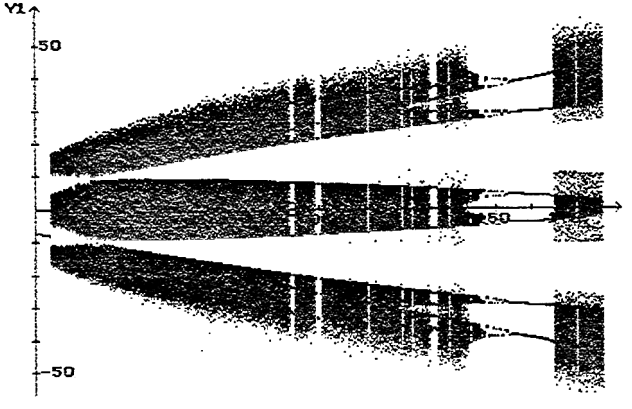


Рис. 4.

Рогов Е. В.

Интернет-организация Системы Анализа Динамических Процессов[1]

Введение

В докладе рассматривается архитектура и организация Системы Анализа Динамических Процессов (далее для краткости называемой «Система»). Система предназначена для анализа процессов, представимых в виде многопараметрических временных рядов.

Процессы могут иметь самую разнообразную физическую природу. В качестве примера можно привести поведение плазмы в установке «токамак», изменение котировок акций на фондовой бирже, результаты гидрометеорологических наблюдений и т. д.

Основной интерес представляют «плохо формализованные» процессы, законы поведения которых неизвестны или недостаточно изучены. В таком случае их изучение сводится к анализу накопленной статистики поведения процессов той же природы.

В Системе предпринята попытка собрать представительный набор методов и алгоритмов анализа подобных процессов. Учитывая известную информацию, такую, как объем имеющейся выборки данных, число параметров процесса, наличие зашумленности, точность измерения и т. п., Система должна предлагать исследователю адекватные методы анализа. Таким образом, Система является в определенной мере интеллектуальной.

В Системе широко применяются нейронные сети и генетические алгоритмы. Являясь мощным инструментом изучения «плохо формализованных» процессов в условиях неприменение точных математических методов, они обеспечивают и независимость алгоритмов анализа от конкретной предметной области. В то же время использование нейросетей и генетических алгоритмов является крайне ресурсоемким, что делает актуальной задачу их распараллеливания для вычислений на многопроцессорных системах.

Отсутствие привязки к предметной области позволяет исследователю применять инструментарий Системы для решения широкого круга задач, а возможность автоматического выбора приемлемых методов и параметров не потребует от него глубоких специальных знаний по нейросетям и генетическим алгоритмам.

Система разрабатывается коллективом сотрудников, аспирантов и студентов факультета ВМК МГУ им. М. В. Ломоносова. Это определяет еще одну задачу Системы — служить своеобразным «учебным пособием» для студентов, работающих с нейросетями и генетическими алгоритмами. В Систему легко добавляются новые методы анализа, что позволяет интегрировать разработки, обычно являющиеся разрозненными и несовместимыми друг с другом.

---

1 Работа выполнена при поддержке гранта РФФИ № 99-07-90229

Общая структура

Структурно Система состоит из трех частей: рабочего стола, базы данных о процессах и набора вычислительных методов.

Рабочий стол — интерфейсная часть, отвечающая за взаимодействие с пользователем и визуализацию данных. Рабочий стол позволяет пользователю манипулировать данными и производить их анализ с помощью расположенного на нем инструментария. При этом рабочий стол может настраиваться на пользователей с различными научными интересами.

База данных хранит описание исследуемых процессов, каждый из которых определяется значениями своих параметров, взятых в определенные моменты времени. Дополнительно хранятся свойства, относящиеся к процессу в целом, такие, как оценка качества процесса, принадлежность его какому-либо классу.

Вычислительные методы можно разделить на три категории: методы предварительной обработки (фильтрация шумов, приведение к общей временной шкале, восстановление пропущенных данных и т. п.), стандартные методы, ставшие классическими и присутствующие в различных системах и пакетах обработки данных (такие, как методы статистического анализа, математической статистики, анализ с использованием преобразований Фурье и wavelet) и, наконец, в качестве основного инструмента изучения динамических процессов Система предоставляет нейросети и генетические алгоритмы.

Реализация в Интернете

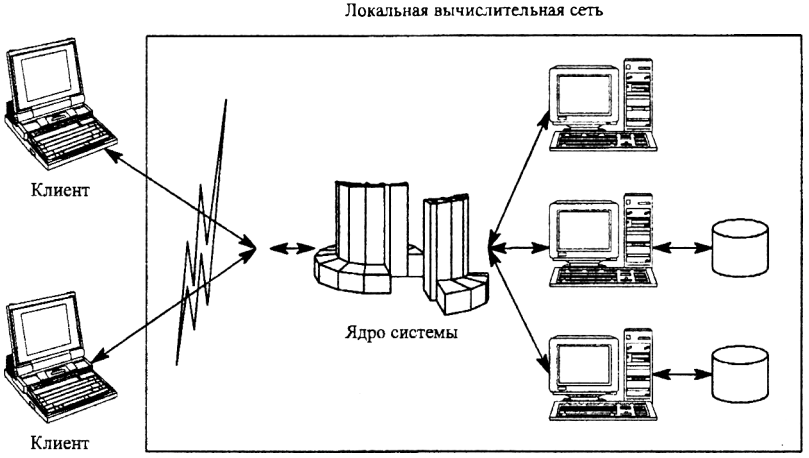


Рис. 1. Аппаратная архитектура Системы

Разрабатываемая реализация Системы предназначается для использования в сети Интернет. При этом для пользователей Система является сервером, предоставляющим методы анализа и базу данных, используя web-интерфейс. Серверная

часть Системы является масштабируемой и состоит из ряда вычислительных машин, объединенных в локальную сеть.

Такой подход дает удаленным пользователям возможность работать со своими данными, используя вычислительные мощности сервера Системы, что особенно важно при работе с такими ресурсоемким методами, как нейросети и генетические алгоритмы. Фактически, пользователю требуется лишь выход в Интернет и браузер с поддержкой языка Java. Возможна совместный анализ одних и тех же данных несколькими пользователями.

Учитывая обилие всевозможных программно-аппаратных архитектур, Система реализуется так, чтобы быть платформонезависимой всюду, где это не противоречит эффективности.

Программная архитектура

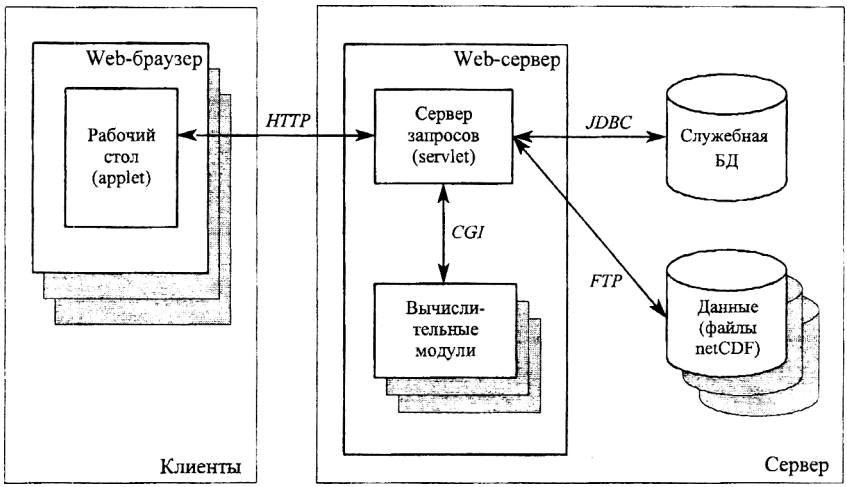


Рис. 2. Программная архитектура Системы

Система имеет трехзвенную архитектуру клиент-сервер. *Клиентская часть* Системы представляет собой Java-апплет и выполняется web-браузером. Основная задача клиентской части состоит в организации пользовательского интерфейса. Это — первое звено архитектуры.

Вторым, промежуточным звеном является *ядро Системы*, через которое идет весь информационный обмен между клиентской и серверной частями. Ядро реализовано в виде Java-сервлета.

Наконец, третье звено — *вычислительные методы и база данных*. Вычислительные методы реализуются как CGI-программы, в качестве хранилища данных используются реляционная СУБД и файловая система.

Взаимодействие происходит следующим образом.

Клиентская часть посылает ядру все информационные и вычислительные запросы. Они могут содержать имя пользователя Системы и его пароль, тип запроса (соединение с сервером, получение данных, выполнение вычислений и т. п.), идентификаторы необходимых данных и вычислительных методов. Ядро разбирает поступивший запрос, выполняет его, обращаясь к конкретным вычислительным серверам и серверам баз данных и возвращает результаты клиенту.

Плюсы такой организации состоят в том, что внутренняя структура сервера скрыта от клиента. Это позволяет изменять структуру хранения данных, добавлять и перемещать вычислительные методы, производя лишь настройку ядра Системы, без каких бы то ни было изменений в клиентской части. Кроме того, нет необходимости пересылать по сети Интернет пароли для доступа к компонентам серверной части, уменьшая тем самым безопасность Системы.

Рассмотрим подробнее функции и детали реализации перечисленных модулей Системы.

Рабочий стол

Рабочий стол Системы написан в виде апплета на языке Java, что обеспечивает независимость от программно-аппаратной платформы пользователя. При внесении изменений в клиентскую часть разработчикам достаточно выложить новую версию апплета на сервер, и она автоматически будет загружена очередному пользователю. При этом от пользователя не требуется выполнения традиционной процедуры инсталляции. Ему достаточно указать в браузере, поддерживающем Java, адрес web-сервера Системы.

Функционально, рабочий стол обеспечивает графический пользовательский интерфейс, взаимодействие с ядром Системы для получения данных и выполнения вычислений, а также отображает данные.

Система поддерживает группы пользователей, отличающихся друг от друга используемым инструментарием и исследуемыми данными. При загрузке апплет производит идентификацию пользователя по его имени и паролю. Идентификация используется для получения от сервера информации, необходимой для соответствующей настройки рабочего стола.

Для организации обмена с ядром Системы применяется механизм сериализации, позволяющий организовать взаимодействие в терминах объектов Java.

Для визуализации исходных данных и результатов вычислений используются графические возможности языка Java. Система оперирует типизированными данными, и для различных типов существуют классы, реализующие их отображение.

База данных

Данные, хранимые в Системе, разделяются на исходные и производные, а также на служебную информацию.

Исходные и производные данные составляют информацию о предметной области и являются описаниями многопараметрических временных процессов вида

$P = \{ P(t_1), P(t_2),\ldots, P(t_N) \}$.

В точке t_i вектор параметров процесса $P(t_i) = (p_1(t_i), p_2(t_i),\ldots, p_n(t_i))$. Точки регистрации параметров t_1, $t_2,\ldots$, t_N образуют временную шкалу, на которой задан процесс, которая может быть как равномерной, так и неравномерной.

Исходные данные поступают в Систему извне, например, через Интернет из открытой базы данных. Производные данные являются локальными для пользователей и образуются из исходных в процессе работы над ними, как результат ограничивающей выборки или применения к ним вычислительных методов. Существует также возможность генерации процесса заданного пользователем вида, который может использоваться в отладочных целях при создании новых методов анализа.

Таким образом, исходные и производные данные характеризуются простой структурой, но имеют большой объем, что требует их распределения по узлам сети. Для хранения исходных и производных данных используется формат представления научных данных NetCDF.

Служебная информация содержит данные о пользователях Системы, каталог хранящихся описаний процессов, список узлов сети Системы и имеющихся на них вычислительных методов.

В отличие от основных данных служебная информация имеет достаточно сложную внутреннюю структуру и небольшой размер. Она хранится в служебной реляционной базе данных, в качестве которой в настоящее время используется Postgres95.

Методы обработки

Модули Системы, реализующие определенные вычислительные методы, используют интерфейс CGI. Таким образом, они работают под управлением web-сервера. Являясь исполняемой программой, модуль представляет эффективный, хотя и зависимый от платформы способ вычислений. В данном случае такая зависимость не является серьезным недостатком, т. к. число используемых в сети Системы платформ невелико.

Широкое применение ресурсоемких методов, использующих нейросети и генетические алгоритмы, обуславливает организацию распределения вычислений. На узлах сети Системы располагаются модули, реализующие предоставляемые методы и скомпилированные для данной платформы. Ядро Системы, анализируя информацию из служебной базы данных, определяет, на какой узел можно разместить вычислительный запрос, и выбирает исполнителя. В качестве основного критерия выбора используется критерий наименьшей загруженности узла.

Кроме того, часть методов могут быть специально распараллелены для выполнения на многопроцессорной или многомашинной системе.

Помимо предопределенного набора вычислительных методов, использующих нейросетевые алгоритмы, Система предоставляет возможность использования нейросетевых эмуляторов для создания, тренировки и проверки работоспособности своих собственных нейросетей.

В настоящее время для этой цели используется Штутгардский нейросетевой эмулятор SNNS, имеющий открытую архитектуру и программный интерфейс.

Ядро Системы

Ядро осуществляет весь информационный обмен между клиентской и серверной частями Системы. Ядро представляет собой сервлет и написано на языке Java. Использование Java позволяет ядру просто и эффективно взаимодействовать со всеми компонентами Системы.

Для обмена с клиентским апплетом, также написанным на Java, используется механизм сериализации объектов. Взаимодействие с вычислительными методами осуществляется через web-сервер. Доступ к исходным и производным данным происходит по протоколу FTP, а к служебной базе данных — с помощью интерфейса JDBC.

В функции ядра Системы входит принятие запросов от клиентов, их обработка и возвращение результатов. Для этого ядро переадресовывает запросы на выборку данных к узлу, на котором эти данные хранятся, размещает производные данные, полученные во время работы. Также на ядро возложено распределение вычислений по узлам сети.

Заключение

Коллективом сотрудников, аспирантов и студентов факультета ВМК МГУ разрабатывается Система Анализа Динамических Процессов, предназначенная для исследования «плохо формализованных» многопараметрических процессов. Особенностями Системы является независимость инструментария от конкретной предметной области, для чего наряду с традиционными методами широко используются нейронные сети и генетические алгоритмы, а также определенный интеллект, позволяющий исследователю выбирать адекватные методы анализа, не являясь специалистом в области нейросетей и генетических алгоритмов.

Система предназначена для функционирования в сети Интернет и предоставляет доступ к распределенной базе данных и вычислительным ресурсам через web-интерфейс. Основная часть Системы написана на языке Java.

Литература.

[1]. **Н. М. Ершов, Л. Н. Королев, Э. Э. Малютина, А. М. Попов, Н. Н. Попова.** Применение активных баз данных в прогнозировании. — Вестн. Моск. ун-та. Сер. 15, Вычисл. матем. и киберн., 1998, № 1.

[2]. **Л. Н. Королев, А. М. Попов, Н. Н. Попова, Е. В. Рогов.** Реализация системы анализа динамических процессов в сети Интернет. — В кн.: Научный сервис в сети Интернет: Тезисы докладов Всероссийской научной конференции (20—25 сентября 1999 г., г. Новороссийск). — М.: Изд-во МГУ, 1999, с. 162—167.

Эдельман Л.В.

Схема реализации СООБД сетевой структуры на базе COM

Аннотация

В статье предложена схема реализации оригинальной системы обьектно-ориентированных баз данных сетевой структуры на основе технологии COM, которая позволит решить ряд проблем существующих СООБД. Рассмотрены основные отличительные черты предложенной схемы и наиболее важные технические аспекты реализации.

1. Введение

В настоящее время большое внимание уделяется разработке и исследованию систем обьекто-ориентированных баз данных (СООБД). Однако, по нашему мнению, существующие на данный момент подходы к их реализации страдают целым рядом недостатков. В данной статье описаны принципы и решения, на основе которых предлагается реализовать оригинальную СООБД сетевой структуры на основе технологии COM. Мы полагаем, что предлагаемые решения решают ряд проблем.

Во второй части обсуждаются основные цели, поставленные в ходе проекта, разьясняется понятие "СООБД сетевой структуры". В третьей части рассматриваются наиболее важные аспекты предложенной схемы реализации.

2. Цели проекта

Предлагаемые на сегодняшний момент модели СООБД (включая ODMG, фактически являющуюся стандартом) обладают, на наш взгляд, одним весьма существенным недостатком, рождающим целый ряд проблем: будучи моделями для хранения обьектно-ориентированных данных, они сами не являются обьектно-ориентированными. То есть представляют собой единый, неразрывный монолит, включающий в себя ядро, модель данных, язык для выполнения запросов, язык программирования (или связывание с каким-либо из распространенных языков) и т.п. Это приводит к тому, что СООБД получается очень тяжеловесной, требует больших затрат как на программирование самой СООБД, так и на работу с ней. Как хорошо известно из теории программирования, уменьшение числа связей между частями программы способствует улучшению этой программы чуть ли не во всех возможных смыслах - в частности, повышает её надежность, читаемость, модифицируемость. В нашем представлении, СООБД должна быть разбита на некоторое количество отдельных компонентов (обьектов), которые взаимодействуют друг с другом исключительно через заранее определенные интерфейсы. Это, с одной стороны, снизит число внутренних связей, и с другой - позволит сделать часть из таких компонент легко заменяемыми. Скажем, ядро может не зависеть от способа физического хранения данных (в памяти, в файле, в наборе файлов, распределенно) - все, что ядру потребуется, это обьект, реализующий требуемый интерфейс.

Очень важным здесь мы считаем отделить модель данных от СООБД. До сих пор модель данных была неразрывной частью самой СООБД. Для задания схемы БД (детального описания всех обьектов) в СООБД определялся специальный язык.

В состав СООБД приходилось включать транслятор этого языка, средства отображения конструктов этого языка в прикладные ЯП. Любое изменение в схеме БД требовало перекомпиляции использующих БД программ. Мы предлагаем отказаться от всего этого, тем самым существенно упростив СООБД. В качестве модели данных предлагается использовать COM - обьектную модель ОС Windows. (С таким же успехом можно было бы использовать модель CORBA). Всю необходимую информацию об обьектах (о том, как их записывать и считывать, какие у них есть поля, методы и связи) СООБД получает от самих обьектов через соответствующие интерфейсы. Схема БД может изменяться "на лету" и это не требует какой-либо перекомпиляции. Наконец, предполагается, что программирование обьектов, которые могут храниться в БД, практически не требует знания об этой БД вообще. В частности, эти обьекты сами никогда не будут обращаться к интерфейсам БД и не будут знать, что они хранятся в этой БД.

С учетом всего вышеизложенного предлагаемая модель СООБД схематически выглядит следующим образом (Рис.1). Особо отметим, что все компоненты БД – ядро, носитель данных, язык запросов – являются наборами COM-обьектов.

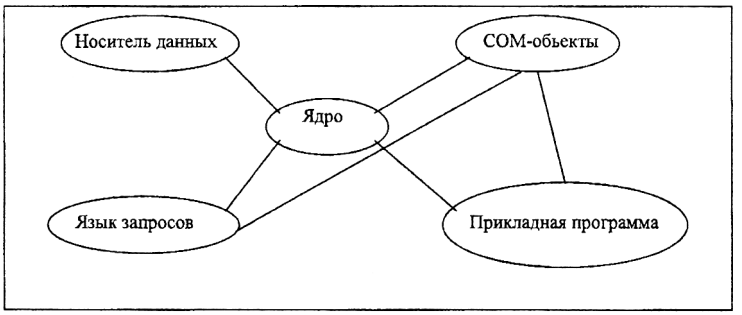


Рис.1 Модель СООБД

Существующие на данный момент СООБД делятся на собственно обьектно-ориентированные БД (в чистом виде) и на обьектно-реляционные БД, представляющие собой компромисс между реляционной и обьектно-ориентированной моделью данных. Предлагаемая СООБД ни относится ни к тому, ни к другому типу, а является СООБД сетевой структуры, то есть компромиссом между сетевой и обьектно-ориентированной моделью данных. Мы предлагаем ограничить типы возможных связей между обьектами упорядоченными связями вида "один ко многим". Это, безусловно, является сужением модели данных по сравнению с чисто обьектно-ориентированными моделями. Однако мы полагаем, что это сужение не скажется существенным образом на функциональности БД, в то же время мы получаем ряд преимуществ, присущих сетевой модели данных: её относительную простоту при больших возможностях, эффективность хранения связей между обьектами, скорость работы с БД, легкость отображения на реляционную модель данных. При этом мы отдаем себе отчет в том, что сетевая модель данных более подходяща для баз данных, ориентированных не на конечного пользователя, а на программы, ис-

пользующие БД для хранения разнообразных внутренних данных. Соответственно, в разработке СООБД мы в меньшей степени ориентируемся на использование её конечным пользователем - поэтому, в частности, высокоуровневому языку запроса не уделяем большого внимания.

Наконец, мы хотим, чтобы СООБД была как можно более гибкой и удовлетворяла как можно более разнообразные потребности (то есть была настраиваемой под специфические потребности конкретной прикладной программы). Поэтому СООБД будет поддерживать различные схемы синхронизации (пессимистическую и оптимистическую), различные схемы одновременного доступа к БД (схему конкурентного доступа и схему совместной работы нескольких процессов), возможность различных высокоуровневых языков запроса (OQL и SQL), возможность распределенного хранения данных. Наконец, СООБД должна уметь работать с большими объектами произвольного размера.

3. Схема реализации

3.1Обьекты. Агрегирование. Стабильность

Единицей представления данных в предлагаемой СООБД является COM-обьект. Сразу отметим, что, очевидно, не любой COM-обьект может стать обьектом базы данных. Он должен обладать минимальным набором необходимых интерфейсов – например, интерфейсом для сохранения своего состояния в постоянной памяти и считывания из неё, соответственно - другими словами, интерфейсом, обеспечивающим стабильность обьекта. Наша цель – свести к минимуму требования к COM-обьекту.

Уже было сказано, что мы полагаем очень существенным то, что обьект по возможности не должен ничего знать о базе данных. В то же время прикладная программа, использующая базу данных, должна отличать обьекты базы данных от всех прочих, и эти обьекты должны предоставлять ей соответствующие услуги - скажем, записать обьект в БД, установить на него захват, получить его уникальный идентификатор (ID) в базе данных. Возможно, конечно, реализовать все эти услуги через интерфейсы ядра СООБД – но это будет и странно с точки зрения обьектной архитектуры (о свойствах обьекта спрашивать не у самого обьекта, а у кого-то третьего), и неэффективно (не обойтись без хэш-таблицы либо дерева, отображающего адреса находящихся в памяти обьектов БД на их идентификаторы). Но COM предоставляет другой способ – гораздо более элегантный. Способ этот называется агрегирование. Агрегирование – это возможность включить COM-обьект (агрегированный обьект) в состав другого (внешний контроллер), интерфейсы агрегированного обьекта дополняются (либо замещаются) интерфейсами контроллера. При этом прикладная программа воспринимает агрегированный обьект и контроллер как единое целое (агрегат). Что нам это дает? Мы не требуем от обьекта реализовывать какие-либо специфические для базы данных интерфейсы. Вместо этого мы каждый обьект из БД «агрегируем» в контроллер, реализующий эти интерфейсы. Контроллер реализуется в ядре СООБД – ни тем, кто программирует обьекты, ни тем, кто использует БД в прикладных программах, про него знать не нужно. При этом от обьектов требуется только одно: они должны поддерживать возможность агрегирования. Требование не слишком обременительное. Что до прикладных программ, им всего лишь нужно знать, что обьект, считанный ими из БД, будет иметь

интерфейсы, специфические для работы с ним как с объектом БД, независимо от его типа. Объект же того же типа, созданный вне БД, этих интерфейсов иметь просто не будет.

Как мы уже говорили, объект должен уметь записывать свое состояние в постоянную память и считывать его. Здесь можно придумать как собственный интерфейс, максимально подходящий именно для записи в БД – но подойдет и какой-либо из стандартных интерфейсов от Microsoft, IPersistStream (лучше всего), IPersistStorage (хуже, но для больших объектов переменного размера подойдет) или даже IDispatch (в этом случае можно полагать, что состояние объекта равно множеству значений его полей). В идеале, СООБД может работать с объектом, реализующим любой из этих интерфейсов – хотя бы один.

3.2 Множества

Уметь хранить объекты – это хорошо, но объектная модель СООБД была бы очень половинчатой без возможности хранить также связи между объектами. В то же время, полагаем, достаточно ограничить эту возможность рамками сетевой модели данных. То есть допускать только упорядоченные связи вида «один ко многим».

Очевидно, если между объектами существует связь, то она существует независимо от БД – то есть объект для каждого типа связи должен иметь интерфейс, через который мы можем перечислить все связанные с ним объекты, добавить новые связи либо разорвать их. Если же объект хранится в БД, то все связанные с ним также должны быть объектами БД (то есть их нужно «агрегировать» нашим контроллером при создании, а при добавлении и удалении связей вносить соответствующие изменения в БД). При этом мы по-прежнему не хотим, чтобы объект что-либо знал о БД. И опять на помощь нам приходит агрегирование. Контроллер просто должен замещать соответствующий интерфейс, вызывая методы «родного» интерфейса тогда, когда необходимо. Здесь есть только одно «но»: чтобы это было возможным, интерфейс, перечисляющий связи, должен быть стандартным – иначе его будет невозможно «перехватить».

Заметим, что можно было бы допустить и связи вида «много ко многим», но для них не существует метода хранения, требующего фиксированного количества памяти на каждый объект.

3.3 Модель транзакций. Синхронизация

Разрабатывая модель транзакций, мы стремились охватить как можно больший круг возможных применений СУБД, сделать модель гибкой. Учитывалась как схема конкуретной работы с БД нескольких процессов, так и схема совместной работы.

Основные понятия предлагаемой модели – контекст и транзакция. Контекст – это, по сути, локальная копия БД. Вся работа с БД происходит внутри контекста. Все изменения, вносимые в БД, в том же контексте видны сразу же, в остальных – только после синхронизации. Внутри контекста каждый объект из БД в память считывается единожды и умеет, тем самым, уникальный адрес в памяти. По сути, контекст аналогичен процессу – за учетом того, что благодаря средствам COM контекст может включать в себя несколько физических процессов, и наоборот. Так как COM позволяет передавать указатели на объекты через границы процессов, ничто не мешает прикладной программе видеть БД из любого числа контекстов одновре-

менно. Т.о., корректная работа с контекстами остается на совести программиста, использующего БД – СООБД только предоставляет необходимые для того средства.

Под транзакцией мы понимаем минимальную последовательность операций с БД, оставляющую БД в логически консистентом виде. Желательно, чтобы транзакции были короткими. Внутри контекста можно одновременно открыть только одну транзакцию. Операции записи можно делать только внутри транзакции. В то же время захват обьектов в любом режиме можно осуществлять как внутри транзакции, так и вне её. Если перед записью не был установлен захват, он устанавливается автоматически. Захваты, установленные внутри транзакции, автоматически снимаются по ее окончании. Синхронизация от контекста к собственно БД происходит по окончании транзакции. Синхронизация от БД к контексту (если изменения внесены транзакцией из другого контекста) происходит сразу же, если контекст не находится в состоянии транзакции, иначе – по её окончанию. Таким образом, транзакции сериализуются, но при этом порядок их выполнения относительно различных контекстов может не совпадать. Схема синхронизации на уровне транзакции может быть как оптимистической (захваты запоминаются только в контексте, реально происходят только в момент синхронизации - и в случае неудачи следует откат), так и пессимистической (захватываются сразу обьекты из самой БД). Захваты прямо из контекста всегда работают по пессимистической схеме.

Прежде всего – почему такое ограничение, только одна одновременная транзакция внутри контекста? Очевидно, транзакции должны быть сериализуемыми и непрозрачными друг для друга – поэтому понятие «транзакция» похоже на понятие «контекст». Вводить два уровня «непрозрачности» - ненужное усложнение. С другой стороны, требовать, чтобы для каждой транзакции, если хочется запустить их одновременно, создавался свой контекст – не противоречит ли это схеме совместной работы с БД? Представим, что совместная работа с БД заключается в конвейерной обработке данных несколькими процессами. Как только один процесс получил некие результаты (но полную обработку ещё не закончил), он уже должен передать их следующему процессу. При этом первый процесс не может прервать транзакцию, потому что работа ещё не завершена и захваты с обрабатываемых обьектов снимать нельзя - даже временно. Если вся работа первого процесса представляет собой длинную транзакцию, никакого конвейера у нас не получится. Поэтому и предлагается, чтобы транзакции были короткими, но при этом разрешается устанавливать захваты на уровне контекста. Тогда любая длинная транзакция можно разбить на набор коротких - при условии, что захваты на обрабатываемые обьекты установлены на уровне контекста. То, что транзакции короткие, гарантирует, что совместно работающие процессы быстро увидят изменения – в то же время захваты на уровне контекста гарантируют, что обрабатываемые данные не изменятся «под ногами», между транзакциями.

3.4 Схема БД

Схема БД аналогична схеме сетевой БД. Она включает в себя информацию о типах хранимых обьектов (CLSID - идентификатор класса обьекта) и о наличии и типах (пара IID - идентификаторов интерфейса) связей между ними. Мы предполагаем, что схема будет легко изменяемой в любой момент.

3.5 Физическая схема хранения

Каждому обьекту при занесении в БД присваивается уникальный ключ – ID, который не меняется. Этот ID представляет собой смещение в промежуточной таблице адресов, по которому прописан реальный физический адрес данных этого обьекта. За счет одного уровня косвенности мы получаем возможность как угодно перемещать данные обьекта, не меняя его ID.

Обьекты каждого типа могут иметь как фиксированную либо ограниченную, так и неограниченную длину. Параметр 'максимальный размер обьекта данного типа' должен заноситься в схему БД. Обьекты ограниченной длины будут храниться в таблицах. Для хранения обьектов неограниченной длины память на носителе должна быть организована сегментным или страничным образом.

Связи между обьектами хранятся так, как это обычно делается в сетевых БД. Для каждого типа связи: владелец связи хранит ID первого и последнего членов связи и их общее количество, член связи хранит ID владельца, следующего и предыдущего членов.

В качестве физического носителя для хранения БД предполагается использовать либо файл собственной структуры, либо compound file – COM-обьект, представляющий собой файл, организованный внутри как файловая система. В любом случае, для ядра СУБД физический носитель будет представлять собой COM-обьект – ядру не нужно знать детали организации хранения данных на внешнем носителе. В частности, возможно хранить данные распределенно.

3.6 Высокоуровневый язык запроса

В отличии от ообщепринятых схем организации СУБД, мы предполагаем, что высокуровневый язык запроса не входит в состав ядра, а является внешней опциональной утилитой, использующей для своей работы интерфейсы ядра и обьектов. В частности, для получения информации о доступных языку запроса свойствах и методах обьектов можно использовать стандартный интерфейс IDispatch (его же можно использовать для связи СООБД с интерпретируемыми языками навроде Visual Basic).

Литература

1. Комитет по развитию функциональных возможностей СУБД. Page: 86 Системы баз данных третьего поколения: Манифест/Пер. с англ. СУБД, 1995 № 2.
2. М.Аткинсон, Ф.Бансилон, Д.ДеВитт, К.Диттрих, Д.Майер, С.Здоник. Page: 86 Манифест систем объектно-ориентированных баз данных/Пер. с англ. СУБД, 1995 № 4.
3. Х.Дарвин, К.Дэйт. Третий манифест/Пер. с англ. СУБД, 1996 № 1.
4. Frank Manola. An Evaluation of Object-Oriented DBMS Developments 1994 Edition. GTE Laboratories technical report TR-0263-08-94-165, 1994.
5. Kraig Brockschmidt. Inside OLE 2. Microsoft Press.
6. Дейл Роджерсон. Основы COM/Пер. С англ. - Москва, издательский отдел "Русская редакция" ТОО "Channel Trading Ltd.", 1997.
7. Bancilhon F., Ferran G. ODMG-93: the object database standard. Data Engineering Bulletin, December 1994, v. 17., N 4.
8. Л.А.Калиниченко. Стандарт систем управления объектными базами данных ODMG-93: краткий обзор и оценка состояния. СУБД, 1996 №1.
9. Дейт К. Введение в системы баз данных. Москва, Наука, 1980.
10. C.Papadimitriou. The theory of database concurrency control. Computer Science Press, 1986.
11. J.Gray, A.Reuter. Transaction processing: concepts and techniques. Corrected second printing, Morgan Kaufmann, 1993.

Маркин М. И.

Об одном методе повышения эффективности обучения нейронной сети прямого распространения

Обучение нейронной сети рассматривается как задача оптимизации многоэкстремальной функции многих переменных. Данная работа посвящена задаче выбора начальных значений весов нейронной сети при ее обучении с помощью методов локальной оптимизации. Предлагаемый алгоритм позволяет выбирать "хорошие" начальные приближения, что значительно сокращает время работы алгоритма локальной оптимизации и позволяет отсекать начальные приближения, находящиеся в зоне притяжения неудовлетворительных локальных минимумов. Результаты экспериментов подтверждают эффективность предлагаемого алгоритма.

Введение

Рассмотрим множество нейронных сетей (НС) прямого распространения с одним внутренним слоем. Пусть функция активации внутреннего слоя - произвольная непрерывная дифференцируемая функция, имеющая обратную; функция активации выходного слоя - линейна. Этот класс НС достаточно широк; он, в частности, включает в себя однослойные персептроны с сигмоидальной функцией активации. Далее будет рассматриваться задача аппроксимации таблично заданных функций с помощью НС из этого класса. Для решения этой задачи требуется обучить НС, то есть решить задачу минимизации нелинейной многоэкстремальной функции многих переменных, причем (как показано в работе [1]) количество локальных экстремумов экспоненциально зависит от размерности задачи и количества точек обучающей выборки. Вопрос об отыскании глобального минимума функции ошибки аппроксимации в общем случае пока остается открытым, поэтому для отыскания удовлетворительного локального минимума целесообразно применять различные алгоритмы локальной минимизации (в том числе метод обратного распространения ошибки и его вариации ([2],[5]), метод Левенберга-Марквардта [3],[4], и т.д.). Их общим недостатком является зависимость результата от выбора точки начального приближения. Для отыскания удовлетворительного локального минимума необходимо провести несколько прогонов алгоритма локальной минимизации, используя каждый раз новое начальное приближение, и затем выбрать из нескольких полученных локальных минимумов наилучший (метод мультистарта). Обычно в качестве очередного начального приближения берется случайная точка из области определения функции.

Метод случайного выбора точки начального приближения обладает тем недостатком, что он никак не учитывает структуру целевой функции. Поэтому на практике очень часто выбранная случайная точка лежит либо в области притяжения неудовлетворительного локального минимума, либо в области уже достигнутого на одном из предыдущих шагов локального минимума. В обоих случаях применение алгоритма локальной минимизации не может привести к отысканию лучшего (по сравнению с уже найденными) решения задачи.

В данной работе рассматривается метод выбора точки начального приближения, обеспечивающего по возможности "хорошее" значение целевой функции в выбранной точке. Требование суб-оптимальности начального приближения сокращает необходимое количество шагов алгоритма локальной минимизации и отсекает начальные приближения, лежащие в области притяжения неудовлетворительных локального минимумов. Возможность построения такого метода обеспечивается использованием информации о структуре целевой функции в данной конкретной задаче.

Используемые обозначения и термины

- Множество пар SET = {(x,y)}, где $x \in \Re^n$ - вектор-аргумент, y - скалярзначение, будем именовать выборкой. Размерность векторов x будем называть размерностью выборки. Мощность выборки SET определяется как количество пар (x,y). (обозначается |SET|=P). Целевыми значениями выборки SET будем называть y-компоненты пар (x,y).
- Подмножество SUBSET множества пар SET будем именовать подвыборкой и обозначать SUBSET $\subset$ SET.
- Под нейронной сетью прямого распространения с одним внутренним слоем и скалярным выходом (далее просто НС) будем понимать параметрическую функцию NET: $\Re^n \to \Re$, осуществляющую следующее преобразование:

$$NET(x,v,w) = \sum_{i}^{H} v_i f_i\left(\sum_{j=1}^{n} x^{(j)} w_i^{(j)} + w_i^{(n+1)}\right) + v_{H+1} \qquad (1)$$

где n - размерность вектора аргументов x; H - количество нейронов внутреннего слоя НС; $x \in \Re^n, x = \{x^{(1)}, x^{(2)}, \ldots, x^{(n)}\}$ - входной вектор; $v = (v^{(1)}, v^{(2)}, \ldots, v^{(H)}, v^{(H+1)}), v \in \Re^{H+1}$ и $w = \{w_i^{(j)}\}, w \in \Re^{(n+1)\times H}$ - настраиваемые параметры функции NET; $f_i(z)$ - функции активации нейронов внутреннего слоя. Далее будем предполагать, что все функции $f_i(z)$ непрерывны и дифференцируемы, а также существуют обратные функции $f_i^{-1}(z)$. Например, этим условиям удовлетворяет сигмоида $f(z) = \frac{1}{1+e^{-z}}$.

- Среднеквадратичной ошибкой аппроксимации НС NET на выборке SET будем называть функцию MSE:$\Re^{(n+1)(H+1)} \to \Re$, определенную на множестве допустимых значений параметров персептрона v,w как

$$MSE_{SET}(v,w) = \sum_{(x_p,y_p)\in SET} (NET(x_p,v,w) - y_p)^2 \qquad (2)$$

- Обучением персептрона NET на выборке SET будем называть процесс поиска такого набора (v,w) параметров персептрона, который минимизирует ошибку MSE - то есть решение оптимизационной задачи

$$\min_{v,w} MSE_{SET}(v,w) \quad . \qquad (3)$$

Свойства задачи обучения НС

Для начала рассмотрим свойства задачи обучения НС (3). Множество аргументов функции MSE (2) можно разделить на два множества - внешних весов v и внутренних весов w.

Свойство 1: При фиксированных значениях внутренних весов w возможно указать оптимальные значения внешних весов v.

Доказательство: Действительно, пусть внутренние веса $w = \{w_i^{(j)}\}, w \in \Re^{(n+1)\times H}$ имеют фиксированные значения $\hat{w} = \{\hat{w}_i^{(j)}\}, w \in \Re^{(n+1)\times H}$, тогда целевая функция MSE приобретает вид

$$MSE_{SET}(v) = \sum_{(x_p, y_p)\in SET} (NET(x_p, v, \hat{w}) - y_p)^2 = \sum_{(x_p, y_p)\in SET} (\sum_{i}^{H} v_i f_{ip} + v_{H+1} - y_p)^2, \text{ где}$$

$$f_{ip} = f_i(\sum_{j=1}^{n} x_p^{(j)} \hat{w}_i^{(j)} + \hat{w}_i^{(n+1)}) = const$$

Таким образом, при фиксированных значениях внутренних весов w задача (3) сводится к хорошо изученной линейной задаче о наименьших квадратах $\min_x \|Ax - b\|$. В нашем случае

$$A = \begin{vmatrix} f_{11} & f_{21} & \cdots & f_{H1} & 1 \\ f_{12} & f_{22} & \cdots & f_{H2} & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ f_{1P} & f_{2P} & \cdots & f_{HP} & 1 \end{vmatrix} \in \Re^{P\times(H+1)}, \; b = \begin{vmatrix} y_1 \\ y_2 \\ \cdots \\ y_P \end{vmatrix} \in \Re^{P}, x = \begin{vmatrix} v_1 \\ v_2 \\ \cdots \\ v_{H+1} \end{vmatrix} \in \Re^{H+1},$$

где H - число нейронов внутреннего слоя, P = |SET| - количество точек выборки.

Как хорошо известно, целевая функция этой задачи является выпуклой (следовательно одноэкстремальной) и ограниченной снизу (нулем). Ее решение может получено как с помощью обратной матрицы Мура-Пенроуза ($x = \overline{A}b = (A^T A)^{-1} A^T b$), так и с помощью любых других методов выпуклой минимизации. •

Рассмотрим аналогичную задачу для случая, когда фиксированы значения внешних весов v, и нам необходимо определить оптимальные значения внутренних весов w. В некоторых случаях это также оказывается возможным.

Свойство 2: Пусть значения внешних весов v фиксированы: $v = \hat{v} = const$ и пусть количество нейронов внутреннего слоя H = 1. Имеет место следующая альтернатива:

- либо существует такой набор внутренних весов w^*, для которого $MSE(\hat{v}, w^*) = 0$, и этот набор весов можно получить как решение системы линейных уравнений;
- либо для любых значений внутренних весов w справедливо $MSE(\hat{v}, w) > 0$.

Доказательство: Рассмотрим целевую функцию $MSE(\hat{v},w)$ при H = 1. Очевидно, что равенство $MSE(\hat{v},w)=0$ может достигаться в том и только в том случае, если выполняются следующие равенства:

$$\hat{v}_1 f(\sum_{j=1}^{N} x_p^{(j)} w^{(j)} + w^{(N+1)}) + \hat{v}_2 = y_p \quad (\text{p} = 1,2..,\text{P})$$

(4)

(здесь N - размерность выборки SET (размерность векторов x), P - мощность выборки SET (количество точек в выборке)).

Рассмотрим сначала вырожденный случай $\hat{v}_1 = 0$. Равенства (4) выполнимы тогда и только тогда, когда $\hat{v}_2 = y_p$ для p = 1,2..,P. В этом случае значение функции MSE не зависит от внутренних весов w:

$$MSE_{SET}(\hat{v},w) = \sum_{(x_p,y_p)\in SET} (\hat{v}_2 - y_p)^2 = const\,.$$

Поэтому MSE = 0, если $\hat{v}_2 = y_p$ для всех p = 1,2..,P. Рассмотрим теперь случай $\hat{v}_1 \neq 0$. Равенства (4) выполнимы тогда и только тогда, когда

$$f(\sum_{j=1}^{N} x_p^{(j)} w^{(j)} + w^{(N+1)}) = \frac{y_p - \hat{v}_2}{\hat{v}_1} \qquad (\text{p} = 1,2..,\text{P})$$

(5)

Обозначим множество значений функции $f(z)$ активации нейрона внутреннего слоя как $[f_{min}, f_{max}]$ (f_{min} может быть равен -∞, f_{max} может быть равен +∞). Тогда на сегменте $[f_{min}, f_{max}]$ определена обратная функция $f^{-1}(z)$. Если для всех p = 1,2..,P удовлетворяются условия

$$\frac{y_p - \hat{v}_2}{\hat{v}_1} \in [f_{min}, f_{max}],$$

(6)

то равенства (5) можно переписать как

$$\sum_{j=1}^{N} x_p^{(j)} w^{(j)} + w^{(N+1)} = f^{-1}(\frac{y_p - \hat{v}_2}{\hat{v}_1}) \qquad (\text{для p} = 1,2..,\text{P})$$

(7)

В противном случае система уравнений (5) неразрешима и для любых w справедливо $MSE(\hat{v},w) > 0$ (альтернатива 2).

Система уравнений (7) является системой линейной уравнений с N + 1 неизвестными и P уравнениями, и ее разрешимость (или неразрешимость) легко установить. Таким образом, если w^* - решение СЛАУ (7), то $MSE(\hat{v},w^*) = 0$ (альтернатива 1); если СЛАУ (6) неразрешима, то для любых w $MSE(\hat{v},w) > 0$ (альтернатива 2).•

Из доказательства свойства 2 видно, что для любых значений внешних весов $\hat{v}$, удовлетворяющих условиям (6), возможно указать такие значения внутренних весов w, что $MSE(\hat{v},w) = 0$, если P ≤ N + 1 и вектора $\bar{x}_p = (x_p^{(1)} x_p^{(2)} \ldots x_p^{(N)} 1)$ линейно независимы.

Алгоритм выбора начального приближения для случая 1 нейрона во внутреннем слое

Рассматривая задачу (3), представляется перспективным построение итеративного алгоритма обучения НС в следующей форме:

$$
\begin{aligned}
&w^* := w_{start}, v^* := v_{start} \\
&\text{ЦИКЛ} \\
&w' := w^*, v' := v^* \\
&w^* := \arg\min_w MSE_{SET}(v^*, w) \\
&v^* := \arg\min_v MSE_{SET}(v, w^*) \\
&\text{ЕСЛИ } w' = w^* \,\&\, v' = v^*, \text{ ТО ОСТАНОВКА.} \\
&\text{КОНЕЦ ЦИКЛА}
\end{aligned}
\qquad (8)
$$

Подобный алгоритм, будучи построен, обладал бы такими свойствами, как:

- невозрастание значения целевой функции на каждом шаге;
- критерием стационарной точки здесь служит неизменение значений v^*, w^* на протяжении одной итерации, то есть одновременное выполнение условий $w^* = \arg\min_w MSE_{SET}(v^*, w)$ и $v^* = \arg\min_v MSE_{SET}(v, w^*)$. Глобальный минимум функции MSE удовлетворяет этим условиям. Кроме того, предполагается, что множество таких стационарных точек существенно уже множества локальных минимумов функции MSE.

Подзадача $v^* := \arg\min_v MSE_{SET}(v, w^*)$ может быть решена с помощью метода, описанного в доказательстве свойства 1 в предыдущем разделе. Основной трудностью при построении подобного алгоритма является отсутствие эффективного метода решения подзадачи на оптимизацию внутренних весов $w^* := \arg\min_w MSE_{SET}(v^*, w)$. В связи с этим предлагается следующая гипотеза:

Пусть SUBSET - некоторая подвыборка выборки SET, тогда при фиксированных значениях внешних весов v = v* решение "малой" (на выборке SUBSET) задачи $\tilde{w}^* = \arg\min_w MSE_{SUBSET}(v^*, w)$ можно рассматривать как приближенное решение аналогичной "большой" (на выборке SET) задачи

$w^* = \arg\min_w MSE_{SET}(v^*, w)$.

Рассмотренное в предыдущем разделе свойство 2 задачи обучения НС дает нам необходимый аппарат для решения "малых" задач в случае, когда количество точек в подвыборке SUBSET не более чем на единицу превосходит размерность входных векторов ($P \le N + 1$) и когда количество нейронов внутреннего слоя H = 1. Предлагается использовать вариант алгоритма (8), в котором вместо решения "большой" задачи $w^* = \arg\min_w MSE_{SET}(v^*, w)$ решается "малая" задача $\tilde{w}^* = \arg\min_w MSE_{SUBSET}(v^*, w)$. Согласно сформулированной гипотезе, $w^* \approx \tilde{w}^*$, и, таким образом, можно рассчитывать на генерацию этим алгоритмом точек, близких

к стационарным (в смысле алгоритма (8)) точкам функции MSE. Полученные таким образом точки целесообразно использовать в качестве стартовых точек для алгоритма локальной минимизации.

Теперь возможно сформулировать предлагаемый алгоритм сначала для случая H = 1 (один нейрон во внутреннем слое):

Алгоритм выбора начального приближения для задачи обучения НС (H = 1):
ПРОЦЕДУРА NN_INIT_SINGLE_NEURON
АРГУМЕНТЫ
SET : выборка
РЕЗУЛЬТАТ
v_1, v_2 : внешние веса; $v_i \in \Re$
w : внутренние веса; $w \in \Re^{N+1}$
НАЧАЛО

1. SUBSET := Выбрать_Подмножество(SET).

Комментарий: Метод выбора подмножества SUBSET выборки SET существенно влияет на результат работы алгоритма. Методы выбора SUBSET будут обсуждаться ниже. На данном этапе необходимо потребовать лишь линейную независимость векторов $\bar{x}_p = (x_p^{(1)} x_p^{(2)} \ldots x_p^{(N)} 1)$ подвыборки SUBSET и выполнения неравенства |SUBSET| ≤ N + 1, что гарантирует (согласно замечаниям к свойству 2) разрешимость задачи $w^* = \arg\min_w MSE_{SUBSET}(v^*, w)$

2. v^* := произвольные значения, удовлетворяющие условиям (6).

Комментарий: такие значения легко могут быть найдены. Действительно, если

множество значений функции активации $f(z)$ неограничено, то достаточно положить v1 := 1, v2 := 0. Если оно ограничено конечными значениями [fmin,fmax], то допустимыми являются значения v1=(max(y)-min(y))/(fmax-fmin), v2=min(y), где функции max(y) и min(y) обозначают соответственно максимум и минимум по y в выборке SUBSET = { (x , y) }. Для случая, когда только одно из значений fmin, fmax конечно, можно получить аналогичные формулы.

3. $w^* = \arg\min_w MSE_{SUBSET}(v^*, w)$

Комментарий: метод решения этой задачи путем сведения к системе линейных уравнений приведен в доказательстве свойства 2 задачи обучения НС. Выбор подмножества SUBSET и внешних весов v на шагах 1,2 производятся таким образом, чтобы удовлетворять необходимым условиям разрешимости этой задачи.

4. $v^* = \arg\min_v MSE_{SET}(v, w^*)$

Комментарий: метод решения этой задачи путем сведения к линейной задаче о наименьших квадратах приведен в доказательстве свойства 1 задачи обучения НС. Однако необходимо, чтобы полученный в качестве решения набор внешних весов v* удовлетворял условиям (6) для функции активации с ограниченной обла-

стью изменения. Если эти условия не выполняются (что, как показывают эксперименты, является чрезвычайно редким случаем), алгоритм завершает свою работу .

5. Если значение $MSE_{SET}(v,w)$ не улучшилось за последнюю итерацию (пункты 3-5), то остановиться и результатом считать пару (v^*,w^*); иначе перейти к шагу 3.

Комментарий: критерием останова алгоритма также может быть исполнение некоторого максимально допустимого количества итераций (одна итерация - шаги 3,4). На практике исполнение более 5-10 итераций не приводит к заметным улучшениям работы алгоритма.

КОНЕЦ NN_INIT_SINGLE_NEURON

Очевидно, что метод выбора подмножества SUBSET существенно влияет на результат работы алгоритма. Недетерминированность этого шага позволяет использовать предлагаемый алгоритм для генерации множества различных точек начального приближения при обучении сети методом мультистарта.

Алгоритм выбора начального приближения для случая N нейронов во внутреннем слое

Построение алгоритма выбора точки начального приближения для алгоритма локальной минимизации в случае произвольного количества нейронов H во внутреннем слое НС будет проводиться на базе уже разработанного алгоритма для случая H = 1. Рассмотрим задачу выбора начального приближения для НС с H > 1 нейронами во внутреннем слое для обучения на выборке $SET = \{(x_p, y_p)\}, p = 1,2,...,P$. Предлагается пошаговая стратегия построения начального приближения, при которой на каждом шаге определяются значения весов очередного нейрона. Веса каждого из нейронов выбираются как начальное приближение для НС с 1 нейроном при обучении на выборке, целевые значения которой представляют собой разность целевых значений y_p исходной выборки и значений, порожденных сетью из уже инициализированных нейронов. Рассмотрим один шаг такого процесса.

Предположим, что уже определены значения весов для K нейронов ($1 \le K < H$). Обозначим выходной сигнал сети с K такими нейронами во внутреннем слое как $NET_K(x)$. Сформируем выборку $SET_K = \{(x_p, y_p - NET_K(x_p))\}, p = 1,2,...,P$. Целевые значения $y_p - NET_K(x_p)$ выборки SET_K задают отклонение сигнала, генерируемого уже построенной сетью с K нейронами, от целевых значений исходной выборки SET. Веса (K+1)-го нейрона целесообразно выбирать таким образом, чтобы минимизировать это отклонение.

Используя описанный выше алгоритм NN_INIT_SINGLE_NEURON, выберем начальное приближение для НС с 1 нейроном для обучения на выборке SET_K. Обозначим выходной сигнал этой НС как $NET_1(x)$. Значения весов единственного ней-

рона в сети NET_1 и задают веса (K+1)-го нейрона строящейся сети. Формально, на данном этапе необходимо объединить две НС с выходными сигналами $NET_1(x)$ и $NET_K(x)$ в единую НС $NET_{K+1}(x)$ с (K+1) нейроном во внутреннем слое так, чтобы выполнялось $NET_{K+1}(x) = NET_1(x) + NET_K(x)$. Строение функций $NET(x)$ легко позволяет это сделать.

Продолжая такой процесс, за H шагов возможно построить НС с H нейронами во внутреннем слое. Значения ее весов задают начальное приближение при обучении сети методами локальной минимизации. Далее будем обозначать этот алгоритм как NN_INIT.

Результаты экспериментов

С целью более подробного исследования свойств построенного алгоритма было проведено его комплексное тестирование на множестве выборок из аналитических параметрических функций 2-х аргументов.

Набор тестов был построен таким образом, чтобы было возможно сопоставить свойства разработанного метода NN_INIT по сравнению с методом случайного выбора точки начального приближения (обозначим этот метод как RANDOM_INIT). Сравнение проводилось по следующим категориям:

- Близость полученных начальных приближений к точкам оптимума;
- Качество решений, получаемых при инициализации весов обоими методами;
- Вычислительная эффективность.

В процессе тестирования были использованы выборки из следующих параметрических функций:

- Stone: z = 0.5 + sin(k1*pi*sqrt(x^2 + y^2))/2;
- Finger: z = 0.5 + sin(k1*2*atan(x/(y+0.00001)))/2;
- StoneFinger: z = 0.5 + sin(k1*2*atan(x/(y+0.00001)))/4 + sin(k2*pi*sqrt(x*x + y*y))/4.

В модели НС в качестве функции активации использовалась сигмоида $f(z) = \frac{1}{1+e^{-z}}$ (область значений - от 0 до 1), количество нейронов внутреннего слоя варьировалось от 1 до 10. В качестве алгоритма локальной минимизации использовался метод Левенберга-Марквардта ([3],[4]).

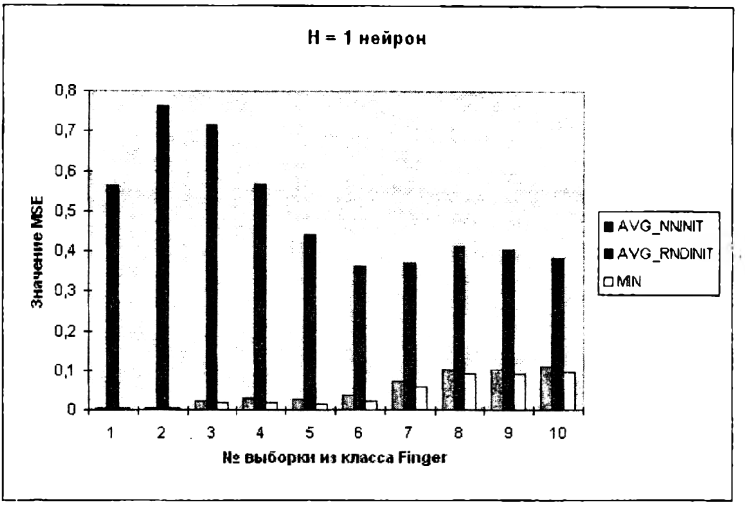
Процесс тестирования заключался в следующем: для каждой из тестовых выборок и каждой из архитектур НС было сгенерировано по 40 точек начального приближения с применением обоих методов NN_INIT и RANDOM_INIT. Далее, к каждому из 80 полученных начальных приближений применялся алгоритм локальной минимизации. Всего использовалось 30 выборок (по 10 из каждого класса Finger, Stone, StoneFinger) и 3 архитектуры НС (с 1, 3 и 10 нейронами во внутреннем слое).

Рассмотрим вопрос о том, насколько "хорошие" начальные приближения генерируются методом NN_INIT. На нижеприведенной гистограмме собраны следующие значения:

• среднее значение целевой функции MSE в точках начального приближения, полученных с помощью метода NN_INIT - обозначены как AVG_NNINIT;

• среднее значение целевой функции MSE в точках начального приближения, полученных с помощью метода RANDOM_INIT - обозначены как AVG_RNDINIT;

• наилучшее из достигнутых значений целевой функции MSE обозначено как MIN.



Для выборок из других параметрических функций (и других количеств нейронов) были получены аналогичные результаты.

Исходя из структуры методов локальной минимизации, было бы логичным предположить, что при использовании метода выбора начального приближения, генерирующего более-менее близкие к оптимуму точки, следует ожидать сокращения среднего количества необходимых итераций метода локальной минимизации. Полученные результаты подтверждают справедливость этого предположения: при использовании метода инициализации NN_INIT требуется в среднем меньшее, чем для RND_INIT, количество итераций метода локальной оптимизации :

NUM	NN_INIT_FEVAL	RND_INIT_FEVAL
1	10002	10002
2	44	82
3	104	116
4	57	117
5	87	211
6	84	122
7	187	267
8	1277	8621
9	443	942
10	1259	1141

Здесь:

- NUM - Номер выборки из класса StoneFinger
- NN_INIT_FEVAL - Среднее количество циклов алгоритма локальной минимизации при выборе начального приближения с помощью метода NN_INIT
- RND_INIT_FEVAL - Среднее количество циклов алгоритма локальной минимизации при выборе начального приближения с помощью метода RND_INIT

Данные приведены для НС 1 нейроном во внутреннем слое. Для других количеств нейронов результаты аналогичны.

Также был исследован вопрос о том, могут ли более "хорошие" начальные приближения улучшить качество достигаемых локальных минимумов при многократном применение метода локальной минимизации с разными начальными приближениями (метод мультистарта). Результаты проведенных экспериментов показывают, что в ряде случаев (~ 30 % от общего числа задач) при использовании метода NN_INIT достигаются лучшие результаты, чем при использовании метода RND_INIT; в остальных случаях наилучший найденный локальный минимум при использовании обоих методов инициализации совпадает с точностью до 0,1%.

Вычислительная сложность метода NN_INIT относительно мала по сравнению со сложностью локальной минимизации. В проведенных экспериментах использовалось ограничение на максимальное количество итераций решения задач на внешние и внутренние веса (см. описание метода NN_INIT_SINGLE_NEURON, шаги 3-5) MAX_IT = 5. Для инициализации весов каждого из нейронов требуется решение не более MAX_IT систем линейных N уравнений с N неизвестными (N - размерность входных векторов выборки) и решение не более MAX_IT линейных задач о наименьших квадратах с 2 неизвестными. Время, затрачиваемое на выбор точки начального приближения по методу NN_INIT составляет приблизительно 0,3 - 0,5 % от времени, затрачиваемого алгоримом локальной минимизации (данные для НС с 10 нейронами во внутреннем слое). С другой стороны, необходимое количество шагов (и, соответственно, время работы) алгоритма локальной минимизации сокращается в ряде случаев до 2-3 раз. Таким образом, применение метода NN_INIT для выбора точки начального приближения является оправданным с точки зрения вычислительной эффективности.

Заключение

В данной работе построен алгоритм выбора точки начального приближения для решения задачи обучения НС методами локальной минимизации. Построенный алгоритм использует специфику и структуру задачи обучения НС, что позволяет выбирать достаточно "хорошие" начальные приближения. Обобщая результаты экспериментов, можно отметить следующие свойства построенного метода:

- сокращение количества необходимых итераций метода локальной минимизации (как следствие того, что метод генерирует начальные приближения, достаточно близкие к точкам оптимума целевой функции);
- общее улучшение качества получаемых решений в случае использования метода мультистарта для решения задач обучения НС (метод с высокой вероятностью отсекает точки начального приближения, лежащие в области притяжения неудовлетворительных локальных минимумов);
- применимость к широкому классу НС (поддерживаются любые НС с 1 внутренним слоем и произвольными типами нейронов во внутреннем слое; единственным ограничением на функцию активации нейронов во внутреннем слое является существование обратной функции);
- вычислительная эффективность (время работы метода пренебрежимо мало по сравнению с временем работы собственно алгоритма локальной минимизации).

Проведенные эксперименты подтверждают целесообразность применения построенного метода для инициализации весов НС при ее обучении.

Литература

1. Auer P., Herbster M. and Warmuth M.K. (1996). "Exponentially many local minima for single neurons", *Technical Report UCSC-CRL-96-1, Univ.of Calif.Computer Research Lab, Santa Cruz, CA*
2. Fahlman, S.E. (1989), "Faster-Learning Variations on Back-Propagation: An Empirical Study", *in Touretzky, D., Hinton, G, and Sejnowski, T., eds., Proceedings of the 1988 Connectionist Models Summer School, Morgan Kaufmann, 38-51.*
3. Levenberg, K. (1944) "A method for the solution of certain problems in least squares," *Quart. Appl. Math., 2, 164-168.*
4. Marquardt, D. (1963) "An algorithm for least-squares estimation of nonlinear parameters," *SIAM J. Appl. Math., 11, 431-441.*
5. Riedmiller, M. and Braun, H. (1993), "A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm", *Proceedings of the IEEE International Conference on Neural Networks 1993, San Francisco: IEEE*

Афанасьев М.К., Дмитриев П.А.

Новые подходы к исследованию поведения генетических алгоритмов с использованием NS Galaxy

В статье описываются возможности программного продукта NSGalaxy, который является универсальным средством для построения генетических алгоритмов и исследования особенностей их работы. По сравнению с системами подобного рода, за счёт внедрения нескольких схем кодирования, был заметно расширен класс решаемых задач. К достоинствам NSGalaxy также относятся: автоматическое распараллеливание работы алгоритма, удобный пользовательский интерфейс, поддержка межсетевого взаимодействия и т.д.

1. Введение.

Целью статьи является описание конструктора генетических алгоритмов, разработанного на факультете ВмиК МГУ при поддержке гранта РФФИ № 99-07-90229. Данный программный продукт отличается от других систем подобного рода рядом новых возможностей и носит название NSGalaxy – New Software Genetic Algorithm Axle.

В настоящее время, в связи с бурным развитием вычислительной техники, наблюдается тенденция к отходу от формальной постановки задачи к менее строгой и применению таких универсальных методов решения как генетические алгоритмы, для которых надо предоставить лишь минимум информации о задаче. В связи с этим возникает необходимость разработки программных средств, с помощью которых можно было бы автоматически строить генетические алгоритмы. Именно к такого рода разработкам и относится NSGalaxy.

Но, несмотря на высокую производительность современных компьютеров, такой универсальный метод, как генетические алгоритмы может работать достаточно долго. Поэтому в NSGalaxy заранее вкладывается возможность параллельных и распределённых вычислений, что позволяет существенно повысить скорость нахождения оптимального решения.

Также NSGalaxy представляет собой удобный инструмент для экспериментального исследования новых свойств и особенностей поведения генетического алгоритма. За счёт внедрения нескольких схем кодирования существенно расширен класс решаемых задач, а графическое представление решения позволяет в динамике наблюдать за работой алгоритма.

Многое из того, что мы видим и наблюдаем в природе, можно объяснить единой теорией - теорией эволюции. Основные механизмы эволюционного развития - наследственность, изменчивость и естественный отбор. Во многих случаях они не могут отразить некоторые специфические особенности развития, но, тем не менее, эти механизмы объясняют широкий спектр явлений, наблюдаемых в природе. Возможность того, что вычислительная система, наделенная простыми механизмами наследственности, изменчивости и отбора, могла бы функционировать в соответствии с законами эволюции в природных системах, издавна привлекала учёных. Это привело к появлению ряда вычислительных методов, построенных на принципах эволюционной теории. Это эволюционные алгоритмы, такие как Эволюционное

Программирование (Evolutionary Programming), Генетические Алгоритмы (Genetic Algorithms), Эволюционные Стратегии(Evolution Strategies) и др.

Генетические алгоритмы (далее ГА) применяются во многих научных и технических отраслях. Они используются при создании различных вычислительных структур, например, автоматов или сетей сортировки, применяются при управлении роботами и проектировании нейронных сетей. ГА также используются при моделировании развития в биологических (экология, иммунология и популяционная генетика), социальных (такие как экономика и политические системы) и когнитивных системах. Например, при решении проблемы эффективного менеджмента организации Паккард (Packard) использовал генетический алгоритм для определения зависимости между проблемами, решением которых занимается офис и эффективностью данного офиса [1]. Им были описаны 19 различных проблем, ассоциированных с различными офисами организации, и определена мера эффективности (производительности) каждого офиса. ГА использовался для поиска комбинаций проблем, которые сильно влияли на производительность офисов.

Тем не менее, наиболее популярное приложение ГА - оптимизация многопараметрических функций [6]. Многие реальные задачи могут быть сформулированы как поиск оптимального значения, где значение – сложная функция, зависящая от некоторых входных параметров. В некоторых случаях необходимо найти те значения параметров, при которых достигается точное наилучшее значение функции. В других случаях точный оптимум не требуется - решением может считаться любое значение, которое лучше некоторой заданной величины. В этом случае ГА - часто наиболее приемлемый метод для поиска "хороших" значений.

Приведём несколько примеров задач, которые могут быть решены с помощью ГА: поиск экстремума функции многих переменных, простая и множественная проблемы упаковки рюкзака, проблема размещения.

Однако нередки случаи, когда ГА не работает так эффективно, как ожидалось. Предположим, есть реальная задача, сопряженная с поиском оптимального решения. Как узнать, является ли ГА хорошим методом для ее решения? В настоящее время не существует строгого ответа на этот вопрос. Однако многие исследователи разделяют предположение, что если пространство поиска, которое предстоит исследовать, большое, и предполагается, что оно не гладкое и унимодальное (т.е. содержит один гладкий экстремум), или если функция качества с шумами, или если задача не требует строго нахождения глобального оптимума - т.е. если нужно просто достаточно быстро найти приемлемое "хорошее" решения (что довольно часто имеет место в реальных задачах) - ГА будет иметь хорошие шансы стать эффективной процедурой поиска, конкурируя и превосходя другие методы, которые не используют знания о пространстве поиска.

Если пространство поиска небольшое, то решение может быть найдено методом полного перебора, и можно быть уверенным, что найдено наилучшее возможное решение, тогда как ГА с большей вероятностью может сойтись к локальному оптимуму, а не к глобально лучшему решению. Если пространство гладкое и унимодальное, любой градиентный алгоритм, например метод градиентного спуска, будет более эффективен, чем ГА. Если о пространстве поиска есть некоторая дополнительная информация (как, например, пространство для хорошо известной задачи о коммивояжере), методы поиска,

использующие эвристики, определяемые пространством, часто превосходят любой универсальный метод, каким является ГА.

2. Этапы построения генетического алгоритма.

Итак, цель применения ГА состоит в том, чтобы найти по возможности лучшее решение или решения задачи. Чтобы реализовать алгоритм, нужно сначала выбрать подходящую структуру для представления возможных решений (**схему кодирования**) (Coding Scheme). Каждый конкретный экземпляр этой структуры будем называть **хромосомой**(Chromosome). В простейшем случае каждая хромосома представляет собой просто битовую строку. Часто ГА используют более сложную схему кодирования, которая может оперировать не с хромосомами, а с особями, состоящими из нескольких хромосом [2], вместо же двоичного может использоваться алфавит из большего числа символов [3-5]. Однако бинарный случай самый простой и наиболее распространённый. По аналогии с биологическими системами каждая позиция в хромосоме называется **геном**(Gen).

Чтобы оптимизировать структуру, используя ГА, нужно задать некоторую меру качества для каждой хромосомы. Для этой цели используется **функция качества** (приспособленности) (Fitness Function). Например, для задач отыскания экстремума функции многих переменных или многокритериальной минимизации в качестве функции качества выступает сама функция или функционал. В качестве хромосомы – вектор $\{x_1,...,x_N\}$, в качестве отдельного гена – компонент вектора. В качестве популяции выступает некоторое конечное подмножество векторов.

Задание схемы кодирования и функции качества оказывают решающее влияние на скорость сходимости генетического алгоритма

Схема простейшего ГА в нотации языка Си выглядит следующим образом:

```
Инициалиализация(популяция);
If (Решение найдено) return решение;T = 0;
While (T < Тост) do { T = T+1;
    Селекция(популяция);
    Рекомбинация(популяция);
    Мутация(популяция);
    If (Решение найдено) return решение;}
```

Система NSGalaxy нацелена на инструментальную поддержку этих этапов разработки ГА.

Рассмотрим подробнее эту схему. Сначала происходит инициализация начальной популяция хромосом. Работа ГА представляет собой итерационный процесс, который продолжается до тех пор, пока не будет получено заданное число поколений или не выполнится какой-либо иной критерий остановки. В каждом поколении реализуется селекция пропорционально приспособленности (Selection), скрещивание (кроссовер) (Crossover) и мутация(Mutation) (рис.1).

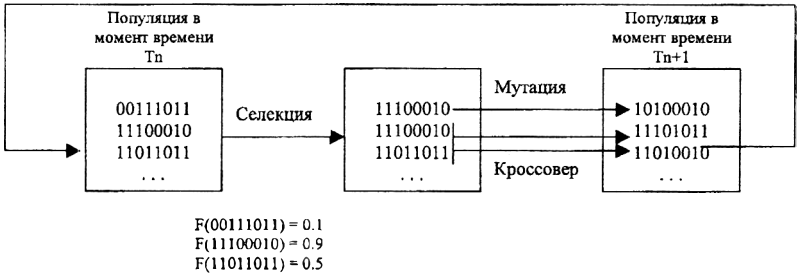


Рис1. Работа простейшего генетического алгоритма.

В первом прямоугольнике показана популяция из трёх особей в момент времени T_n. Каждой особи сопоставляется значение функции качества, и в соответствии с этим значением происходит селекция. На следующую фазу (второй прямоугольник) выбрано две копии второй особи (11100010) и одна копия третьей особи (11011011). Затем первая особь подвергается мутации (второй бит изменяется на противоположный), и из двух оставшихся применением одноточечного кроссовера

Сначала пропорциональный отбор назначает каждой структуре вероятность $P_s(i)$ равную отношению ее приспособленности к суммарной приспособленности популяции:

$P_s(i) = f(i)/\Sigma f(j)$, $j = 1..N$, $f(x)$ – функция качества.

Затем, согласно величине $P_s(i)$, происходит отбор(селекция) P особей для дальнейшей генетической обработки. Простейший пропорциональный отбор - рулетка - отбирает особей с помощью P "запусков" рулетки. Колесо рулетки содержит по одному сектору для каждого члена популяции. Размер i-ого сектора пропорционален соответствующей величине $P_s(i)$. При таком отборе члены популяции с более высокой приспособленностью будут выбираться с большей вероятностью, чем особи с низкой приспособленностью.

После отбора выбранные особи подвергаются скрещиванию (иногда называемому рекомбинацией или кроссовером). P строк случайным образом разбиваются на P/2 пары. Для каждой пары применяется, например, одноточечный кроссовер.

Одноточечный кроссовер работает следующим образом. Пусть длина каждой хромосомы m. Сначала случайным образом выбирается одна из m-1 точек разрыва. (Точка разрыва - участок между соседними битами в строке.) Обе родительские хромосомы разрываются на два сегмента по этой точке. Затем соответствующие сегменты различных родителей склеиваются и получаются два генотипа потомков. Например, пусть один родитель состоит из 6 нолей, а другой - из 6 единиц, и из 5 возможных точек разрыва выбрана точка 2. Родители и их потомки показаны ниже.

Одноточечный кроссовер

Родитель 1:	000000	00~0000	-->	Потомок 1:	11~0000	110000
Родитель 2:	111111	11~1111	-->	Потомок 2:	00~1111	001111

После того, как закончится стадия скрещивания, выполняются операторы мутации. В каждой строке, которая подвергается мутации, каждый бит с вероятностью P_m изменяется на противоположный. Таким образом, мы получаем P особей-потомков.

Затем из исходной популяции ещё раз выбираются P особей. Отбор осуществляется согласно описанному выше алгоритму, но уже в соответствии с вероятностью $P_k(i) = 1-P_s(i)$ (т.е. с большей вероятностью будут отобраны хромосомы с худшей приспособленностью). Эти хромосомы удаляются из популяции, и вместо них записываются особи-потомки. Последующие поколения обрабатываются таким же образом: отбор, скрещивание, мутация и обновление популяции.

Необходимо отметить, что существует много других операторов отбора, скрещивания и мутации. Вот лишь наиболее распространённые из них. Прежде всего, турнирный отбор. Этот метод реализует n турниров, чтобы выбрать n особей. Каждый турнир построен на выборке k элементов из популяции, и выборе лучшей особи среди них. Наиболее распространен турнирный отбор с k=2.

Элитные методы отбора гарантируют, что при отборе обязательно будут выживать лучший или лучшие члены популяции. Наиболее распространена процедура обязательного сохранения только одной лучшей особи. Данная идея может быть внедрена практически в любой стандартный метод отбора.

Двухточечный кроссовер и равномерный кроссовер - вполне достойные альтернативы одноточечному оператору. В двухточечном кроссовере выбираются две точки разрыва, и родительские хромосомы обмениваются сегментом, который находится между двумя этими точками. В равномерном кроссовере каждый бит первого родителя наследуется первым потомком с заданной вероятностью; в противном случае этот бит передается второму потомку.

3. Обзор существующих конструкторов.

На сегодняшний день существует довольно много программных средств, которые обучают и помогают применить ГА. Все эти средства можно разделить на три категории:

- *Средства, созданные для объяснения, что же такое ГА, и как он работает.*
- *Библиотеки классов,* которые вы можете использовать в своих программах для решения задач с помощью ГА.
- *Инструментальные средства,* позволяющие пользователю, настраивая определённым образом параметры, описать свою задачу, и затем получить её решение с помощью ГА.

Рассмотрим подробнее каждую из этих категорий.

К *средствам, объясняющим работу ГА*, относятся такие программы, как BUGS(Better to Use Genetic Systems)(C++), GA Optimizer(Java) и много других2. В

2 Более подробную информацию об этих и других продуктах, упоминающихся в данном разделе, можно получить на сайте www.aic.nrl.navy.mil.

этих программах работа ГА рассматривается на примере какой-то одной задачи (например, задачи об упаковке рюкзака). Пользователь может изменять некоторые параметры работы алгоритма (вероятность мутации, количество особей в популяции, тип скрещивания и др.) и смотреть, как это сказывается на сходимости алгоритма к лучшему решению. С помощью таких средств вы не можете решать какие-то свои задачи – вы можете только исследовать поведение ГА в зависимости от некоторых параметров.

Библиотеки классов предоставляют программисту набор классов, реализующих основные элементы ГА: кодовую схему, различные виды скрещивания и мутации. Также, с помощью задания соответствующих параметров вы можете устанавливать вероятность мутации, количество особей для размножения и т.п. В своей программе вы реализуете 'каркас' алгоритма, вызывая в нужных местах соответствующие функции из библиотеки. Наиболее известны такие библиотеки, как GAL.ib(Genetic Algorithm Library)(C++), libga100(C), GAJIT(Genetic Algoruithm Java Implementation Toolkit)(Java). Очевидное преимущество при использовании таких библиотек состоит в том, что если вас не устраивает какая-то реализация какой-то функции из библиотеки, вы всегда можете сами написать нужную реализацию. Но использование библиотеки классов сопряжено с большими трудностями. Во-первых, вы должны владеть хорошо языком программирования, на котором написана выбранная вами библиотека. Во-вторых, прежде чем приступить к использования библиотеки в своих программах, вы потратите немало времени на изучение функциональных возможностей, соглашений, интерфейсов и т.п. данной библиотеки, после чего может оказаться, что очень многое в данной библиотеке вас не устраивает, и большую часть кода вам всё равно придётся писать самим. Кроме выше перечисленных проблем, часто существует проблема интеграции двух программных продуктов, написанных, в общем случае, для разных реализаций языка программирования.

Третья категория представляет собой *инструментальные программные средства*, которые позволяют пользователю, совершенно не знакомому с программированием, формально описать свою задачу и попытаться построить для неё ГА. К таким средствам относятся: GENOCOP(GEnetic algorithm for Numerical Optimization for COnstrained Problems), GENESIS, GAPlayground, GECO и др. Хотя эти средства не обладают такой гибкостью, как библиотеки классов, наличие большого числа генетических операторов и задаваемых параметров может приблизить их к библиотекам классов. Необходимым элементом такого средства является удобный графический пользовательский интерфейс(GUI), большое количество примеров, объясняющих применение данного средства на конкретных задачах и оптимально подобранные значения параметров по умолчанию. Таким образом, вы можете очень быстро приступить к использованию этого средства для решения своей задачи. Важными достоинствами инструментальных средств являются также графическое представление решения и возможность динамически изменять параметры работы генетического алгоритма. К сожалению, ни одно из имеющихся на сегодняшний день средств не обладает выше перечисленными свойствами в достаточной мере. Большинство из них не обеспечивает должной гибкости при задании кодовой схемы (хромосома – строка битов) и функции качества (обычно задаётся в виде тек-

стовой строки), хотя предоставляют большой набор генетических операторов. Одним из основных недостатков современных инструментальных средств является или полное отсутствие GUI, или исключительная его бедность (большинство средств предоставляют текстовый интерфейс, а если GUI и присутствует, то он представляет из себя одно окно с несколькими текстовыми полями для ввода параметров). В нашем конструкторе мы попытались устранить эти недостатки, добавив, вместе с тем, довольно много новых возможностей, которые будут описаны ниже.

4. Общая структура конструктора

Как видно из приведенного выше обзора большинство существующих на сегодняшний день конструкторов обладают рядом серьезных недостатков: небольшим спектром решаемых задач, слабым графическим интерфейсом, отсутствием какого-либо межсетевого взаимодействия и т.д.

Но одной из основных проблем, с которыми приходится сталкиваться пользователю является проблема формальной постановки задачи и задание параметров работы генетического алгоритма. Так, например, удачный подбор начальной популяции позволяет заметно сократить время поиска оптимального решения.

Детальный анализ генетических алгоритмов дает возможность выделить три основных этапа задания входных данных: кодирование особи, параметризацию популяции и определение взаимодействия между популяциями.

На первом из вышеперечисленных этапов пользователь задает внутреннюю структуру особи и набор методов для работы с ней: инициализацию, мутацию, получение потомка, функцию качества особи и т.д.

Наиболее крупными структурами, с которыми работает генетический алгоритм, являются популяции. Каждая популяция представляет собой группу особей, над которой определены механизмы мутации, скрещивания и селекции. Популяций может быть несколько и на этапе параметризации для каждой популяции задается динамика изменения ее параметров во времени. К таким параметрам относятся: размер популяции, количество мутаций и скрещиваний в каждый момент времени, выбор родителей и кандидатов на мутацию, схема селекции и др. Здесь же определяется механизм построения начальной популяции.

Взаимодействие между популяциями происходит за счет механизма миграций, определение которого происходит уже на последнем этапе задания входных данных. Здесь же задаются условия нормального прекращения работы алгоритма как для каждой популяции так и для всего алгоритма в целом.

Рассмотрим каждый из этих этапов более подробно.

5. Кодирование особи.

5.1. Внутренняя структура особи

Как уже отмечалось выше, на этом этапе задается внутренняя структура особей и определяется набор функций для работы с ней. Основная проблема, решаемая на этом этапе, - постановка задачи.

Особь состоит из нескольких хромосом. Их число фиксировано и задается пользователем в качестве одного из основных параметров. Каждой хромосоме ставится в соответствие ее порядковый номер. Таким образом, можно рассматривать особь как массив хромосом.

Инициализацию особи можно рассматривать как процесс инициализации (заполнения некоторыми значениями) каждой хромосомы Аналогично мутация особи представляет собой случайные изменения в одной или нескольких хромосомах. Скрещивание определяется как создание потомка двух или более родителей за счет обмена генами хромосом, имеющих одинаковые порядковые номера.

Для каждой хромосомы определяется ее длина и метод кодирования, соответствующие ему методы инициализации, мутации и скрещивания, а также, если необходимо, некоторый набор системных функций.

Следует отметить, что изменение значений ген или метода кодирования одной хромосомы не влечет за собой изменений в других хромосомах особи. В итоге хромосомы одной особи оказываются как бы независимыми друг от друга. В большинстве случаев это требование позволяет заметно упростить процесс постановки задачи. Но в некоторых конкретных ситуациях эта независимость может наоборот усложнить процесс кодирования особи, поэтому в конструкторе была предусмотрена возможность нарушения данного требования.

Каждая хромосома представляет собой набор генов, имеющих одинаковую внутреннюю структуру. Число ген в хромосоме может быть либо постоянно, либо меняться со временем. Каждый ген состоит из некоторого заранее заданного числа полей разного типа. Именно в них и хранится конкретное значение гена. Как правило, оно явно или неявно зависит от значений других ген хромосомы. Эта зависимость определяется с помощью метода кодирования хромосомы. Таких зависимостей может быть очень много, поэтому вместе с конструктором поставляется лишь библиотека наиболее часто используемых методов. Для кодирования хромосомы пользователь может воспользоваться одним из них. В случае же необходимости ему предоставляется возможность написать самому на одном из объектно-ориентированных языков программирования требуемый метод кодирования. Это не приведет к выдвижению каких-либо дополнительных требований к входным данным ни при кодировании других хромосом, ни на последующих этапах задания входных данных.

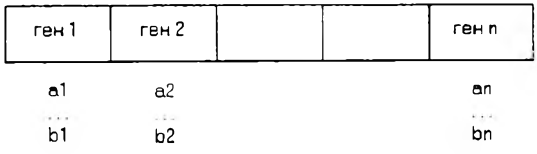
Рассмотрим лишь некоторые из поставляемых с конструктором методов кодирования хромосом и соответствующие им способы инициализации, мутации и скрещивания:

5.1.1. Метод многомерного куба[3]

Это, пожалуй, наиболее распространенный метод кодирования хромосом. Для каждого гена пользователь задает диапазон, в котором могут изменяться его значения. В простейшем случае он может быть задан с помощью минимального и максимального значений, принимаемых геном. Таким образом, область поиска задается в виде многомерного прямоугольного параллелепипеда. С помощью явного задания системы уравнений, которым должна удовлетворять хромосома, определяется некоторая подобласть этого параллелепипеда, в которой и осуществляется поиск решения.

---

3 На сегодняшний день не существует устоявшихся названий методов кодирования хромосом, поэтому приводимые здесь названия используются лишь в рамках данного конструктора генетических алгоритмов



В общем случае эта подобласть может быть весьма сложной, и такая универсальность метода является его неоспоримым достоинством. С другой стороны объем области поиска может быть много меньше объема ограничивающего ее n-мерного параллелепипеда, что приводит к большому числу неудачных попыток задания значений хромосомы. Так, например, в частном случае объем области поиска может быть равен нулю. Поэтому использование других методов кодирования может заметно ускорить процесс нахождения требуемого решения.

Рассмотрим некоторые возможные функций для работы с этим методом кодирования.

Инициализация. Каждому гену случайным образом ставится некоторое число из области его значений. Далее проверяется, принадлежит ли хромосома области поиска. Если нет, то процесс повторяется сначала до тех пор, пока значение хромосомы не попадает в область поиска.

Мутация. Некоторым образом выбирается ген хромосомы, над которым будет произведена мутацию, и его предыдущее значение заменяется новым. Затем происходит проверка хромосомы на принадлежность области поиска. Если хромосома не удовлетворяет заданной системе уравнений, процедура мутации повторяется снова.

Скрешивание. Как уже отмечалось выше, особь может иметь двух или более родителей. Поэтому для каждого гена хромосомы потомка случайным образом выбирается один из родителей и происходит копирование из него потомку значения гена. Далее, как и в случае мутации, происходит проверка на принадлежность хромосомы области поиска и повторение скрещивания в случае, если хромосома не удовлетворяет заданной системе уравнений.

Пользователем также задается, какие действия должны быть предприняты системой в случае, если при данных ограничениях на область с k-ой попытки не удалось провести инициализацию, мутацию или скрещивание.

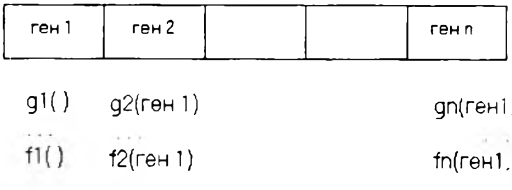
Следует отметить, что в общем случае область изменения значений каждого гена будет представлять собой не один отрезок, а некоторое их множество, но к какому-либо существенному изменению метода кодирования или функций для работы с ним это не приводит. Это же замечание верно и для остальных рассматриваемых здесь методов кодирования.

5.1.2 Метод динамических областей

Пусть хромосома состоит из n перенумерованных ген. Для каждого гена пользователь задает функции, которые позволяют найти область изменения его значений, зная значения предыдущих ген. Таким образом заранее фиксированной

оказывается лишь область изменения значений первого гена, с которого и начинается процесс кодирования хромосомы этим методом.

Как и в предыдущем методе, рассмотрим случай, когда область значений каждого гена задается двумя функциями, определяющими минимальное и максимальное значения данного гена.



Метод динамических областей позволяет довольно просто и эффективно кодировать деревья перебора или аппроксимировать области поиска с помощью прямых и эллипсов. Особенностью данного метода является неравнозначность ген, в результате чего происходит неравномерное распределение их значений. В некоторых задачах это может привести к тому, что поиск решения будет осуществляться лишь в какой-то подобласти области поиска. Поэтому при необходимости могут быть задействованы специальные механизмы решения данной проблемы: распределение вручную приоритетов получения геном того или иного значения, схемы Маркова различной глубины и т.д.

Инициализация. Случайным образом из отрезка [f1,g1] выбирается значение первого гена. Это позволяет определить область изменения значений второго гена и его конкретное значение. Зная значение первого и второго ген можно определить область изменения значений третьего гена и уже его конкретное значение. Так продолжается до тех пор пока не будет определено значение последнего гена. В общем случае каждый ген состоит из нескольких полей, поэтому для некоторых границ fi(ген1,...,генi) и gi(ген1,...,генi) операции сравнения больше/меньше могут быть не определены. Это является своеобразным «подводным камнем», приводящим к откатам или даже повторной инициализации особи. Такая же проблема возникает при мутации и скрещивании закодированных данным методом хромосом.

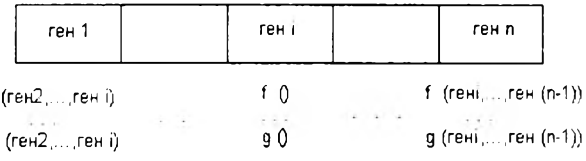
Мутации. Как и в предыдущем методе, некоторым образом выбирается ген хромосомы над которым будет произведена мутация и определяется область его значений. После этого ген получает некоторое новое значение из этой области. Но в свою очередь это приводит к изменению областей значений следующих за ним генов. Поэтому возникает необходимость перекодирования этих ген, т.к. находящиеся в них старые значения могут не попадать в новую область изменения своих значений.

Скрещивание. Этот процесс состоит из n шагов (по количеству ген), на каждом из которых определяется значение одного гена. На первом шаге выбирается один из родителей, чье значение первого гена передается потомку. Это позволяет определить область изменения значений следующего гена потомка. На втором шаге случайным образом среди родителей, у которых значение второго гена принадле-

жит этой области, определяется тот, чей ген будет передан потомку. Знание значений первого и второго генов потомка позволяет определить область изменения значений третьего гена и перейти к следующей итерации. Здесь возможна ситуация, что на некотором шаге ни у одного из родителей ген не попадает в область изменения значений, тогда производится откат или, в зависимости от заданных параметров, процесс скрещивания начинается с начала.

Если с k-ой попытки не удалось произвести инициализацию, мутацию или скрещивание, то пользователю, как и в предыдущем методе, предоставляется возможность определить дальнейшие действия системы: сгенерировать исключение, попытаться еще и т.д.

Обобщением данного метода является предоставляемая пользователю возможность выбора гена с которого начинается кодирование хромосомы.

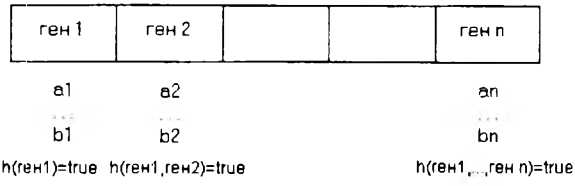


Еще одним обобщением метода является возможность задания, по аналогии с первым методом, некоторого набора функции которым должна удовлетворят хромосома. Такое обобщение будет верно практически для всех методов кодирования хромосом, представленных на сегодняшний день в конструкторе.

5.1.3 Метод пошаговых проверок

Данным метод объединяет в себе ряд характерных черт двух предыдущих. Пусть гены хромосом опять перенумерованы. Для каждого из них пользователь задает диапазон, в котором могут изменяться его значения. Рассмотрим опять простейший случай, когда этот диапазон определяется с помощью минимального и максимального значений принимаемых геном. Таким образом, как и в первом из рассмотренных нами методов, область поиска задается в виде многомерного прямоугольного параллелепипеда.

Но для каждого гена также задается булевская функция, которая должна принимать значение истинности при подстановке в нее конкретных значений данного и предыдущих генов.



Этот метод позволяет эффективно кодировать области поиска $\{(x_1,x_2,...,x_n)$: $A<F(f1(x_1,y_1),f2(x_2,y_2),...fn(x_n,y_n))<B\}$4, где $(y_1,y_2,...,y_n)$ - некоторая фиксированная точка, A и B - неотрицательные числа, а на функции $f_i(x_i,y_i)$ наложены определенные ограничения. Именно таким требованиям удовлетворяют области, задаваемые с помощью наиболее распространенных метрик: $F(X,Y)=\sum|x_i-y_i|$, $F(X,Y)=\sum(x_i-y_i)^2$ и $F(X,Y)=\max|x_i-y_i|$, где $X=(x_1,...,x_n)$ и $Y=(y_1,...y_n)$. Это позволяет в достаточно удобной форме определять окрестности точек, кодировать расстояния и т.д.

Инициализация. Как и в методе многомерного куба, каждый ген, начиная с первого, случайным образом получает некоторое значение. Если оно вместе со значениями предыдущих ген не удовлетворяют уравнению h() инициализируемого гена, происходит его повторный выбор. Так продолжается до тех пор пока значения ген не будут удовлетворять уравнению, после этого аналогичные действия проделываются для следующего гена. Если для одного из ген в течении достаточно долгого периода не удается подобрать значение, инициализация может быть повторена с начала.

Мутация. Определенным образом выбирается ген хромосомы, над которым будет произведена мутация. Если при новом значении гена соответствующая ему формула принимает значение ложно, производится повторная мутация этого же гена до тех пор, пока формула не примет значение истинности. После этого происходит проверка на то, чтобы значения следующих за мутированным геном ген удовлетворяли соответствующим им уравнениям, при необходимости производится изменение уже их собственных значений.

Скрещивание. Этот процесс состоит из n шагов (по количеству ген), на каждом из которых определяется значение одного гена. На первом шаге выбирается один из родителей, и значение его первого гена передается потомку. Оно, несомненно, удовлетворяет уравнению первого гена. На следующем шаге случайным образом среди родителей, на значении второго гена которых формула с учетом первого гена потомка имеет значение истинности, выбирается тот, чей ген будет передан потомку. Далее аналогичные действия производятся для определения значения третьего гена потомка и т.д. Здесь возможна ситуация, что на некотором шаге ни у одного из родителей ген не попадает в область изменения значений, тогда производится откат или же процесс скрещивания начинается с начала. Подобного рода проблемы могут возникнуть при инициализации и мутации. Поэтому если с k-ой попытки не удалось произвести одну из этих операций, пользователю, предос-

4 В общем случае область может выглядеть

$$\{(x_1,x_2,...,x_n): A<F(f1(x_1,y_1),f2(x_1,y_1,x_2,y_2),...fn(x_1,y_1,x_2,y_2,...,x_n,y_n))<B\}$$

тавляется возможность определить дальнейшие действия системы: сгенерировать исключение, попытаться еще и т.д.

Для всех вышеперечисленных методов существует целый ряд их обобщений, которые также поставляются вместе с конструктором: динамическое изменение области поиска (например, в зависимости от времени), модификации методов на случай хромосом с переменным числом ген, возможность задания нелинейной многомерной структуры хромосомы (которая может также еще и динамически изменяться) и т.д.

Для примера рассмотрим модификацию метода многомерного куба на случай хромосом переменной длины.

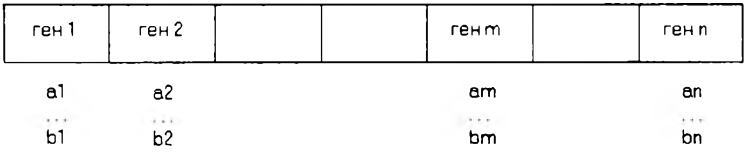
5.1.4 Метод разных длин.

В данном случае пользователем задаются:

-минимальное и максимальное возможное количество ген в хромосоме. Пусть эти числа соответственно m и n. В конструкторе имеется возможность задания n равным бесконечности. Это означает, что количество ген может быть любым числом большим m. Данный случай мы рассматривать пока не будем.

-для каждого гена задается фиксированная (не зависящая от длины хромосомы) область изменения его значений.

Программа в процессе поиска сама должна подобрать приемлемые длину хромосомы и значения, входящих в нее ген. Это допускает одновременное наличие в популяции особей с разными длинами хромосом.



Данный метод может быть эффективно использован при решении задач оптимального распределения ресурсов в случае, когда информация о количестве этих ресурсов минимальна (например, известно лишь что надо затратить как можно меньшее количество этих ресурсов). Пример: расположить неподвижные спутники связи над Землей так, чтобы они могли посылать сигналы в любую точку планеты, при этом количество спутников должно быть минимально.

Инициализация. Для каждой особи случайным образом определяется длина ее хромосомы. Далее опять с помощью датчика случайных чисел для каждого гена определяется его значение.

Мутация. Здесь возможны два вида мутаций, которые могут использоваться одновременно:

- за счет изменения значения одного из ген. Используемый в данном случае механизм аналогичен механизму мутации метода многомерного куба.
- увеличения/уменьшения длины хромосомы на единицу.

Скрещивание. Длина хромосомы потока L вычисляется по некоторому предварительно заданному правилу (например, как среднее арифметическое длин хромосом его родителей). Теперь мы можем «забыть» о тех родительских генах, номе-

ра которых больше L, и производить скрещивание также как это делалось в методе многомерного куба.

Данный метод может быть легко обобщен за счет введения системы булевых уравнений f(ген1,ген2,...,ген n, length)=true, которым должна удовлетворять хромосома. В данном случае под параметром length подразумевается длина хромосомы.

Следует отметить, что приведенные выше подходы кодирования хромосом, хоть их описание и производилось в терминах генетических алгоритмов, могут быть применены также при решении задач и другими методами, например, полным перебором. Теперь мы закончим с методами кодирования и перейдем к дальнейшему рассмотрению этапа кодирования особи.

5.2. Функция качества и способы ее задания

Возможна ситуация, когда в процессе работы алгоритма у большинства «хороших» особей значения i-ых хромосом окажутся одинаковыми. Тогда можно предположить, что и у искомого решения i-ая хромосома будет содержать такое же значение. Это бы позволило бы зафиксировать значение i-ой хромосомы, уменьшив тем самым время работы алгоритма. Обобщением вышесказанного является следующий механизм:

Для каждой хромосомы задается ее функция качества Q_i5(ген1,...,ген n),а функция качества особи определяется как $Q=Q(Q_1,...,Q_n,H_1,...,H_n, t)$, где Hi - значения ген i-ой хромосомы, t- время. Если оказывается, что у большинства «хороших» особей Q_i одинаковы (т.е. эти хромосомы особи одинаково хорошо приспособлены к окружающей среде), то имеет смысл производить фиксацию значений этих хромосом, которые, в общем случае, могут и различаться. После остановки работы алгоритма значение i-ой хромосомы решения может быть определено, например, с помощью перебора этих фиксированных значений. Однако при использовании данного механизма следует быть осторожным, т.к. функция Q может быть неустойчивой по значениям ген зафиксированной хромосомы. Еще одно достоинство введения Q_i - возможность их параллельного вычисления при нахождении функции качества особи.

Теперь рассмотрим способы задания функций качества хромосом и особи в целом. Таких способов на сегодняшний день в конструкторе предусмотрено четыре:

1. Выражение, вводимое пользователем в виде строки. Здесь в качестве переменных выступают: Q_i (функция качества i-ой хромосомы), H_iG_k.field (значение поле field k-ого гена i-ой хромосомы), t - время. Также имеется набор математических функций, который позволяет расширить возможности данного метода.

2. Написание функции качества на одном из объектно-ориентированных языков программирования. В этом случае появляется уже возможность использования условных операторов, циклов и т.д.

3. Использование нейросетей для аппроксимации функции качества. Особенно эффективно генетические алгоритмы используются для решения задач, которые по каким-то причинам не могут быть четко сформулированы. Поэтому не исклю-

5 Иногда функцию качества хромосомы также называют функцией фиксации значений хромосомы

чены ситуации, когда явное задание функции качества сопряжено с большими трудностями или вообще невозможно. Именно в таких случаях и используются нейросети. Пусть имеется набор особей каждой из которых сопоставлено некоторое число, характеризующее ее приспособленность к окружающей среде. На этом наборе особей происходит обучение нейросети, которая в дальнейшем будет использоваться в качестве функции качества.

4.Аппроксимация функции качества с помощью численных методов.

Подводе итог всему вышесказанному перечислим те данные, которые пользователю необходимо задать на этапе кодирования особи:

1. Количество хромосом в особи; оно фиксировано и не изменяется в течении работы всего алгоритма

2. Для каждой хромосомы: определить поля входящих в нее ген, метод кодирования и соответствующие ему механизмы инициализации, мутации и скрещивания, функцию качества хромосомы Q_i, задать ряд системных функций (как правило, все они генерируется автоматически, но, при необходимости, их можно задать и вручную)

3. Функцию качества особи как $Q=Q(Q_1,...,Q_n,H_1,...,H_n,t)$, где H_i - значения ген i-ой хромосомы, t - время.

6. Параметризация популяции

Данные, задаваемые на этом этапе, позволяют уменьшить время, затрачиваемое на поиск требуемого решения.

К обязательным из задаваемых здесь параметров можно отнести: способы инициализация начальной популяции, механизмы мутации и скрещивания, задание способов естественного отбора и выбора мигрантов (только в случае нескольких популяций). Также здесь задается и ряд вспомогательных параметров: динамика изменения размера популяции во времени, зависимость числа скрещиваний и мутаций на каждом шаге от времени, зависимость числа появившихся детей на каждом шаге от времени и номера скрещивания и т.д.

Существует огромное количество вариантов задания всех этих параметров. Поэтому для каждого из них поставляется небольшая библиотека, где прописаны наиболее типичные подходы их задания:

Инициализация начальной популяции сильно зависит от конкретной решаемой задачи. Рассмотрим лишь некоторые способы инициализации:

Случайным образом. Это, наверное, самый популярный способ инициализации начальной популяции. Здесь особи начальной популяции получают свои значения с помощью датчика случайных чисел.

Нейросеть. Этот способ оказывается эффективен, когда с помощью генетических алгоритмов решается много идейно достаточно близких задач. Например, данный подход используется при разрезании графа-алгоритма с помощью генетических алгоритмов в задачах автоматического распараллеливания программ. Обучение нейросети происходит на основе результатов выполнения предыдущих запросов.

Посредством задания расстояния между особями. Это позволяет равномерно распределить начальную популяцию по области поиска, что, в свою очередь, уменьшит вероятность нахождения локального решения.

Также, несомненно, пользователю предоставляется возможность в явном виде задать начальную популяцию или самому написать функцию-инициализатор.

Механизмы мутации. Здесь, в основном, определяется количество мутаций в каждый момент времени, производится выбор кандидатов на мутацию и т.д. Одним из возможных вариантов выбора кандидата на мутацию является следующий: каждой особи популяции ставится в соответствие по ее номеру (который характеризует то, сколько особей имеют функцию качества лучше, чем у данной особи) некоторое число, определяющее ее вероятность стать мутантом.

Механизмы скрещивания. Здесь также определяется количество скрещиваний в каждый момент времени, для каждого скрещивания производится выбор родителей и определяется число появившихся в результате этого скрещивания детей и т.д. Примером возможного метода выбора k-ого родителя может служить пример, приведенный для механизма мутаций.

Естественный отбор. Здесь определяется схема естественного отбора. Возможно два варианта:

-жесткая схема естественного отбора, т.е. в итоговую популяцию войдут только лучшие особи.

-мягкая схема естественного отбора, когда каждая особь имеет возможность попасть в итоговую популяцию. Существует несколько способов задания вероятности попадания особи в итоговую популяцию. Наиболее распространены два варианта:

-для каждой особи по ее номеру ставится в соответствие вероятность ее попадания в итоговую популяцию.

-вероятность попадания каждой особи в итоговую популяцию определяется по формуле $p_k=Q_k/Q$, где Q_k- функция качества i-ой особи, $Q=\sum Q_k$ - сумма функций качества особей, участвующих в естественном отборе.

Также на этом этапе могут определяться динамика изменения размера популяции во времени, сколько в новую популяцию войдет мутантов, потомков, мигрантов и т.п.

Выбор мигрантов. Здесь также возможны две схемы выбора мигрантов: жесткая и мягкая. Механизмы их работы таковы же как и при естественном отборе.

Следует отметить возможность предоставления или не предоставления возможности участвовать в естественном отборе родителей, особей на базе которых создаются мутанты или «копий» особей-мигрантов.

7. Определение взаимодействия между популяциями.

Генетическим алгоритмам изначально присущ мощный внутренний параллелизм. И одним из способов распараллеливания генетических алгоритмов является использование нескольких популяций.

На этапе взаимодействия между популяциями задается количество популяций, а также, сколько и в какую популяцию будет передано мигрантов. В каждый момент времени из популяций «мигрируют» несколько особей. Какие особи будут мигрировать (и их количество) из популяции для каждой популяции определяется еще на этапе параметризации. На рассматриваемом же этапе задается, в какую популяцию каждая из этих особей будет мигрировать.

Также здесь определяются условия нормальной остановки работы генетического алгоритма. Перечислим некоторые из них:

1.Окончание выделенного лимита времени.

2.Функция качества «лучшей» особи достигла определенного заранее заданного значения. Это позволяет решать задачу нахождения для некоторой функции переменных, на которых она достигает некоторого своего значения.

3.Разность между функциями качества особей меньше некоторого фиксированного значения

4.Появление в популяции особи (или особей), которые близки к некоторой эталонной особи в смысле расстояния Хэмминга (или какого-то другого, заданного пользователем).

Возможны также как различные комбинации выше перечисленные условий, так и написание условий остановки на одном из объектно-ориентированных языков программирования.

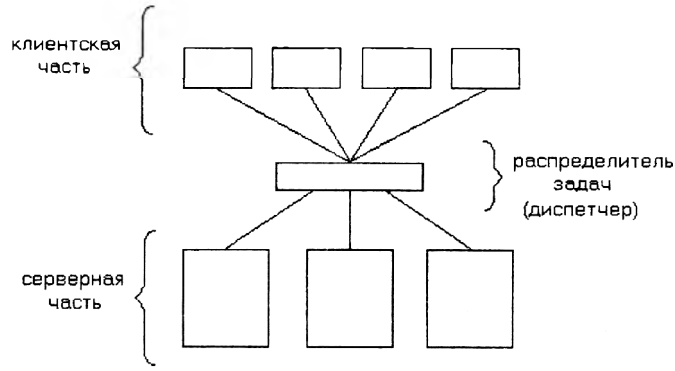
На этом же этапе задаются точки сохранения, точки показа промежуточных результатов работы алгоритма и т.д.

8.Интерфейс.

Как уже отмечалось, одним из основных недостатков существующих на сегодняшний день конструкторов является отсутствие в них удобного пользовательского интерфейса. NS GALAXY предоставляет удобный GUI, написанный на Java с использованием библиотеки Swing, что дополнительно позволило реализовать такие функции, как, например, независимость внешнего вида GUI от платформы, на которой вы запускаете программу. Если вы работаете под Unix, ваш GUI может выглядеть так, как если бы вы работали под Windows. Важное место отводится представлению результатов работы ГА. Пользователь может в динамике наблюдать графики изменения некоторых параметров, изменять некоторые параметры и сразу же видеть, как эти изменения отражаются на сходимости алгоритма.

9. Межсетевая архитектура.

При рассмотрении межсетевого взаимодействия между удаленной пользовательской машиной и сервером можно выделить следующие три этапа работы системы: клиентская часть, распределитель задач(диспетчер) и серверная часть.



Клиентская часть представляет собой полностью интегрированную оболочку взаимодействия между пользователем и конструктором. Ее основная задача - поддержка удобного пользовательского интерфейса и средств визуализации, взаимодействие с диспетчером задач и серверной частью. После задания пользователем параметров работы конструктора и первоначальной проверки их корректности, запрос отсылается на распределитель задач, где он автоматически встает в очередь на выполнение. Диспетчер в каждый момент времени отслеживает работу серверной части и в зависимости от загруженности каждой из серверных машин, принимает решение о том, какая из них будет выполнять тот или иной запрос из очереди. Но возможны ситуации, что распределитель задач и сервера оказываются весьма удаленными друг от друга[6]. Это приводит к тому, что некоторые серверные машины, успевшие уже выполнить поставленную задачу, но не получившие еще новую, будут простаивать. Чтобы не было такой ситуации, каждый сервер имеет свою локальную очередь задач, размер которой определяется диспетчером исходя из удаленности и грубой оценки того, сколько времени требуется на выполнение того или иного запроса.

В целом же такая архитектура построения системы дает целый ряд преимуществ: равномерная загруженность серверных машин, высокая степень надежности и масштабируемости, возможность параллельной обработки запросов, безболезненный переход на новые версии программы и многое другое.

Следует отметить, что клиентская часть написана полностью на Java, что делает ее легко переносимой с одной платформы на другую. По тем же соображениям серверная часть тоже была написана на Java. В силу ряда причин в ближайшее время планируется написание еще одной серверной части, но уже на C++, используя библиотеку параллельного программирования MPI, что позволит увеличить скорость выполнения запроса. Описанный здесь механизм работы системы позволяет использовать обе эти версии одновременно.

В ближайшее время планируется добавить в конструктор новые методов кодирования, различные способы взаимодействия с базами данных и прикладными

---

6 Имеется в виду не физическая, а логическая удаленность (т.е. то, сколько в среднем затрачивается времени на передачу сообщения)

программами. Также в Galaxy весьма активно используются нейросети, поэтому пользователю будет предоставлена возможность самому выбирать архитектуру нейросети, используя подключаемый нейросетевой конструктор.

Литература

[1]N.H.Packard, *Complex Syst.* 4(5) (1990).

[2]W.D.Hills, *Physica D* 42, 228 (1990).

[3]L.Davis and S. Coombs, in *Proceeding of the Second International Conference on Genetic Algorithms*, J.J.Grefenstette, Ed.(Erlbaum, Hillsdale, NJ, 1987), pp.252-256.

[4]D.E.Glover, in *Genetic Algorithms and Simulated Annealing*, L.Davis, Ed.(Pitman, London, and Morgan Kaufman, Los Atlos, CA, 1987), pp.12-32.

[5]J.R.Koza, *Genetic Programming*(MIT Press, Cambridge, MA, 1992).

[6]K.A.DeJong, *thesis*, University of Michigan, Ann Arbor, MI (1975).

[7] Н.М. Ершов, Л.Н. Королев, Э.Э. Малютина, А.М. Попов, Н.Н.Попова Применение активных баз данных в прогнозировании. - Вестник Моск. ун-та., Сер.15, Вычисл.матем. и киберн. 1998, N 1, стр.32-43

[8] Levy, Steven. *Artificial Life: the Quest for a New Creation.* Pantheon Book, 1992.

[9] Д.И.Батищев. *Генетические алгоритмы решения экстремальных задач.* 1995.

[10] Davis, Lawrence. *Research Notes in Artificial Intelligence, Genetic Algorithms, and Simulated Annealing.* Morgan Kaufmann. 1987.

[11] Holland, H. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press,1975.

[12] Goldberg, David E. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Westley,1989.

Буряк Д.Ю.

Особенности применения нейронных сетей к задаче распознавания символов

В работе проведен сравнительный анализ применения нейронных сетей и методов классификации по критерию минимального расстояния к задаче распознавания символов. Выявлены условия и особенности применения некоторых стандартных конфигураций нейронных сетей для решения подобных задач. Проведено исследование быстрых нейронных сетей и сетей с удаленными «слабыми» связями.

Введение

В настоящее время известно большое количество примеров успешного применения как стандартных математических методов, так и нейронных сетей для решения задачи распознавания символов: системы FineReader, CuneiForm, OmniPage и др. Точность, обеспечиваемая этими системами, зависит от размера вводимых символов, от разрешения сканера и находится в пределах от 74% до 99%.

Существует множество экспериментальных разработок решения задачи распознавания символов. В 1990 году в Атлантик Сити на конференции по распознаванию образов была представлена нейронная сеть, разработанная для распознавания цифровых почтовых кодов (ZIP-кодов). Она представляет собой частично связанную сеть, с тремя внутренними слоями. На вход подается изображение 16*16 пикселов. Для увеличения способности сети к обобщению и уменьшения объема необходимых вычислений было проведено удаление слабо используемых весов. Нейронная сеть обучалась на наборе из 7300 символов, тестировалась на наборе из 2000 символов. Ошибки распознавания составляли приблизительно 1% на обучающем наборе и 5% на проверочном.

На конференции в Монреале (1990г.) была представлена система распознавания рукопечатных цифр. Поступающие на распознавание изображения приводятся к размеру 16x16 пикселов. После этого они подвергаются дополнительной обработке с целью выделения участков с наибольшими перепадами в яркости. Используемая нейронная сеть имела только один внутренний слой, но применялась совместно с другими алгоритмами. Обучение и тестирование проводилось на символах, взятых из трех независимых баз данных. Из каждой базы данных использовалось от 4000 до 6000 символов на обучение и от 2000 до 4000 символов на тестирование. Процент ошибок существенно менялся в зависимости от базы данных, на которых проводилось тестирование, и составлял 0.60%-2.2%.

Фирма NEC для решения задачи распознавания символов применила сети обратного распространения совместно с обычными алгоритмами, достигнутая точность превысила 99%.

Из приведенных примеров следует, что искусственные нейронные сети часто и достаточно эффективно применяются в оптических системах распознавания символов. Задачей данной работы было выявление условий и особенностей применения некоторых стандартных конфигураций нейронных сетей к задаче распознавания символов, а также их сравнительный анализ с методами классификации по

критерию минимального расстояния, как наиболее часто используемыми при решении подобных задач.

Постановка задачи

Для исследования применения нейронных сетей и методов классификации по критерию минимального расстояния к задаче распознавания символов были приняты следующие методы:

- метод классификации по критерию минимального расстояния, случай единственного эталона [4];
- нейронные сети: частично и полносвязанные двухслойные и трехслойные персептроны.

Для выявление условий и особенностей применения некоторых стандартных конфигураций нейронных сетей к задаче распознавания символов предполагается решение следующих задач:

1. Исследование влияния типовых настроек нейронных сетей на точность распознавания. В качестве таких настроек здесь рассматриваются:
   - удаление из сетей с большим количеством соединений связей с весами, модуль значения которых, меньше некоторого заданного порога;
   - быстрые нейронные сети [3].
2. Исследование способности нейронных сетей проводить эффективное распознавание при использовании различных представлений изображения.
3. Исследование способности нейронных сетей проводить эффективное распознавание без предварительной обработки изображений.
4. Исследование поведения нейросетевых методов при предъявлении им символов с уровнем искажения, превышающем прогнозируемый, т.е. предъявление символов, искаженных больше чем те, на которых производилось обучение сетей.
5. Сравнение результатов, полученных при решении задач из пунктов 1-4, с результатами применения к тем же исходным данным метода классификации по критерию минимального расстояния.

Сравнение указанных выше методов проводилось на специально созданных тестирующих выборках. Для оценки эффективности применения методов были использованы следующие критерии:

- среднее число ошибок на тестирующей выборке;
- среднее время распознавания примеров из тестирующей выборки;

В качестве исходных данных исследуемых методов рассматриваются бинарные изображения прописных символов английского алфавита, формируемые шрифтом Arial, размера 11. Символы локализованы, т.е. каждый символ находится в некотором прямоугольнике, в котором нет элементов других символов, и нормализованы – каждый символ нормализован по наклону, ширине и высоте. Размер изображения символа не более 18*16 пикселей. Входными данными методов являются искаженные изображения этих символов.

Каждый из методов применялся к трем различным представлениям распознаваемых изображений:

- растровое представление (бинарная матрица размера 18*16);

- представление в виде моментов (вектор, элементами которого являются моменты различных порядков, вычисленные для распознаваемого изображения) [5];
- зонное представление (вектор, элементами которого являются значения, вычисленные в рамках зонного метода выделения признаков).

Исследуемые методы

Нейронные сети.

Для исследования были приняты:

- полносвязанные двухслойные и трехслойные нейронные сети прямого распространения [1,2];
- нейронные сети, получившиеся из полносвязанных удалением связей, имеющих вес ниже некоторого порога;
- двухслойные быстрые нейронные сети [3].

Метод классификации по критерию минимального расстояния. Исследовался метод с единственным эталоном для каждого класса символов. Использовалась метрика абсолютного значения.

Если символ представлен в виде бинарной матрицы, то сначала он и текущий эталон должны быть совмещены по своим центрам тяжести. На первом шаге происходит расчет центров тяжести для символа и эталона. Далее центры тяжести символа и эталона совмещаются (т.е. происходит перенос системы координат), и в этой системе координат вычисляется расстояние между матрицами. Была рассмотрена модификация данного метода. Совмещение центра тяжести символа происходит не только по центру тяжести эталона, но и по нескольким точкам, находящимся в некоторой его окрестности, и для каждой точки вычисляется расстояние между матрицами. Далее выбирается наименьшее из полученных расстояний.

Допущения и ограничения

Исходные данные. В качестве исходных данных были приняты прописные английские символы шрифта Arial, размера 11 (неискаженные, а также с внесением искажений).

Предполагается, что «фон» символа не содержит шума. Для получения искаженных символов использовалась разработанная программа генерации формозависимых искажений символа. Входными данными для нее является битовое изображение символа и вероятности сдвига и изменения размеров отрезков символа. На начальном этапе производится обход изображения по строкам с целью выделения непрерывных сегментов, принадлежащих символу. Далее строится ориентированный граф, в вершинах которого находятся найденные сегменты, а ребра соответствуют парам соседних отрезков. Под соседними здесь понимаются отрезки, чьи проекции пересекаются или касаются. На каждой итерации происходит случайный выбор одной из вершин графа и изменение ее размеров и (или) положения в соответствии с вычисленной вероятностью, которая зависит от относительного расположения рассматриваемого отрезка и его непосредственных соседей, а также от задаваемых вероятностей сдвига и изменения размеров элементов символа. Аналогичным образом обрабатываются столбцы изображения. В результате работы программы получается битовое изображение искаженного символа.

В данной работе использовались следующие обучающие и тестирующие выборки:

- Обучающая, подтверждающая и тестирующая выборки для обучения нейронных сетей, применяемых к растровому и зонному представлениям изображений. Для их получения использовались следующий вектор параметров (0.1,0.05,0.1,0.05). Обозначим данную тестирующую выборку как *Выборка 1*.
- Обучающая и подтверждающая выборки для работы с признаками, выделенными методом моментов. Эти выборки содержат описания только эталонных изображений.
- Тестирующая выборка (*Выборка 2*). Использовались значения параметров: (0.15,0.1,0.15,0.1).
- Обучающая, подтверждающая и тестирующая выборки, содержащие неправильно отцентрированные (смещенные к левому краю) изображения символов. В обучающей выборке таких примеров было 2.44% от общего числа, в подтверждающей – 4.76%, в тестирующей – около 1%. Для их получения использовались следующий вектор параметров (0.1,0.05,0.1,0.05).

Примечание. При описании выборок использовался вектор вероятностных параметров (p1,p2,p3,p4), где p1 – максимальная вероятность сдвига концов горизонтальных отрезков, p2 – максимальная вероятность сдвига концов вертикальных отрезков, p3 – максимальная вероятность сдвига середин горизонтальных отрезков, p4 – максимальная вероятность сдвига середин горизонтальных отрезков.

Предобработка изображения символа. Для удаления небольших пробелов и наростов на изображении символа применялись (в ряде случаев) операции заполнения и утоньшения.

Распознавание символа.

Выбор эталонов. В качестве эталонов для метода классификации по критерию минимального расстояния были выбраны неискаженные изображения символов.

Метод классификации по критерию минимального расстояния. В качестве окрестности для модификации данного метода были использованы 4 точки, охватывающие центр тяжести изображения эталона и расположенные в виде креста.

Нейронные сети. Для обучения нейронных сетей использовалась тренировочная выборка размером 1326 элементов (по 51 примеру на каждый символ) и утверждающая (validation) выборка из 676 элементов (по 26 примеров на каждый символ). Тестирующая выборка состояла из 2626 элементов (по 101 тесту на каждый символ).

Обучение нейронных сетей проводилось методом обратного распространения ошибки [1,2]. Начальная инициализация весов проводилась использованием равномерно распределенной случайной величины. Использовался метод мультистарта (в среднем проводилось по 3 запуска алгоритма обучения для каждой нейронной сети).

Размерность входного слоя нейронной сети равнялась размерности вектора признаков, полученного в рамках одного из трех указанных выше представлений.

Размерность выходного слоя всех смоделированных сетей равнялась 26 (количеству букв английского алфавита). В зависимости от варианта нейронной сети каждый из внутренних слоев содержал от 10 до 100 нейронов.

Всего в данной работе было исследовано 36 различных вариантов нейронных сетей.

Результаты

На первом этапе выполнения данной работы проводились сравнительные исследования эффективности применения методов к изображениям символов, представленных бинарными матрицами. Все рассматриваемые здесь нейронные сети были двухслойными, имели входной слой размерности 18*16 и отличались друг от друга только внутренним слоем. Матрицы, сравниваемые методом минимального расстояния, имели так же размерность 18*16.

Результаты применения методов классификации по критерию наименьшего расстояния к тестирующим выборкам 1 и 2 сведены в таблицу 1.

Модификации	Число ошибок (%)	Время работы (мс)
Совмещение по центру тяжести эталона	17.71 (19.35)	2400
Совмещение по окрестности центра тяжести эталона	0.34 (0.69)	8800

Таблица 1. Результаты применения методов классификации по критерию минимального расстояния к выборкам 1 и 2.

Примечание. В скобках указаны показатели работы на тестирующей выборке 2.

В таблице 2 представлены результаты применения двух наиболее точных полносвязанных сетей.

Внутренний слой	Количество связей	Число ошибок (%)	Время работы (мс)
5*5 нейронов	7850	0.08 (0.53)	4300
7*7 нейронов	15386	0.11 (0.46)	8600

Таблица 2. Результаты применения полносвязанных нейронных сетей к выборкам 1 и 2.

Примечание. В скобках указаны показатели работы на тестирующей выборке 2.

В таблице 3 представлены наиболее удачные результаты применения нейронных сетей, полученных путем удаления части связей из сетей, представленных в таблице 2.

Внутренний слой	Порог для удаления связей	Количество связей	Число ошибок (%)	Время работы (мс)
5*5 нейронов	0.2	6345	0.11 (0.65)	4000
5*5 нейронов	0.4	4933	0.76 (1.83)	2900
5*5 нейронов	0.5	4210	1.87 (4.38)	2700
7*7 нейронов	0.08	14135	0.11 (0.42)	8200
7*7 нейронов	0.2	12264	0.19 (0.46)	7300
7*7 нейронов	0.5	7731	0.3 (1.64)	4400
7*7 нейронов	0.6	6294	0.42 (2.17)	4100

Таблица 3. Результаты применения нейронных сетей после удаления «слабых» связей к выборкам 1 и 2.

Примечание. 1. Удалению подвергались все связи с весом по модулю меньшим, чем значение, указанное во втором столбце.

2. В скобках указаны показатели работы на тестирующей выборке 2.

В таблице 4 сведены результаты работы двух лучших вариантов сконструированных быстрых нейронных сетей.

Внутренний слой	Количество связей	Число ошибок (%)	Время работы (мс)
3*12 нейронов	1044	0.72 (2.32)	1300
13*18 нейронов	4212	0.08 (0.46)	3900

Таблица 4. Результаты применения быстрых нейронных сетей к выборкам 1 и 2.

Примечание. В скобках указаны показатели работы на тестирующей выборке 2.

Нейронные сети при введенных условиях распознают символы с достаточно большой точностью (таблицы 2, 3, 4). Низкая точность метода минимального расстояния (таблица 1) при использовании совмещения по центру тяжести эталона объясняется малой приспособленностью данного метода к работе с таким представлением изображения. Точность этого метода существенно возросла, когда стало применяться совмещение по окрестности центра тяжести эталона, но при этом в 4 раза уменьшилась скорость распознавания (таблица 1). Использование предобработки изображений перед распознаванием не привело к увеличению точности.

Высокую точность нейронные сети показывают и на повышенном уровне зашумления символов (таблицы 2, 3, 4).

Удалось добиться увеличения скорости распознавания путем применения быстрых нейронных сетей и удаления «слабых» связей. БНС увеличивают скорость работы, при этом точность распознавания остается такой же как у лучших представителей полносвязанных сетей (таблицы 2 и 4). Причем это подтверждается на обоих вариантах выборок.

Следует отметить недостаток быстрых нейронных сетей. Размерность выходного слоя в рассматриваемом варианте равна 26=2*13. Количество сомножителей в

разложении мало, и это существенно ограничивает выбор архитектур для построения нейронной сети. Это связано с зависимостью количества и размерности нейронных ядер от сомножителей в разложении.

Удаление «слабых» связей позволило определить «избыточность» сети с внутренним слоем 7*7 (таблица 3). Кроме этого в ряде вариантов применением этого метода удалось увеличить скорость работы нейронной сети, оставляя точность распознавания на достаточно высоком уровне (таблица 3).

Из проведенных экспериментов следует, что на данной задаче БНС являются более эффективными, чем нейронные сети, полученные удалением «слабых» связей (таблицы 3, 4).

Все результаты работы нейронных сетей были получены без предварительной обработки изображений символов, как на стадии обучения, так и на стадии распознавания.

На втором этапе проводились исследования применимости методов к изображениям представленным в виде зонных признаков.

Результаты применения метода классификации по критерию минимального расстояния к тестирующей выборке 1 при различных разбиениях входного изображения на зоны приведены в таблице 5.

Вариант разбиения на зоны	Число ошибок (%)	Время работы (мс)
5*5	8.04 (14.32)	451
7*7	7.35 (14.74)	701
9*11	5.56 (8.87)	1183

Таблица 5. Результаты применения методов классификации по критерию минимального расстояния для зонных признаков к выборке 1.

Примечание. 1. В скобках указаны результаты работы данного метода без предобработки изображений.

2. Предобработка всей тестирующей выборки занимает 4000 мс.

В таблице 6 представлены аналогичные результаты для тестирующей выборки 2.

Вариант разбиения на зоны	Число ошибок (%)	Время работы (мс)
5*5	14.89 (22.89)	440
7*7	14.89 (25.89)	760
9*11	10.59 (15.73)	1250

Таблица 6 Результаты работы методов классификации по критерию минимального расстояния для зонных признаков на выборке 2.

Примечание. 1. В скобках указаны результаты работы данного метода без предобработки изображений.

2. Предобработка всей тестирующей выборки занимает 4000 мс.

Применяемое сглаживание изображений символов позволило в 1.5-2 раза повысить точность распознавания (таблица 6).

Использование зонных признаков исследовалось на двухслойных и трехслойных сетях. Применялись такие же варианты разбиения на зоны, что и в методе минимального расстояния.

Результаты наиболее удачных экспериментов приведены в таблице 7.

Нейронная сеть	Число ошибок (%)	Время работы (мс)
Входной слой: 5*5 Внутренний слой: 5*5 Выходной слой:1*26	1.49 (3.27)	1200
Входной слой: 7*7 Внутренний слой: 6*6 Выходной слой:1*26	1.03 (2.25)	2000
Входной слой: 7*7 Внутренний слой 1: 6*6 Внутренний слой 2: 4*5 Выходной слой:1*26	1.64 (3.58)	2300
Входной слой: 9*11 Внутренний слой: 6*6 Выходной слой:1*26	1.03 (3.16)	2550

Таблица 7. Результаты применения нейронных сетей для зонных признаков к выборкам 1 и 2.

Примечание. В скобках указаны показатели работы на тестирующей выборке 2.

Точность метода минимального расстояния для зонных признаков (таблицы 5 и 6) уступает точности нейронных сетей (таблица 7), показывая при этом более высокую скорость распознавания (без учета времени на предобработку изображений). Низкая точность метода минимального расстояния объясняется слишком простой предобработкой изображений. Однако стоит заметить, что даже используемая предобработка отнимает достаточно много времени, понижая, таким образом, скорость распознавания.

При использовании зонных признаков точность классификации построенных нейронных сетей приближается к точности сетей, работающих с растровым изображением символа (см. таблицы 2, 3, 4 и 7). При этом размерности векторов признаков уменьшились, отсюда произошел рост скорости классификации.

Обращает на себя внимание низкая точность, показываемая трехслойной сетью (таблица 7). Кроме данной было рассмотрено еще 2 трехслойные сети с таким же входным слоем, они также показали низкую точность. Вероятными причинами этого являются: малая размерность входного вектора и недостаточный размер обучающих выборок.

Для рассмотренных вариантов разбиения на зоны не удалось построить быструю нейронную сеть с точностью классификации хотя бы как у метода минимального расстояния. Это связано с ограничениями, накладываемыми на структуру быстрых нейронных сетей, которые не позволяют для используемых размерностей входных и выходных векторов построить двухслойную сеть необходимой емкости.

При исследовании нейронных сетей, применяемых к признакам, полученным методом моментов, использовались моменты порядка 6*6 и 11*11. Было смоделировано несколько двухслойных и трехслойных сетей. Исследованные нейронные сети показали плохую способность к обучению и классификации, и это при том, что, как отмечалось выше, они применялись исключительно к эталонным изображениям символов. В таблице 8 представлены лучшие из полученных результатов.

Нейронная сеть	Число ошибок (%)	Время работы (мс)
Входной слой: 6*6 Внутренний слой: 5*5 Выходной слой:1*26	19.23	560
Входной слой: 6*6 Внутренний слой 1: 6*6 Внутренний слой 2: 4*5 Выходной слой:1*26	11.54	1100
Входной слой: 6*6 Внутренний слой: 9*9 Выходной слой:1*26	11.54	1700
Входной слой: 11*11 Внутренний слой: 5*5 Выходной слой:1*26	57.69	980

Таблица 8. Результаты применения нейронных сетей для признаков, полученных методом моментов.

Примечание. Тестирование проводилось на обучающей выборке (1066 примеров).

Метод классификации по критерию минимального расстояния проработал на той же выборке со 100% точностью. Время работы составило для моментов порядка 6*6 – 390 мс, для моментов 11*11 – 730 мс.

В качестве вероятных причин низкой точности распознавания нейронных сетей можно отметить: большие числовые значения входных данных, приводящих, в соответствии с используемой функцией активации, к только двум значениям 1 и 0, появляющимся на выходах нейронов первого внутреннего слоя. Этот факт ограничивает способности сети к запоминанию образов. Предлагается несколько вариантов решения данной проблемы. Первый из них предполагает введение дополнительных внутренних слоев. Это приводит к некоторому уменьшению скорости классификации, но точность при этом увеличивается (таблица 8).

Второй способ связан с увеличением размерности внутреннего слоя. В таблице 8 приведен пример такой сети. Удалось достигнуть такой же точности, как у трехслойной сети. Скорость работы уменьшилась.

Еще один способ связан с нормировкой всех входных значений, делением их на наибольший момент. Но признаки имеют большой разброс числовых значений, поэтому в нормированном векторе признаков будет много чисел близких к нулю. Следовательно, сохранится очень слабое влияние признаков, имеющих небольшие числовые значения.

Для усиления влияния признаков со сравнительно небольшими числовыми значениями, а также для увеличения способности нейронной сети к распознаванию предлагается провести изменения функции активации нейронов первого внутреннего слоя. Логистическую функцию надо сдвинуть вдоль оси ох. Величина этого сдвига зависит от максимального по модулю значения входных данных. В результате сдвига количество значений на выходах нейронов первого внутреннего слоя увеличится. Большее число признаков (даже с небольшими числовыми значениями) будут оказывать влияние на конечный результат распознавания. Этот метод достаточно сложен для реализации, т.к. величина сдвига функции активации зависит от входных данных.

В процессе выполнения данной работы также была исследована способность нейронных сетей обучаться на выборках, в которых часть образов (изображений) сильно отличалась от остальных. Были созданы обучающая и подтверждающая выборки с обычной степенью искажения, но часть примеров в них была отцентрирована отлично от всех остальных. Были построены несколько нейронных сетей, принимающих на вход бинарную матрицу изображения. Они были обучены на данных выборках. При проверке на тестирующей выборке, также содержащей неправильно отцентрированные изображения, было получено от 1.9% до 2.1% ошибочных решений. Следовательно, даже небольшое число изображений, сильно отличающихся от остальных, способно значительно повлиять на качество обучения и распознавания нейронных сетей.

Выводы

1. Быстрые нейронные сети, использующие растровое представление изображения, являются наиболее эффективными из всех рассмотренных методов в данных условиях искажения символов.
2. Нейронные сети являются более эффективными при распознавании символов, чем методы классификации по критерию минимального расстояния.
3. Нейронные сети проводят эффективное распознавание без предварительной обработки изображений.
4. Использование метода классификации по критерию минимального расстояния оправдано, если значительная доля предварительной обработки символов проводится на этапах сегментации строк.
5. Нейронные сети, использующие зонные признаки, обладают более высокой, чем у сетей с растровыми признаками, скоростью распознавания и достаточно высоким уровнем точности (98-99%).
6. Признаки, имеющие большие числовые значения (например, выделенные методом моментов), плохо применимы в нейронных сетях.
7. Метод удаления «слабых» связей позволяет добиться существенного (до 50%) роста скорости работы нейронных сетей, оставляя количество ошибок в допустимых рамках (1%).
8. Однородность обучающей выборки оказывает существенное влияние на результаты распознавания: количество ошибок может возрасти на порядок, если даже небольшое (1-2%) число изображений в выборке смещены относительно центра на несколько пикселей.

Данная работа выполнялась в рамках проекта по разработке информационной системы обработки экспериментальных данных (грант РФФИ N 99-07-90229), который выполняется в МГУ на кафедре АСВК факультета ВМиК.

Литература.

1. Уоссермен Ф. Нейрокомпьютерная техника: теория и практика. Москва. 1992.
2. Symon Haykin. Neural Networks a comprehensive foundation. IEEE PRESS. 1994.
3. Нейроинформатика-2000, 2-я Всероссийская научно-техническая конференция. Сборник научных трудов. Часть 1. Москва. 2000
4. Ту Дж., Гонсалес Р. Принципы распознавания образов. Москва. 1978.
5. Дуда Р., Харт П. Распознавание образов и анализ сцен. Москва. 1976.
6. Рабинер Л., Гоулд. Теория и применение цифровой обработки сигналов. Москва. 1978.
7. Трахтман А.М., Трахтман В.А. Основы теории дискретных сигналов на конечных интервалах. Москва. 1975.

Царьков Д.В.

Использование модульности при верификации распределенных программ.

1. Введение

Одной из основных проблем, возникающих при проектировании и реализации распределенных программных систем, является проверка правильности их работы. Традиционная методика проверки программ – тестирование – представляется в этом случае неудовлетворительной, так как в силу целого ряда причин чисто физического характера, два вычисления распределенной программы на одних и тех же исходных данных способны, вообще говоря, привести к различным результатам. Помимо этого, тестирование требует большого количества трудоемкой ручной работы и неспособно *доказать* корректность программы.

Альтернативой тестированию является формальная проверка правильности программ. При формальном подходе требования к программе формулируются на специальном языке и проверка соответствия программы этим требованиям осуществляется автоматически.

Одним из вариантов формального подхода является *логический подход*, при котором формальный язык описания требований и методы их проверки основываются на языке и аппарате математической логики. В настоящее время чаще всего за основу берут темпоральную логику, поскольку она позволяет достаточно просто формулировать и анализировать свойства программных вычислений, связанных с течением времени.

Основной проблемой, возникающей при проверке свойств программ этим способом, является взрывной рост количества состояний модели при увеличении числа параллельно работающих процессов. Поэтому при практической реализации систем верификации приходится использовать методики сокращения системы переходов: редукцию частичных порядков, использование симметрии модели, абстракцию и т.п.

Известно, что сложные программы имеют модульную структуру. Программа состоит из функционально замкнутых частей, каждая из которых решает какую-то подзадачу. Эти части могут, в свою очередь, состоять из более мелких блоков и т.д. В зависимости от задач, решаемых модулями, они могут быть слабо связаны (запрос сервиса – получение ответа; передача исходных данных – прием результата вычислений) и сильно связаны (обмен промежуточными данными при параллельных вычислениях). В реальных системах стараются сделать связь модулей как можно более слабой, чтобы уменьшить простои и трафик между отдельными исполнителями.

Если интересующее нас свойство *локально,* т.е. целиком относится к одному модулю, то нет необходимости проверять его на реальной модели. Как правило, можно заменить окружение модуля более простым, сохраняя при этом истинность исследуемого свойства. В этом и состоит идея модульной верификации: при определенных условиях заменить большую часть программы набором простых заглушек, сохранив интересующие нас свойства модуля.

Статья организована следующим образом. Во втором разделе описывается темпоральная логика ветвящегося времени. Язык описания программ, порождающий модели для этой логики, описан в третьем разделе. Основные идеи, лежащие в основе модульной верификации, представлены в разделе 4. В разделе 5 описывается методика модульной верификации и доказывается ее корректность. В заключении приводятся результаты экспериментов, полученные с помощью данного метода.

2. Темпоральная логика CTL

Логика вычислительных деревьев (Computational Tree Logic, CTL) [5] предназначена для описания последовательностей событий. Она является расширением обычной пропозициональной логики: помимо формул, истинных на *состояниях* (обычная пропозициональная логика), она содержит формулы *путей*, определяющие последовательности состояний. Формально, *формулы состояния* логики CTL – это:

- Пропозициональная переменная x;
- Конъюнкция, дизъюнкция или отрицание формул состояния: $p \wedge q$, $p \vee q$, $\neg p$ для формул состояния p,q;
- Замыкание экзистенциальным или универсальным квантором формулы пути: Af, Ef для формулы пути f.

Формулы пути – это конструкции Xp, $U(p,q)$, где p,q – формулы состояния.

Семантика формул логики CTL определяется на бесконечных деревьях (S,R) с конечной степенью ветвления. Здесь S - множество вершин дерева, а R – отношение непосредственного следования вершин. Каждый узел s такого дерева содержит оценку $L(s)$ истинности всех пропозициональных переменных в этой вершине.

Истинность формул состояния определяется в узлах дерева, а истинность формул пути – на *путях* – бесконечных последовательностях состояний, являющихся ветвями дерева. При этом дерево называется *моделью* формулы. Формально:

Формула состояния p истинна в вершине s дерева (обозначается $s \models p$), если:

- Пропозициональная переменная x истинна в вершине s, если $x \in L(s)$
- $s \models p \wedge q$, если $s \models p$ и $s \models q$;
- $s \models p \vee q$, если $s \models p$ или $s \models q$;
- $s \models \neg p$, если неверно, что $s \models p$;
- $s \models Ef$, если найдется путь r, начинающийся в s, на котором будет истинна формула f: $\exists r=s,s_1,s_2,\ldots: r \models f$.
- $s \models Af$, если на любом пути r, начинающимся в s, будет истинна формула f:

$$\forall r=s,s_1,s_2,\ldots: r \models f.$$

Формула пути f истинна на пути $r = s_0,s_1,s_2,\ldots$ (обозначается $r \models f$), если:

- $r \models Xp$, если $s_1 \models p$;
- $r \models pUq$, если найдется вершина s_n, такая что $s_n \models q$ и $\forall i: 0 \le i < n: s_i \models p$.

Назовем два пути *эквивалентными относительно формулы пути f* (обозначается $r_1 \sim_f r_2$), если на этих путях одновременно истинна или ложна эта формула.

Каким образом построить модель формулы по программе? Для этого строится промежуточное представление – *структура Крипке*. Это ориентированный граф с выделеной начальной вершиной, вершины которого помечены множеством истинных в них переменных. Бесконечное дерево получается путем развертки этого графа.

3. Модели для логики

Для получения структуры Крипке по программе необходимо определить язык программирования и процесс трансляции из него в структуру Крипке. Мы рассмотрим язык MDL (Model description language) который предназначен для построения моделей реальных программ и их верификации. Он имеет довольно прозрачную автоматную семантику и может быть легко переведен в форму, удобную для верификации - структуру переходов (для автоматной верификации), функцию перехода (для символьной верификации) и т.д.

Программы, описываемые на этом языке, представляют собой множество последовательных процессов, исполняющихся параллельно и взаимодействующих через разделяемые переменные.

Описание программы на языке MDL состоит из трех основных частей:

- описание переменных;
- описание прототипов процессов;
- определение процессов программы.

3.1 Описание переменных

Описание переменных имеет следующий вид:

.VARS <var_decl> {, <var_decl> }
[.METAVARS <var_decl> {, <var_decl> }]
<var_decl> ::= <name> [< <size>]

После ключевого слова .VARS перечисляются все переменные, фигурирующие в описании программной системы. В секции .METAVARS описаны переменные, являющиеся формальными параметрами MDL-процессов (см. ниже). Каждая переменная, помимо имени, может иметь размер - указание максимального числа, которое она способна хранить. Например, переменная, описанная как v < 5 может хранить целочисленные значения от 0 до 4. Если размер отсутствует, считается, что переменная логическая, и может хранить только значения T и $\perp$.

3.2 Описание прототипов процессов

Секция прототипов процессов выглядит следующим образом:

.GENERIC <прототип> { <прототип> }
<прототип> ::= <name> : <size> < <parameter> {, <parameter>}>
[.LOCAL <var_decl> {, <var_decl> }]
<transition> { <transition> }
<transition> ::= <int> <condition> => <int> <action>;

| .if <int> <int> <int> <condition>;
<condition> = {, <elem_cond>}
<action> = {, <elem_act>}

При описании процесса указывается его *размер*, то есть число состояний, и *параметры* - список переменных, которые используются в процессе. Переменные должны быть описаны в блоках .VARS или .METAVARS MDL-программы. Помимо этого, процесс может использовать локальные переменные, описанные в блоке .LOCAL. Эти переменные доступны только внутри данного процесса.

Тело процесса состоит из переходов. Каждый переход определяется описанием начального состояния, условий перехода, конечного состояния и действий. Семантика перехода такова. Переход *n*, cond => *m*, act считается *возможным*, если текущее состояние процесса - *n* и условие cond истинно. Если условие пусто, то переход возможен. Если переход возможен, то он может быть *выполнен*. При выполнении перехода текущим состоянием процесса становится состояние *m* и переменные, упомянутые в act, принимают соответствующие значения. Условие представляет собой конъюнкцию элементарных условий, действие - последовательность элементарных действий.

Для удобства написания MDL-программ введен дополнительный тип перехода - *условный* переход. Конструкция **if** *m, n, p*, cond; имеет следующую семантику:

m, cond => *n*;
m, ~cond => *p*;

Заметим, что эта конструкция работает корректно и при сложных (неэлементарных) условиях. Если условие опущено, то оба перехода получаются неохраняемые (недетерминированный выбор); если условие - символ 1 или 0, то выполняется только одна - ненулевая - альтернатива.

Рассмотрим более подробно элементарные условия, возможные в MDL-процессе.

- Логические переменные.

В условии допустимы выражения p (эквивалентно p = T) и ¬p (эквивалентно p = ⊥). Никакие другие операции над логическими переменными недопустимы.

- Ограниченные целочисленные переменные.

В условии допустимы следующие выражения: v = e, v <> e, v > e, v < e, v >= e, v <= e. Здесь v - переменная, описанная в параметрах процесса либо в секции .LOCAL как v < n, а e - либо целое число от 0 до n-1, либо переменная, описанная точно также, как v. Смысл этих выражений очевиден.

Перед любым выражением с целочисленной переменной может стоять знак отрицания. При этом значение выражения меняется на противоположное.

Элементарные действия над логическими переменными совпадают по форме с элементарными условиями и состоят в присваивании переменной соответствующих значений. Для ограниченных целочисленных переменных определено единственное элементарное действие v = e, смысл v и e - как в условиях.

3.3 Определение процессов программы

За описанием прототипов процессов следует заключительная часть программы - описание процессов, которые составляют MDL-программу. Она начинается

ключевым словом .REAL_MODEL и состоит из объявления нового процесса, указания его прототипа и, возможно, изменении параметров.

.REAL_MODEL <процесс> { <процесс> }
<процесс> ::= <имя_процесса> = <имя_прототипа>
[[<переименование> {, <переименование> }]]
<переименование> ::= <переменная> / <параметр>

Здесь <переменная> должна быть описана в секции .VARS программы в точности так же, как <параметр>. При таком описании в программе образуется копия процесса-прототипа с текстуальной заменой параметров на соответствующие имена переменных. После изменения параметров в описании процесса должны присутствовать только переменные, описанные в блоке .VARS.

3.4 Ограничение взаимодействия

Проблема решения задачи верификации в общем виде довольно сложна, так как нам необходимо зафиксировать множество событий, интересных для наблюдения. При различной организации распределенной программы они будут различны. Например, для взаимодействия через семафоры необходимо наблюдать (и уметь описать!) события «захватить семафор», «освободить семафор», «ожидание семафора»; для взаимодействия через общие переменные – события «записать переменную», «считать переменную». В этой работе мы будем рассматривать взаимодействие параллельных процессов через посылку и прием сообщений. Множество сообщений представлено ограниченным целочисленным типом. Процесс может посылать сообщения через выходные каналы связи и принимать их через входные каналы. Нас будут интересовать события «посылка сообщения m в канал o», «прием сообщения m по каналу i», «ожидание сообщения по каналу i». Все другие виды взаимодействия запрещены.

Рассмотрим одну из возможных реализаций такого взаимодействия. Пусть v – канальная переменная, использующаяся для хранения сообщения, fl – логическая переменная, инициирующая прием или передачу сообщения, и ne – флаг непустоты входного канала. Тогда посылка сообщения m в канал o будет описываться следующим образом:

$n, \sim o\_fl \Rightarrow n+1, o\_fl, o\_v = m;$
$n+1, \sim o\_fl \Rightarrow n+2;$

Прием сообщения m из канала i будет описываться следующим образом:

$n, i\_ne \Rightarrow n+1, i\_fl, m = i\_v;$
$n+1 \Rightarrow n+2, \sim i\_ne, \sim i\_fl;$

Для связи каналов o и i мы отождествляем переменные o_v и o_fl с переменными i_v и i_ne соответственно. Переменная i_fl является локальной для принимающего процесса и описывается в секции .LOCAL. При этом событие «посылка сообщения m в канал o» описывается предикатом $o\_fl=T \wedge o\_v = m$, событие «прием сообщения m по каналу i» – предикатом $i\_fl=T \wedge i\_v = m$, а событие «ожидание сообщения по каналу i» – предикатом $i\_ne=T$.

4. Предпосылки для использования модульности

Все свойства темпоральной логики интерпретируются в терминах путей –последовательностей состояний. При квантификации мы фиксируем множество путей, на котором должно выполняться искомое свойство: квантор «A» интерпретируется «для всех путей в модели истинно свойство», квантор «E» – «в модели найдется путь, на котором свойство истинно». Поэтому, идеальный вариант для моделирования – чтобы всякому пути в программе однозначно соответствовал путь в модели. Это значит, что для формулы f и пути в программе r должен существовать единственный путь r' в модели, такой, что $r \sim_f r'$. При этом, очевидно, формула будет истинна на программе тогда и только тогда, когда она будет истинна на модели. Такой подход называется *«бимоделирование»* (bisimulation, [6]). Недостатки этого подхода: большая сложность процесса построения модели; необходимость построения программы целиком; достаточно большой размер модели (сравним с размером исходной системы).

Слабая практическая применимость этого подхода привела к необходимости дальнейших исследований. Из практики выяснилось, что большинство формул содержат только кванторы всеобщности. Назовем такие формулы A-формулами. Это значит, что в каждом случае нас интересуют **все** возможные пути вычисления. Поэтому нам достаточно, чтобы каждому пути (вычислению) в программе соответствовал путь в модели (нет требования однозначности: в модели могут быть «лишние» пути). Этот подход называется *моделирование.* Тогда из истинности A-формулы на модели будет следовать истинность этой формулы в программе. Недостаток этого метода очевиден: если A-формула ложна на модели, мы не можем ничего сказать об ее истинности в программе. Но зато модель получается значительно (на порядки) меньше исходной системы переходов, и мы можем последовательно уточнять исследуемое свойство с целью точного определения множества путей в модели.

Таким образом, суть модульной верификации состоит в следующем. *В распределенной программе заменить окружение выделенного модуля более простой средой, обеспечивающей моделирование исходной системы.* Для построения модели распределенной программы с целью модульной верификации надо ответить на несколько вопросов. Во-первых, что считать модулем. Во-вторых, какие свойства считать локальными. И, наконец, каким образом строить окружение для модуля, чтобы полученная система моделировала исходную.

4.1 Определение модуля

Наиболее просто считать модулем отдельный MDL-процесс. В большинстве случаев это оправдано. Но иногда исследуемое свойство охватывает несколько процессов (например, «если контроллер послал инициализационное сообщение, то абонент его обязательно получит»). В этом случае модулем можно считать связную часть программы. Конечно, чем меньше эта часть, тем компактнее получится модель. В случае, когда абонент и контроллер значительно разнесены, мы не сможем исключить ни одного промежуточного звена и, следовательно, получим весьма незначительный выигрыш. Лучше всего считать модулем близко расположенные и тесно связанные компоненты программы, например, общая память и ее мультиплексор, шина и контроллер шины, и т.п.

4.2 Локальность свойства

Глядя на структуру распределенной программы, написанной на языке MDL, можно выделить четыре типа переменных. Переменные *i_v* доступны процессу только на чтение, а переменные *o_v* – только на запись. Состояние процесса и переменные *i_fl* – это локальные переменные процесса: только он может читать и изменять их значения. И, наконец, переменные *o_fl* и *i_ne* являются разделяемыми: они доступны обоим процессам (посылающему и принимающему) на чтение и на запись. Но схема взаимодействия процессов такова, что процесс, единожды установив разделяемую переменную в некоторое состояние, ждет, пока абонент не изменит это состояние, и только после этого продолжает свою работу. Поэтому соответствующие флаги можно считать такими же входными и выходными переменными, как и соответствующие им информационные. Помимо этого, есть еще внешние переменные. Это переменные других процессов распределенной программы, к которым исследуемый модуль не имеет доступа ни на чтение, ни на запись. Очевидно, что такие переменные не должны присутствовать в описании локального свойства.

Таким образом, понятно, как определить локальность свойства. Назовем свойство *локальным* по отношению к некоторому модулю, если в нем присутствуют только локальные, входные и выходные переменные этого модуля, причем входные переменные могут встречаться в нем **только** в левой части импликаций: «если входные данные удовлетворяют некоторым ограничениям, то модуль работает корректно».

5. Как построить модель

Итак, нам нужно для выделенного модуля (к которому относится исследуемое локальное свойство) построить окружение, имитирующее работу остальной части параллельной программы таким образом, чтобы получившаяся система переходов была моделью исходной. Поскольку окружение реальной программы, как правило, состоит из многих модулей, нет смысла делать его одним процессом – у него будет слишком сложная структура. Целесообразно составить модельное окружение по крайней мере из стольких отдельных процессов, сколько их в реальном окружении. Но и этот путь довольно сложен: при автоматической верификации мы не можем знать логику работы модулей окружения реальной программы, и, следовательно, неспособны адекватно отразить ее в заглушках. Поэтому предлагается строить отдельные заглушки для каждой канальной переменной.

При таком подходе вид заглушек будет весьма прост – это либо бесконечный прием сообщений (в случае заглушки для выходного канала), либо бесконечная посылка сообщений из некоторого фиксированного множества сообщений M (в случае заглушки входного канала). В последнем случае посылаемые сообщения выбираются недетерминировано. Назовем модуль *максимально замкнутым,* если все входные заглушки генерируют множество сообщений, имеющихся в исходной программе, и только их.

Теорема 1. *Максимально замкнутый модуль моделирует исходную распределенную программу.* **Доказательство** этого утверждения в достаточной степени тривиально. Рассмотрим какое-нибудь вычисление в исходной программе, проходящее

через модуль M. Оно является поведением модуля M, канальные переменные которого недетерминировано принимают значения из множества M_{max}. Но всякое такое поведение имеется и у максимально замкнутого модуля, т.к. заглушки способны выдать любое значение соответствующих канальных переменных.

Эта теорема дает простой и очевидный способ построения заглушек. Но в реальных программах в одном модуле используется только малый процент сообщений, появляющихся во всей программе. При этом могут возникать неприятности следующего порядка: модуль проверяет входные сообщения и выдает ошибку, если они не предназначаются ему. При этом формула становится ложной на модели, но может быть истинной в программе.

Естественным усилением предыдущей теоремы является

Теорема 1'. *Минимально замкнутый модуль моделирует исходную распределенную программу.* Здесь *минимально замкнутый модуль* – модуль, канальные переменные которого принимают в точности те же значения, что и соответствующие канальные переменные в программе.

Основная сложность в применении последнего результата заключается в том, что мы, вообще говоря, не знаем, какие именно сообщения являются входными для данного канала. Это свойство нужно проверять отдельно, и оно может не оказаться локальным для процесса, который шлет исходные данные на вход исследуемому модулю. Например, если процесс просто передает на выход сообщения, полученные им на входе, то множество генерируемых им сообщений нельзя определить локально. Поэтому сейчас такое множество сообщений строится вручную.

6. Заключение

Данная техника была использована в подсистеме верификации среды имитационного моделирования DYANA [1]. Среда имитационного моделирования DYANA предназначена для моделирования программных и аппаратных компонентов встроенных и автономных распределенных систем с целью исследования их работы и производительности. Используя инструментарий, предоставляемый средой, можно описывать модели программ и аппаратуры с любой степенью детальности - от блок-схем до работающей программы на языке высокого уровня.

Исследование эффективности может производиться различными способами - от оценки времени работы реальной программы на исполнителе (подсистема оценки времени выполнения) до отладки распределенной программы и оптимизации выполнения (анализ трасс вычислений программы).

Помимо исследования производительности распределенной программной системы, среда DYANA позволяет проводить верификацию программ с использованием темпоральной логики CTL. Алгоритм верификации описан в [2]. При верификации структура Крипке представляется в символьном виде [3,4], что позволяет эффективно обрабатывать структуры размером 10^{40} состояний и более.

Одной из моделей, рассчитываемых в среде DYANA, является навигационная система самолета, позволяющая выполнять ему задачи полета по маршруту, коррекции местоположения по сигналам от маяков, маловысотного полета и т.п. [7]. При попытке верифицировать навигационный комплекс обнаружилось, что модель слишком велика для верификации стандартными методами: из-за недостатка памя-

ти не удавалось даже просто построить структуру Крипке, описывающую навигационный комплекс. При модульной верификации отдельных компонентов (общей памяти) и систем компонентов (связки датчик-управляющий алгоритм) удалось проверить выполнение спецификаций этих частей системы и доказать корректность их работы в модели навигационного комплекса.

Литература.

[1] Бахмуров А.Г., Костенко В.А., Смелянский Р.Л., Среда моделирования DYANA: синтез, анализ и оптимизация вычислительных систем // Тезисы докладов Всероссийской научной конференции "Фундаментальные и прикладные аспекты разработки больших распределенных программных комплексов" (сентябрь 1998 г., Новороссийск) М.: Изд-во МГУ, 1998 - С. 28-34.

[2] Захаров В.А., Царьков Д.В. Эффективные алгоритмы проверки выполнимости формул темпоральной логики CTL на модели и их применение для верификации параллельных программ//Программирование, 1998, #4, с.3-18

[3] Царьков Д.В. Верификация программ с использованием BDD. // Тезисы докладов IV Всероссийской научной конференции "Дискретные структуры в науке и управлении", в печати.

[4] Burch J.R., Clarke E.M., McMillan K.L., Dill D.L. and Hwang L.J.. Symbolic Model Checking: 10^{20} states and beyond // Information and Computation, **98**, 142-170 (1992)

[5] Clarke E.M., Emerson E.A.. Design and synthesis of synchronization skeletons using Branching Time Temporal Logic, in Proc. Workshop on Logics of Programs, Lecture Notes in Computer Science, Vol. 131, p.52-71 (1981)

[6] Kupferman, O. And Vardi M.V.. On the complexity of branching modular model checking., in Proc. 6th Conferance on Concurrency Theory, LNCS 962, p.408-422 (1995).

[7] The DrTesy project, http://www.first.gmd.de/~drtesy

Сообщения

Рамиль Альварес Х.

Статистический анализ результатов контрольных работ и тестов в МСО "Наставник"

В 1999 году исполнилось 25 лет с начала практического применения автоматизированной системы обучения "Наставник" [1, 2], разработанной в 1972-73гг. проблемной лабораторией ЭВМ Московского университета. 1 марта 1974г. первые две группы студентов третьего курса факультета вычислительной математики и кибернетики прошли на "Наставнике" подготовленный профессором Бахваловым Н.С. автоматизированный коллоквиум по курсу "Численные методы".

Стартовый вариант "Наставника" был разработан и реализован на базе малой цифровой машины "Сетунь 70" [3] и эксплуатировался на факультете ВМиК до мая 1986г. В октябре 1986г. началось использование микрокомпьютерного варианта "Наставника" на базе микро-ЭВМ ДВК-2, который продолжает работать и сейчас. В этом году проходит опытная эксплуатация системы на базе персонального компьютера PC IBM.

Развитие программного оснащения МСО "Наставник"

Программное оснащение стартового варианта "Наставника" состояло из подсистемы "Обучение" и служебной подсистемы подготовки управляющей информации необходимой для работы системы. По результатам использования "Наставника" для автоматизированных коллоквиумов в 1976г. была разработана и реализована специальная компонента, получившая название подсистемы "Экзамен", и соответствующая компонента подсистемы подготовки управляющей информации. Весной 1983г. для проведения тестирования с использованием аппаратуры "Наставника" была разработана подсистема "Тест" и компонента подсистемы "Подготовка" для получения управляющей информации для тестов.

При создании программного оснащения первой микрокомпьютерной версии "Наставника" на базе микро-ЭВМ "Электроника НЦ-04" (1979-83гг.) программное оснащение "Наставника" реализовывалось в диалоговой система структурированного программирования (ДССП) [4], легко переносимой на компьютеры различной архитектуры. В 1986г. был осуществлен перенос системы на микрокомпьютеры унифицированной архитектуры (Электроника 60, ДВК-2). В подсистемах "Обучение" и "Тест" микрокомпьютерной системы была введена возможность записи протоколов занятий (архивов) на магнитную дискетку. Одновременно в программное оснащение была включена подсистема "Обработка". Практическое применение получила лишь обработка архивов тестирования.

Сейчас при переносе "Наставника" на компьютеры PC IBM запись протоколов реализована и в подсистеме "Экзамен". При этом разрабатываются и апробируются процедуры обработки архивов, учитывающие специфику области примене-

ния подсистемы (например, психодиагностическое тестирование, тестирование по математике и т.д.). Для подсистем "Экзамен" и "Тест" реализованы процедуры статистического анализа успешности ответов, позволяющие, в частности, выявлять узкие места в учебных материалах.

Использование МСО "Наставник" на факультете ВМиК

К марту 1974 профессор Бахвалов Н.С. подготовил учебный материал для подсистемы "Обучение" по курсу "Численные методы", который он читал на одном из потоков третьего курса факультета. Первый автоматизированный коллоквиум был проведен аспирантами Бахвалова Н.С., находившегося тогда в командировке заграницей. По возращению лектор по протоколам выставил оценки, которые были потверждены при приеме экзамена в летнюю сессию.Таким образом коллоквиум принимался до 1982г. (ежегодно через систему проходило 150 студентов, при этом некоторые дважды, что разрешалось).

Весной 1978г. руководство факультета приняло решение об обучении с помощью "Наставника" всех студентов 2-го курса языку Фортран. В новом здании факультета был оборудован терминальный класс системы и с сентября в нем начались занятия. Курс, по содержанию соответствующий стандарту Basic Fortran, осваивался студентами при двухразовых занятиях в неделю, за 3-4 недели, после чего они выполняли соответствующее задание в практикуме по программированию. На протяжении последующих лет (до 1986г.) в терминальном классе "Наставника" освоили программирование на Фортране более трех тысяч студентов, аспирантов, преподавателей и научных сотрудников факультетов МГУ и сторонних организаций.

В августе 1983г. проведено первое тестирование на знание английского языка поступивших на первый курс с использованием подсистемы "Тест". Учебный материал был разработан преподавателем кафедры английского языка механико-математического факультета Брагиной Е.В. Результаты тестирования оценивают уровнь знания языка и позволяют сформировать однородные языковые группы. Проверка на практике результатов потвердила эффективного машинного тестирования, а учебные материалы используются и до настоящего времени. За 17 лет через "Наставник" прошло около 5 тысяч зачисленных на факультет студентов.

В марте 1987г. на системе были проведены первые контрольные работы по дифференциальным уравнениям для студентов второго курса. В дальнейшем учебный материал был расширен: окончательный вариант позволяет проводить четыре контрольные, по две в каждом семестре, и охватывает полный курс по этому предмету. Контрольных проводятся постоянно.

С 1987г. в терминальном классе с использованием подсистемы "Тест" в режиме самопроверки проводятся занятия со студентами первого курса по грамматике английского языка.

В апреле 1988г.в "Наставнике" была проведена контрольная работа по теме интегралы из курса "Математический анализ" для студентов первого курса. Эти котрольные с использованием подсистемы "Экзамен" проводятся ежегодно.

Осенью 1989г. для подсистемы "Экзамен" был подготовлен учебный материал по курсу "Оптимальное управление (3-й курс)". В декабре проведена первая контрольная работа, и с тех пор материал используется систематически.

В октябре 1993г. подсистема "Тест" была применена для отбора слушателей подготовительных курсов МГУ для факультета ВМиК. В 1995г.были подготовлены новые учебные материалы - два теста, которые использовались до 1998г. За 6 лет около 1,5 тысяч желающих поступить на курсы прошли автоматизированный отбор.

В 1996 для подсистемы "Тест" под руководство заведующей кафедрой английского языка Саратовской Л.Б. подготовлено учебное пособие "English grammar through computer" по морфологии английского языка, используемое в режиме самопроверки студентами первого курса.

Многолетнее использование учебных материалов

Одним из основных правил, которому следуют разработчики "Наставника" при выборе вносимых изменений в систему при ее модернизации, они не влияли на совместимость по имеющимся учебным материалам. Это и то, что программы базовых курсов на факультете в последние годы практически не меняются, позволило некоторые материалы использовать в течение ряда лет, лишь исправляя в них обнаруживаемые ошибки.

Другой фактор, позволивший в течение многих лет использовать одни и те же учебные материалы - это их защищенность от недобросовестных учащихся. Она обеспечивается, в частности, простой процедурой изменения раскладки вариантов по терминалам. За все годы не обнаружено ни одной попытки использования "отмычек", т.е. набора правильных ответов на упражнения тестов и контрольных работ.

Анализ результатов тестирования

Алгоритм функционирования подсистемы "Тест" таков, что все тестируемые по конкретному тесту работают с одним и тем же набором упражнений, но задаваемым учашимся в разной последовательности, так что рядом сидящие в каждый момент выполняют разные упражнения. Поэтому обработка накопившихся результатов выполнения тестов за многие годы позволяет получить объективные данные как о характеристиках контингента тестируемых, так и о эффективности самих материалов.

Тестирование абитуриентов по английскому языку проводится с целью разбиения их на группы по уровню знаний. Оно производится по сумме числа правильных ответов с учетом весов упражнений в секциях (максимальная сумма равна 200 баллам). Формируются три группы абитуриентов. В первую, разговорную, включаются получившие не меньше 180 баллов; во вторую, продолжающую - получившие от 160 до 179; а третью, начинающую - все оставшиеся.

В первые годы проводилась скрупулезная проверка получаемых с помощью "Наставника" результатов преподавателями кафедры английского языка механико-

математического факультета. Они продолжали параллельно с работой "Наставника" распределять студентов по группам традиционным способом - с помощью индивидуального собеседования. И хотя эти результаты не полностью совпадали, заведующая кафедрой Глушко М.Б. решила произвести распределение по результатам машинного тестирования и проверить его уже в ходе обучения в первом семестре. Практика показала, что автоматизированное тестирование более объективно оценивывает знания студентов чем индивидуальное собеседование, на результаты которого влияет субъективный фактор.

Статистический анализ результатов тестирования в течение 17 лет показывает, что начиная с 1994г. поступившие на первый курс существенно лучше владеют английским языком чем поступавшие в предыдущие годы. При этом ежегодно все лучше и лучше: так процент вошедших в разговорную группу вырос с 21,8% в 1994г. до 27,9% в 1998г., а 1999г. до 38,5%. В Таблице 1 приведены проценты студентов, распределенных в разговорную группу и в разговорную или продолжающую группы за периоды с 1983 по 1989гг. и с 1994 по 1998гг.

Таблица 1.

Период	Общее число тестируемых	% вошедших в разговорную группу	% вошедших в разговорную или продолжающую группы
1983-1989гг. (без 1986г.) 6 лет	1719	7,7%	19%
1994-1998гг. 5 лет	1705	22,8%	52,2%

В 1995г. отбор слушателей подготовительных курсов МГУ для нашего факультета проводился по новым учебным материалам, которые использовались до1998г. Новый материал как и старый состоял из двух тетстов с независимыми наборами задач (по 20 задач) и их попеременно предлагали при отборе. Все эти годы авторы тестов, принимали участие в отборе: просматривали сдаваемые тестируемыми записи решений, обращая внимание на нерешенные задачи и изредко корректируя машинные результаты. Однако в целом результаты автоматизированного отбора соотвествовали реальности.

Статистичсский анализ результатов тестирования показывают, что за период с 1994 по 1997гг. уровень подготовки абитуриентов на подготовительные курсы в целом не изменился - каждый год 40% решало не менее половины задач и около 9% - не менее 15 задач из 20 задаваемых. В 1998г. эти показатели были выше (56% и 17% соотвественно), но следует иметь в виду, что число тестируемых было в два раза меньше и что курсы уже были непосредственно при нашем факультете. Возможно, это повлияло на характер контингента: пришли только те, кто действительно потом поступал на наш факультет.

Авторы учебного материала считали оба теста равноцеными. Однако статистика результатов тестирования показывает, что второй тест выполнялся успешнее

чем первый: процент решивших не менее 15 задач второго теста более чем в два раза превышает аналогичный показатель первого.

Таблица 2.

Вариант	Число тестируемых	% решивших не менее 15 задач	% решивших не менее 10 задач
1	505	5,5%	33,6%
2	521	13,8%	47,7%
Оба теста	1026	9,7%	42%

Статистический анализ архивов подсистемы "Экзамен"

При переносе "Наставника" на пересональный компьютер PC IBM опробывались некоторые процедуры обработки архивов подсистемы "Экзамен". Были обработаны протоколы второй контрольной работы по диффециральным уравнениям (1 семестр 1999/2000 уч/г.). Этот учебный материал использовался многократно с 1987г. Студенты могли не соглашаться с решением системы и преподаватели рассматривали их претензии. В ряде случаев, особенно в первые годы, таким образом обнаруживались ошибки и вносились соответствующие коррективы в учебный материал.

Контрольная работа проверяет знания по четырем темам и по каждой теме имеется 24 задачи. Система выбирает случайным образом задачу в теме и предоставляет две попытки на ее решение. Были обработаны данные по результатам работы 230 студентов. Очевидно, что объем обработанных данных слишком мал, чтобы на их основе делать серьезные заключения. Однако, если упражнение выполняло 10-14 студентов и ни один из них не решил его за две предоставленные попытки, то следует проверить формулировку условия задачи и решение, признанное верным. Таким образом удалось выявить и устранить ряд ошибок.

Рассмотрен 25-летний опыт использования МСО "Наставник" в учебном процессе на факультете ВМиК МГУ. Приведены некоторые выводы из анализа результатов тестирования по английскому языку поступивших на факультет за период с 1983 по 1999гг. и по математике при отборе слушателей подготовительных курсов МГУ для нашего факультета за 1995-99гг. Приведен пример статистической обработки архивов подсистемы "Экзамен"

Литература

1. Брусенцов Н.П., Маслов С.П., Рамиль Альварес Х. Микрокомпьютерная система обучения "Наставник".- М.: Наука, 1990. - 224с.

2. Брусенцов Н.П., Маслов С.П., Рамиль Альварес Х. Методическое пособие по разработке учебных материалов в микрокомпьютерной системе обучения "Наставник".- М.: Изд-во Моск. ун-та, 1992. - 95с.

3. Брусенцов Н.П., Маслов С.П., Жоголев Е.А. Общая характеристика малой цифровой машины "Сетунь 70". // Вычислительная техника и вопросы кибернетики, вып. 10. - Л.: Изд-во Ленингр. ун-та, 1974. - С. 3-21.

4. Диалоговая система структурированного программирования ДССП-80 / Н.П.Брусенцов, В.Б.Захаров, И.А.Руднев, С.А.Сидоров. // Диалоговые микрокомпьютерные системы. - М.: Изд-во Моск. ун-та, 1986. - С. 3-21.

Сидоров С.А.

Стандарт Open Firmware - критический анализ

Рассматривается стандарт на внутреннее программное обеспечение (ПО) «IEEE Std 1275-1994, Standard for boot (Initialization Configuration) Firmware: Core Requirements and Practices», сокращенно называемый «Open Firmware». Вводятся основные понятия, описываются функции внутреннего ПО, главные компоненты системы внутреннего ПО. Анализируются содержание и слабые стороны стандарта.

1. Определения и источники

Под ***внутренним программным обеспечением*** (ВПО) понимается программа, являющаяся неотъемлемой составной частью компьютера - продолжением его аппаратуры. Этот мостик между чистой аппаратурой (hardware) и программными системами (software) получил название ***firmware***. *«Firmware - это размещенное в ПЗУ программное обеспечение, которое управляет компьютером от момента включения до момента передачи управления операционной системе. В обязанности firmware входит тестирование и инициализация аппаратуры, определение аппаратной конфигурации машины, загрузка операционной системы и предоставление возможностей интерактивной отладки в случае неисправности аппаратуры или программного обеспечения.»* [1]. Появилось внутреннее ПО практически одновременно с микропроцессорными системами. В первую очередь оно предназначалось для замены громоздких и дорогих пультов, состоящих из несметного количества лампочек и тумблеров. Пультовые мониторы (так часто называли внутреннее ПО) первого поколения микропроцессоров (1970-е годы: Intel 8080, LSI-11 и др.) занимали около 4К байт и в основном исполняли функции загрузчика и простейшего отладчика в машинных кодах. Современные компьютерные системы просто немыслимы без внутреннего ПО. Более того, разработчики новых микропроцессоров изначально ориентируются на то, что некоторые действия по жизнеобеспечению компьютера будут реализованы программно.

Изначально внутреннее ПО было крепко привязано к конкретному компьютеру - программы писались в основном на языке ассемблера, да и аппаратура, с которой работало ВПО, была весьма разнообразной у различных производителей. Стремительный рост производительности процессоров, то есть возможность использовать языки высокого уровня, и стандартизация периферии и интерфейсов привели к появлению мобильных систем внутреннего ПО. В этом смысле эволюционный путь ВПО аналогичен развитию операционных систем, для которых поворотным моментом стало появление операционной системы UNIX.

Функции сегодняшнего внутреннего ПО весьма разнообразны. К основным можно отнести следующие:

- инициализация аппаратных компонент;
- определение конфигурации компьютера и поиск устройств;
- тестирование устройств компьютера;
- механизм системных вызовов;
- настройка параметров;
- загрузка операционных систем;

- отладчик программного обеспечения.

По существу, внутреннее ПО из небольшой служебной «прослойки» превратилось в самостоятельный программный продукт, хотя для большинства пользователей-прикладников оно остается неведомым. Впервые громко о ВПО заявила фирма Sun Microsystems более 10 лет назад, создав архитектурно-независимую систему внутреннего ПО OpenBoot. На ее основе был разработан стандарт IEEE [1], над которым трудилась рабочая группа из представителей фирм Sun Microsystems, FirmWorks, Force Computers Inc., Apple Computer Inc., IBM Corporation, Antel Inc. Внутреннее ПО, поддерживающее этот стандарт, существует для многих современных процессоров: Sun Sparc, MC68K, PowerPC, Intel x86, MIPS, Digital Alpha и других. Среди шин, поддерживаемых этим ВПО - PCI, SCSI, ISA, VME, USB и другие.

2. Обзор функций внутреннего ПО

Инициализация аппаратных компонент компьютера - первое, что должно выполнить внутреннее ПО. В первую очередь это относится к жизненно важным устройствам - системному контроллеру, устройству управления памятью, консольному устройству ввода-вывода. Остальные устройства могут инициализироваться либо на этом этапе, либо позже - при конфигурировании компьютера или при подготовке к загрузке операционной системы.

Определение конфигурации компьютера и поиск подключенных устройств подразумевает автоматическое опознавание подключенных к компьютеру устройств с использованием протоколов, предусмотренных стандартами шин и стандартами ВПО. Современные компьютеры обычно используют несколько шин для взаимодействия с периферией, поэтому алгоритмы идентификации устройств должны быть привязаны к каждой из них. В то же время, методы идентификации не связаны с конкретной архитектурой компьютера. На опознанные устройства настраиваются простейшие драйверы, имеющиеся в составе внутреннего ПО, которые позволяют выполнять необходимые функции обмена и управления, не претендуя на полноту и эффективность. Их основная задача - обеспечение тестирования устройств и загрузки операционной системы.

Тестирование устройств компьютера - наиболее масштабная и неисчерпаемая сфера деятельности внутреннего ПО. По назначению тесты можно условно разбить на три группы. Тесты функционального контроля по начальному включению (ТФКН или POST - Power On Self Test) выполняются всякий раз при запуске компьютера и предназначены для быстрой оценки его работоспособности. Эксплуатационные тесты позволяют более детально проанализировать состояние аппаратуры, обладают развитой диагностикой. Их назначение - периодическая проверка компьютера пользователем в процессе эксплуатации. Производственные тесты предназначены для разработчиков и производителей, которым часто необходимо проверять отдельные сигналы, узлы устройств, работу в специальных режимах или специальных условиях. Строго говоря, с необходимостью в состав внутреннего ПО входит лишь POST. В то же время, производственные тесты, если есть возможность, обычно не удаляют из готового изделия - это может сильно облегчить поиск и устранение неисправностей. Эксплуатационные тесты, как правило, делаются внешними, то есть загружаемыми в составе ОС или тестовой системы.

Механизм системных вызовов («call back») реализует унифицированный доступ к сервису, предоставляемому внутренним ПО. Этот механизм позволяет использовать имеющиеся в ВПО драйверы устройств и другие системные ресурсы, не заботясь о привязке к конкретной периферии. Хотя солидные операционные системы содержат драйверы, специально для них разработанные, обладающие всеми необходимыми возможностями и обычно более эффективные, существует множество задач, которые не могут по разным причинам пользоваться функциями ОС - в первую очередь, это тесты и загрузчики, а также автономные программы, работающие на «голом» компьютере.

Настройка параметров инициализации, конфигурации компьютера и загрузки ОС обычно происходит в диалоге с помощью меню или набора команд. В процессе настройки модифицируются хранящиеся в неизменяемой памяти (NVRAM) параметры, необходимые для работы ВПО и, возможно, операционной системы. В современных компьютерах количество таких параметров может составлять несколько сотен, причем многие параметры взаимосвязаны, - можно оценить сложность как самого внутреннего ПО, так и программы настройки.

Загрузка операционных систем и других программ - одна из важнейших задач ВПО. Внутреннее ПО должно уметь загрузить программу из любого доступного источника: с диска, ленты, из сети и пр. Для этого должны быть стандартизованы протоколы загрузки как для классов устройств, так и для различных операционных систем. Разнообразие загрузчиков конкретного программного обеспечения обычно нивелируется использованием поэтапной загрузки - сначала грузится простой и маленький загрузчик нулевого уровня, который берет на себя все остальные заботы. Для устройств же обычно оговаривается расположение по умолчанию загружаемой программы, или эта информация хранится среди заданных пользователем параметров.

Отладчик программного обеспечения в любом внутреннем ПО предоставляет возможность установки и снятия точек останова, запуска и трассировки программ, просмотра и модификации памяти и регистров процессора. Если позволяют ресурсы, то в состав отладчика включается дизассемблер и простой ассемблер, позволяющий писать небольшие программы не в кодах машины, а на символьном языке. Часто работа с отладчиком и вообще взаимодействие пользователя с ВПО бывает возможно не только с консольного терминала, но и по сети, что особенно важно для встроенных и бортовых машин.

Стандарт IEEE Std 1275-1994 «Open Firmware» подразумевает также наличие расширяемого и программируемого командного языка, основанного на языке программирования Forth [2, 3]. Предоставление пользователю интерпретатора этого языка придает внутреннему ПО новое качество - микросистемы программирования.

3. Компоненты Open Firmware

Согласно стандарту, центральное звено всей системы ВПО - ***дерево устройств***. Корнем дерева считается центральный процессор, а узлами - устройства, шины и программные модули. Каждый узел содержит набор свойств, описывающих устройство, и методы доступа к устройству - процедуры тестирования, простые драйверы и служебные программы. Форматы свойств и методов не зависят от архитектуры центрального процессора. Базовая часть дерева устройств описывает

постоянно подключенные («впаянные») устройства и служебные программные модули. В процессе инициализации аппаратуры внутреннее ПО «зондирует» возможные места подключения дополнительных устройств и в случае обнаружения включает описывающие их узлы в дерево устройств. Если устройство выполнено с учетом требований стандарта IEEE 1275-1994 и содержит в своем ПЗУ программу (драйвер, тест, список свойств), то вся эта информация включается в описание устройства в дереве устройств. Если устройство не поддерживает данный стандарт, то все равно, как правило, имеется возможность определить его тип и базовые адреса.

Интерфейс к устройствам позволяет внутреннему ПО находить и опознавать подключенные устройства, добавляя их описание в дерево устройств. В самых общих чертах интерфейс к устройствам состоит из правил и методов «зондирования», то есть поиска устройств, и способа взаимодействия с найденными устройствами посредством FCode-программ, содержащихся в ПЗУ этих устройств. FCode - это байт-кодированное подмножество языка программирования Forth. FCode-программа считывается и выполняется в процессе «зондирования».

Клиентский интерфейс предназначен для обслуживания вызовов от операционной системы. Вызвать можно *любую* из имеющихся в Open Firmware функций, можно выполнить также целую фразу или несколько фраз на языке Forth. На практике, однако, операционной системе нужна лишь информация об аппаратной конфигурации компьютера, результаты тестов и иногда команды загрузки с того или иного устройства. Вся информация, предоставляемая операционной системе, черпается из дерева устройств. Стандарт предусматривает около трех десятков обязательных *сервисов* - функций, которые могут потребоваться операционной системе. Формат обращения к клиентскому интерфейсу не опирается на синтаксис языка Forth, это просто текстовое имя функции и массив аргументов/результатов.

Пользовательский интерфейс предназначен для интерактивного взаимодействия с пользователем через консольный терминал. Диалог с Open Firmware требуется в случае проверки/изменения параметров работы Open Firmware (включение-выключение «зондирования» конкретных слотов, тестов, указание источника загрузки операционной системы и пр.), интерактивной отладки программ, отладки подключенной аппаратуры, углубленного тестирования компьютера и т.д. Стандарт предусматривает использование интерпретатора системы Forth в качестве пользовательского интерфейса. При этом пользователю доступны все имеющиеся в Open Firmware и в языке Forth функции.

Тестовая система в Open Firmware состоит из тестов устройств, хранящихся в соответствующих узлах дерева устройств. Используются эти тесты при выполнении стартовой последовательности (инициализация-зондирование-тестирование-загрузка операционной системы). Иногда целесообразно добавление более тщательных и «придирчивых» тестов, особенно при отладке аппаратуры. В этом случае тестовая система может быть оформлена как автономная, со своим набором команд.

Forth - это язык программирования и система программирования. Основные свойства языка: вся обработка данных происходит в стеке, все команды обработки данных постфиксные. Конструкции управления - циклы и ветвления - близки к конструкциям языков высокого уровня. Синтаксис языка тривиален: единственное синтаксическое понятие в языке - это слово, составленное из произвольного набора

знаков. Важнейший элемент структурирования программы - процедура, определяемая как последовательность уже известных процедур. Вновь определенная процедура готова для выполнения сразу же после определения. Передача параметров осуществляется через стек, что не требует накладных расходов. Состав операций достаточно традиционен, необычны лишь команды манипулирования ячейками стека. Система программирования строится как интерпретатор сшитого кода (промежуточного представления программы в виде последовательности адресов вызываемых процедур). Для интерактивной работы служит интерпретатор входной строки, причем язык общения с системой совпадает с языком программирования. Главная составляющая системы программирования Forth - это словарь, содержащий имена и тела всех известных системе процедур. Словарь используется компилятором при обработке определения новой процедуры, интерпретатором входной строки при распознавании команд из входного потока. Из общих свойств Forth стоит отметить высокую мобильность, компактность кода, интерактивность и то, что новички в программировании осваивают его на пользовательском уровне за несколько часов. В то же время, профессиональным программистам, имеющим большой опыт в традиционных языках, Forth дается с трудом, из-за чего оппонентов у Forth гораздо больше, чем сторонников.

4. Критика

Стандарт Open Firmware оставляет противоречивое ощущение.

В стандарте заложена прогрессивная идея описания устройств компьютера в виде объектов со своими наборами свойств и методов. Эти объекты объединяются в дерево, представляющее всю конфигурацию компьютера в целом. Для работы с деревом служат три интерфейса: интерфейс к устройствам, с помощью которого создается и наращивается дерево, клиентский интерфейс, предназначенный для передачи информации операционной системе или прикладной программе, и пользовательский интерфейс, поддерживающий интерактивное взаимодействие с пользователем. По существу, наиболее важным из всех описанных в стандарте интерфейсов является интерфейс к устройствам, т.е. представленная в FCode программа, которая хранится в ПЗУ подключаемого устройства и считывается оттуда при зондировании. Таким образом, устройство содержит полную информацию о себе, включая описание свойств, методы доступа и тесты. Это наиболее критичный с точки зрения стабильности элемент системы внутреннего ПО, т.к. предполагает неизменность интерфейса к устройствам на протяжении всей жизни компьютера и такую же неизменность ответной части интерфейса в подключаемых устройствах. Модификации пользовательского интерфейса просто отражаются в инструкции, изменения в клиентском интерфейсе могут быть учтены при установке операционной системы, и только подключаемое устройство либо сопрягается с внутренним ПО, либо нет. Поэтому большая часть стандарта посвящена именно спецификации FCode, в которой подробно одна за другой описаны все команды. Описания достаточно формальны и зачастую не дают ясного представления о том, что за ними стоит на самом деле. Объяснить это стремлением авторов избежать машинно-зависимых деталей не удается, так как не хватает информации о взаимодействии достаточно высокоуровневых функций, описанных в самом стандарте. С этой точки зрения создание удовлетворяющего ему внутреннего ПО оказывается непростой задачей.

Как уже отмечалось, стандарт IEEE 1275-1994 закрепляет, немного обобщив, результаты работы группы программистов фирмы Sun, фиксируя то, что сделано, а не то, что в идеале хотелось бы получить. Вся работа базировалась на использовании языка Forth, поэтому стандарт получился полностью ориентированным на структуру, синтаксис и идеи этого языка. Сам Forth стал составной частью стандарта, а кроме того, чуть раньше в том же году был принят стандарт ANSI X3.215-1994 «Programming Languages - Forth» [3], разработанный при активном участии фирмы Sun. Ориентация данного стандарта на Forth подвергается серьезной критике [4]. Попробуем разобраться, что же на самом деле значит Forth для Open Firmware.

5. Место и роль языка Forth в Open Firmware

Насколько принципиально использование Forth в качестве среды и языка для создания внутреннего ПО? Попытаемся ответить на этот вопрос, проанализировав перечисленные в предыдущем разделе компоненты ВПО.

Представление *дерева устройств* именно в виде дерева отражает объективную реальность - аппаратная часть компьютера имеет древовидную структуру, где узлы соответствуют устройствам и шинам, а ребра описывают их взаимосвязи. Узел должен содержать список свойств устройства, список методов, т.е. процедур для работы с данным устройством, и список параметров, используемых при открытии узла. Такие узлы хорошо отображаются на словарь системы Forth: все списки имеют аналогичную структуру - это линейная последовательность элементов, каждый из которых имеет имя (название метода или свойства) и тело (код процедуры или значение свойства). Списки методов могут выстраиваться в цепочки, по которым стандартными процедурами Forth идет поиск. Формирование списков осуществляется также стандартными средствами, теми же, что и формирование словаря. Таким образом, Forth обеспечивает значительную часть программных готовых средств для работы с деревом устройств.

Интерфейс к устройствам, содержащим FCode-программы, это наиболее Forth-зависимая часть Open Firmware, т.к. FCode - это просто слегка упрощенный и отчасти специализированный вариант Forth, точнее, промежуточное представление Forth-программы в более компактном виде. Если устройства с FCode не используются, этот интерфейс никакой существенной связи с языком Forth не имеет. В то же время, Forth, а точнее его байт-кодированный вариант FCode, по многим факторам очень хорош для программирования устройств: весьма компактен; позволяет хранить не только данные, но и процедуры, которые после загрузки в дерево устройств немедленно готовы к выполнению; элементарен настолько, что применим ко всем мыслимым устройствам; полностью машинно-независим; однозначно (и обратимо) преобразуется в выполняемую Forth-программу простейшим конвертором.

С точки зрения реализации *клиентского интерфейса* специфические возможности Forth-системы нужны лишь для поиска в словаре указанных в клиентском запросе функций.

Для поддержки *пользовательского интерфейса* необходим механизм, обеспечивающий ввод команд и их аргументов (как числовых, так и текстовых), распознавание команд, т.е. поиск по словарю, их выполнение и выдачу результатов. Идеально для этой цели подходит интерпретатор входного потока системы Forth, изначально предоставляющий все необходимое. Несколько непривычным может пока-

заться задание сначала аргументов, а затем потребляющей их команды, но существенного значения это не имеет. В то же время, для отладки программ и аппаратуры возможности Forth весьма ценны - можно, например, задавать значения параметров в виде выражений, определять и тут же использовать процедуры, и пр. Язык Forth в качестве пользовательского интерфейса выглядит очень органично. Если учесть, что реально с пользовательским интерфейсом внутреннего ПО приходится работать крайне редко и очень немногим людям, то стиль интерфейса можно выбирать без оглядки на мнение большинства пользователей компьютеров.

Таким образом, использование Forth как базового языка программирования и среды исполнения ВПО вполне оправдано. Без преувеличения можно сказать, что Forth идеально подходит для этой задачи. Он обеспечивает адекватное отражение структуры и задач внутреннего ПО, позволяет создать мощную и компактную систему, мобильную и легко адаптируемую к различным условиям. Forth сочетает в одной системе программирования разнообразные полезные свойства, востребованные в реализации стандарта внутреннего ПО.

Однако столь безальтернативное употребление Forth вызывает и наибольшие возражения. Среди обязательных требований к внутреннему ПО - *простота сопровождения* на протяжении достаточно долгого времени. Это требование подразумевает, что внесение изменений и исправление обнаруженных ошибок должно быть легко осуществимо практически любым программистом и не должно начинаться с изучения скорее всего незнакомого ему языка Forth. Несмотря на то, что сам язык стандартизован, общепризнанных стандартных мобильных реализаций его нет, хотя большинство авторов утверждают, что придерживаются стандарта. Сомнения в «избранности» Forth для создания ВПО возникают у многих. В частности, существуют разработки внутреннего ПО, соответствующие стандарту, но предоставляющие возможность разработчикам устройств и компьютеров программировать на языке C [4]. Достигается это с помощью конвертора C Fo r t‡a далее используется обычная для Open Firmware схема.

6. Выводы

Приведенное краткое описание сути и задач внутреннего ПО показывает, что по многофункциональности, сложности и объему ВПО вполне сравнимо с операционными системами. Так, для работы OpenBoot на рабочей станции Sparc требуется не менее 2 мегабайт памяти, даже с учетом того, что язык программирования Forth дает очень компактный код. Тенденции в развитии hardware не оставляют сомнений в том, что внутреннее ПО будет усложняться и приобретать новые свойства и возможности, многие из которых хорошо просматриваются уже сейчас: все более полная унификация и стандартизация как пользовательских, так и аппаратных интерфейсов; повсеместное использование «интеллектуальной» периферии, способной общаться с центральным процессором и другими устройствами на «содержательном» языке (как, например, FCode в Open Firmware); дальнейшее развитие тестовых и диагностических функций; включение в ряде случаев в состав внутреннего ПО ядра ОС реального времени, особенно во встроенных и промышленных машинах, и так далее.

Нормальным явлением в такой ситуации является работа по стандартизации ВПО. То, что организаторами и участниками этой деятельности стали такие «ки-

ты», как Sun, Apple и IBM говорит и о серьезности проблемы, и о серьезности намерений. Содержание стандарта и его качество, конечно, не бесспорны, но это совершенно необходимый первый шаг в очевидно правильном направлении. В нынешнем, 2000, году стандарту исполняется 6 лет. За это время он не менялся, добавлялись лишь сопутствующие документы, в которых определяется порядок зондирования устройств на различных шинах. Любой стандарт начинает устаревать в тот момент, когда он принят, и происходит это тем быстрее, чем детальнее содержащаяся в нем информация. Весьма вероятно в скором времени появление нового стандарта и, соответственно, новых ВПО-продуктов, учитывающих нынешнюю ситуацию, тенденции и опыт применения существующего стандарта.

Литература

1. IEEE Std 1275-1994, Standard for boot (Initialization Configuration) Firmware: Core Requirements and Practices.
2. Brodie, Leo. A Threaded-Code Microprocessor Bursts Forth. Dr Dobb's Journal, October, 1985.
3. ANSI X3.215-1994. Programming Languages - Forth.
4. Child, Jeff. Open-Platform Boot Software Eases Development. Electronic Desin/ June 22, 1998, pp.48-54.

Рогов Е. В.

Технологии создания web-приложений на языке Java*

В статье предлагается ряд технологий, которые могут быть использованы при разработке систем клиент-сервер, предназначенных для функционирования в сети Интернет и обосновывается выбор для этих целей языка Java.

Необходимость в таких технологиях возникла в ходе проектирования и реализации системы анализа динамических процессов [1]. Это распределенная многопользовательская система с возможностью удаленного доступа через Интернет с помощью web-браузера. Она предоставляет пользователям спектр возможностей для хранения, обработки и отображения экспериментальных данных, проведения вычислительных экспериментов, моделирования нейросетей и генетических алгоритмов.

Основная часть этой системы написана на языке Java1. Ее исходные тексты послужили источником для примеров, приведенных в статье.

Оправдан ли выбор Java в качестве языка программирования для реализации подобного рода систем? Для ответа на этот вопрос сформулируем, что необходимо разработчикам от языка и его инструментального окружения, какие задачи им приходится решать.

Расширяемость и наращиваемость. При создании системы должна быть возможность использования уже существующих программных модулей и, кроме того, создаваемая система сама должна быть пригодной в будущем для повторного использования [4]. Уже созданная система в ряде случаев должна быть наращиваемой без модификации основного кода.

Надежность. Язык программирования должен способствовать созданию надежной системы, не допуская использование «опасных» программных приемов и обнаруживая потенциальные ошибки на возможно ранней стадии.

Безопасность. Система, работающая в открытой сети, должна обеспечивать безопасность в смысле защиты программного кода и данных от несанкционированного доступа.

Независимость от платформы. Разнообразие вычислительной техники, используемой в настоящее время, сильно затрудняет создание и поддержку отдельных версий для каждой программно-аппаратной платформы. В большинстве случаев значительно удобнее разработать одну программу, способную выполняться на любой из них.

Простота использования и поддержки. Система должна быть проста и удобна в обращении с точки зрения пользователя и в поддержке с точки зрения разработчиков.

---

* Работа выполнена при поддержке гранта РФФИ № 99-07-90229

1 За исключением вычислительных модулей, для которых критическим параметром является скорость счета.

Взаимодействие клиентской и серверной частей. Должен обеспечиваться удобный и эффективный механизм клиент-серверного взаимодействия, который предусматривает не только обмен данными, но и более высокоуровневые операции.

Взаимодействие с базами данных. При существующем многообразии СУБД крайне полезно иметь универсальный способ обращения к ним, особенно в случае одновременного использования нескольких СУБД или при необходимости перехода с одной СУБД на другую.

Мы покажем, что Java удовлетворяет всем перечисленным требованиям. Во многом это обеспечивается свойствами самого языка, отчего в ряде случаев на Java можно написать просто и красиво то, что в других языках потребовало бы громоздкого программирования с использованием специальных библиотек. Поэтому начало статьи посвящено рассмотрению собственно языка Java и его основных особенностей, а уже затем описываются технологии, позволяющие решать задачи, возникающие при создании распределенных систем.

Язык Java был создан в 1995 году фирмой Sun в рамках исследовательского проекта по разработке программного обеспечения для различных сетевых устройств и встроенных систем. Первоначально в качестве языка программирования был выбран C++, однако вскоре оказалось, что выгодней создать совершенно новый язык, чем преодолевать сложности, возникающие при использовании старого. Таким языком стал Java. И хотя свой синтаксис он в значительной степени унаследовал от C++, однако семантически существенно от него отличается. В частности, в Java отсутствуют неразумно сложные и потенциально опасные особенности C++, такие как множественное наследование, перегрузка операторов, указатели, препроцессор, оператор goto и т. д. Вместе с тем в язык добавлены или переведены из экзотики в разряд обычных вещей многие полезные свойства, такие как обработка *исключений* (exception), автоматический *сбор мусора* (garbage collection), *многопоточность* (multithreading).

Java является объектно-ориентированным языком программирования. Практически все сущности, с которыми имеет дело Java, являются *классами* (class). Механизм наследования позволяет строить новые классы, расширяя возможности уже существующих, а не создавая их заново. Совокупность классов, предоставляющих определенный сервис, образует *пакет* (package). В состав Java входит ряд пакетов для организации графического интерфейса, ввода-вывода, работы с сетью, взаимодействия с СУБД и др.

Java — язык надежного программирования. Масса потенциальных ошибок выявляется на стадии компиляции, а отсутствие указателей устраняет целый класс проблем при работе с памятью.

Большое внимание при разработке Java было уделено проблемам безопасности, поскольку язык предназначен для работы в сети.

Java — независимый от программно-аппаратной платформы язык. Это обеспечивается, во-первых, тем, что он является интерпретируемым языком. Программы на Java компилируются в т. н. байт-код, который затем выполняется виртуальной Java-машиной (JVM, Java Virtual Machine). Во-вторых, в спецификации языка отсутствует понятие «зависит от реализации» (например, по отношению к размеру числовых типов данных).

В языке Java осуществляется динамическая загрузка классов. Обычно это происходит автоматически при необходимости, однако, используя Reflection API, можно не только загружать классы из сети, но и динамически получать о них информацию. Если принять во внимание наличие сетевого пакета, то можно говорить о том, что языку присуща врожденная распределенность.

В результате, использование языка Java позволяет с минимальными затратами создавать надежные, переносимые и эффективные распределенные сетевые приложения. Однако, прежде чем перейти к техническим аспектам создания таких приложений, рассмотрим среду, под управлением которой работают Java-программы на стороне клиента и сервера.

Для клиентской части существуют два способа выполнения. Первый состоит в том, что программа является независимым приложением и исполняется непосредственно интерпретатором Java. Второй способ — создание *апплета* (applet), который предназначен для исполнения web-браузером.

В последнем случае, пользователь и разработчик получают весьма удобный способ работы. От пользователя не требуется никакой установки программного продукта (кроме, конечно, web-браузера). Ему надо лишь зайти на страницу, с которой произойдет загрузка апплета. Разработчик, создавая новую версию программы, просто выкладывает ее на web-сервер, и при следующем обращении пользователю будет загружен уже обновленный апплет. Пожалуй, единственным недостатком2 такого подхода являются ограничения, накладываемые на апплет из соображений безопасности. Например, апплету запрещено обращаться к локальному диску, для него ограничен сетевой доступ. Впрочем, эти ограничения могут быть сняты при использовании технологии *подписанных апплетов* (signed applets).

В случае серверной части используется специально разработанный интерфейс *сервлетов* (Servlet API).

Сервлеты — достаточно новая (по сравнению с CGI) технология создания серверных приложений. От CGI она отличаются преимущественно следующим. Во-первых, если CGI не накладывает ограничений на язык программирования, то сервлет по определению представляет собой программу на Java. Во-вторых, одновременные запросы к сервлету обрабатываются в разных потоках одного и того же объекта (multithreading). Это значительно эффективнее, чем, в случае CGI, порождение нового системного процесса, загрузка программы и, при необходимости, интерпретатора для нее, с последующим освобождением всех занимаемых ресурсов. В-третьих, при первом обращении к сервлету он загружается в Java-машину и остается активным до завершения работы web-сервера (persistence). Таким образом, сервлет может сохранять информацию между запросами, например, установить соединение с БД и использовать его, вместо того, чтобы соединяться с БД каждый раз заново. Кроме того, предусмотрен и механизм *сессий* (session), который позволяет отслеживать действия клиента во времени и объединять несколько сервлетов в единый программный комплекс.

Если используется Java версии 1.0 или 1.1, то для написания сервлетов необходимо дополнительно установить Java Servlet Development Kit, который предоставляет два пакета — javax.servlet и javax.servlet.http. В первом содержится интер-

2 Есть, впрочем, и другой — невысокое качество Java-машин, применяемых в современных браузерах. Остается надеяться, что в ближайшем будущем ситуация изменится к лучшему.

фейс Servlet, который должны реализовывать все сервлеты. В нем определяются три основных метода — init() для инициализации сервлета, service() для обслуживания запросов и destroy() для освобождения занимаемых ресурсов. Как правило, сервлеты наследуются от класса GenericServlet, представляющего собой «минимальный» сервлет, или от класса HttpServlet, описанного во втором пакете и предназначенного для работы с протоколом HTTP.

В случае http-сервлета, метод service() обрабатывает стандартные http-запросы, перенаправляя их соответствующим методам, например, при поступлении запроса GET для его обработки будет вызван метод doGet() и т. п. Методы-обработчики получают два параметра с интерфейсами HttpServletRequest и HttpServletResponse, которые служат для получения данных от клиента и возвращения результата.

Приведем пример простого сервлета, который служит для обработки данных, пришедших из html-формы по запросу GET или POST. Подобный подход можно применять при использовании сервлетов в качестве замены для CGI-программ. В этом примере сервлет возвращает html-страницу с перечислением всех переданных ему параметров. Для этого используются методы getParameterNames() и getParameterValues() интерфейса ServletRequest, возвращающие имена параметров и их значения. Сначала примерная html-страница с формой:

```
<html>
<head>
 <title>Servlet test</title>
</head>
<body>
 <form action="http://host.domain/servlets/Test" action=get>
  <input type=text name=field1>
  <input type=text name=field2>
  <input type=submit>
 </form>
</body>
</html>
```

И сам сервлет:

```
import java.io.*;
import java.net.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Test extends HttpServlet
{
 /** Обработка запросов POST
  */
 public void doPost(HttpServletRequest req, HttpServletResponse resp)
  throws ServletException, IOException
 { doService(req,resp);
 }
```

```
/** Обработка запросов GET
 */
public void doGet(HttpServletRequest req, HttpServletResponse resp)
  throws ServletException, IOException
{ doService(req.resp);
}

/** Собственно обработчик
 */
public void doService(HttpServletRequest req, HttpServletResponse resp)
  throws ServletException, IOException
{ resp.setContentType("text/html");
  PrintWriter out=resp.getWriter();
  out.println("<html><head><title>Result</title></head><body>");
  for(Enumeration params=req.getParameterNames();
      params.hasMoreElements();)
    { String name=(String)params.nextElement();
      String[] val=req.getParameterValues(name);
      out.println("<br>Parameter: "+name+", value(s):");
      for(int i=0;i<val.length;i++)
        out.println(val[i]+" ");
    }
  out.println("</body></html>");
  out.close();
 }
}
```

Рассмотрим теперь возможности, которые предоставляет Java для организации взаимодействия клиентской и серверной частей.

Разумеется, Java позволяет использовать стандартную передачу данных с применением механизма *сокетов* (socket), с установлением виртуального соединения или передачей дейтаграмм. Поддерживается и более высокоуровневый режим, при котором приложение имеет дело с *потоком данных* (data stream). Оставляя в стороне эти известные вещи, рассмотрим более интересный подход, при котором используется поток не данных, а объектов.

Такая возможность реализована в Java 1.1 и носит название *сериализации объектов* (object serialization). В пакете java.io определены два класса — ObjectOutputStream и ObjectInputStream. При передаче объекта методу writeObject() класса ObjectOutputStream происходит сериализация этого объекта, то есть запись в поток значений всех его переменных. При этом, если переменная ссылается на другой объект, то и он рекурсивно сериализуется. Таким образом, в поток попадает целая иерархия объектов.

При вызове метода readObject() класса ObjectInputStream происходит обратный процесс — чтение записанной иерархии. При этом объект восстанавливается в точно таком же состоянии, в котором он был сериализован.

Для того, чтобы указать, что объект допускает сериализацию, он должен реализовывать интерфейс Serializable. Те переменные, значения которых не имеет смысла сохранять, можно объявить как transient. Объекты, которым не подходит стандартный способ сериализации, могут реализовывать интерфейс Externalizable, который дает полный контроль над процессом записи и считывания.

В приведенном ниже классе ServerIO этот простой на вид, но весьма мощный по существу механизм используется для организации клиент-серверного взаимодействия. Предполагается, что запрос клиента представляет собой объект RequestObject, а ответ сервера — объект ResponseObject.

```
import java.io.*;
import java.net.*;

public class ServerIO
{ URLConnection uc=null;

 public ServerIO() {}

 /** Установление соединения с сервером
  */
 void Init() throws MalformedURLException,IOException
 { URL url=new URL("http://host.domain/servlets/Server");
  uc=url.openConnection();
 }

 /** Запись объекта в поток
  */
 void Write(RequestObject rq) throws IOException
 { uc.setDoOutput(true);
  uc.setUseCaches(false);
  uc.setRequestProperty("Content-type","application/octet-stream");
  ObjectOutputStream out=new ObjectOutputStream(uc.getOutputStream());
  out.writeObject(rq);
  out.flush();
  out.close();
 }

 /** Чтение объекта из потока
  */
 ResponseObject Read() throws IOException, ClassNotFoundException
 { ResponseObject rsp=null;
  ObjectInputStream in=new ObjectInputStream(uc.getInputStream());
  rsp=(ResponseObject)in.readObject();
  in.close();
  return rsp;
 }
```

```
 /** Устанавливает связь с сервером, посылает запрос и возвращает
  *  полученный ответ.
  */
 public ResponseObject Request(RequestObject rq)
 { Response rsp=null;

   try
   { Init();
     Write(rq);
     rsp=Read();
   }
   catch (Exception ex) {}

   return rsp;
 }
}
```

Теперь приведем пример соответствующего сервлета.

```
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Server extends HttpServlet
{
 /** Получает запрос от клиента, обрабатывает его
  *  и возвращает результат.
  */
 public void doPost(HttpServletRequest req, HttpServletResponse resp)
   throws ServletException, IOException
 { RequestObject rq=Read(req);
   ResponseObject rsp=serveRequest(rq);
   Write(resp,rsp);
   return;
 }

 /** Чтение объекта из потока
  */
 public RequestObject Read(HttpServletRequest req)
   throws IOException
 { RequestObject rq=null;
   ObjectInputStream in=new ObjectInputStream(req.getInputStream());

   try
   { rq=(RequestObject)in.readObject();
   }
   catch (ClassNotFoundException e) {}
```

```
  return rq;
 }

 /** запись объекта в поток
  */
 public void Write(HttpServletResponse resp, ResponseObject rsp)
  throws IOException
 { ObjectOutputStream out=new ObjectOutputStream(resp.getOutputStream());
  out.writeObject(rsp);
  out.flush();
  out.close();
 }

 /** Собственно обработка запроса
  */
 public ResponseObject serveRequest(RequestObject rq)
 { // ...
 }
}
```

Заметим, что для в ряде случаев может оказаться эффективным сжимать данные перед отправкой и восстанавливать их при получении, особенно при большом объеме передаваемой информации и низкой пропускной способности канала. Для этого достаточно входной и выходной поток пропустить через фильтр-архиватор, определенный в пакете java.util.zip, следующим образом:

```
ObjectInputStream in=new ObjectInputStream(
            new GZIPInputStream(req.getInputStream()));
```

и, соответственно,

```
ObjectOutputStream out=new ObjectOutputStream(
              new GZIPOutputStream(resp.getOutputStream()));
```

Рассмотрим теперь задачу, связанную с наращиваемостью приложения. Допустим, требуется предоставлять пользователю ряд возможностей, каждая из которых реализуется отдельным классом. Набор таких классов составляет функциональное наполнение приложения. Хотелось бы сделать так, чтобы добавление новых классов было возможно без модификации основного программного кода. В отличие от традиционных языков со статическим связыванием модулей, динамическая загрузка классов в Java позволяет легко решить эту проблему, используя механизм *рефлексии* (reflection).

Reflection API состоит из класса java.lang.Class, позволяющего заглянуть внутрь любого класса языка Java, и пакета java.lang.reflect. В нем описаны классы Field для доступа к переменным класса, Method для вызова методов класса и Constructor для создания новых объектов.

Применение Reflection API рассмотрим на следующей задаче. В состав серверной части приложения входит ряд вычислительных модулей, каждый из которых требует определенного набора входных параметров, задаваемых пользователем. Клиентская часть должна вызывать диалоговое окно, соответствующее вы-

бранному пользователем модулю, в котором запрашиваются все необходимые параметры.

Диалоговые окна создаются классами, реализующими следующий интерфейс:

```
public interface Dialogue
{ public Boolean show();
  public Parameters getParameters();
}
```

Метод show() служит для отображения диалога и возвращает true, если пользователь подтвердил введенные параметры, или false, если отказался от них. Метод getParameters() возвращает параметры в виде объекта типа Parameters.

Приведем соответствующий метод клиентской программы, который по имени класса вызывает диалог и возвращает введенные параметры или null в случае любой ошибки:

```
public Parameters retrieveParameters(String className)
{ Parameters param=null;

 try
 { // динамическая загрузка класса
  Class c=Class.forName(className);

  // получение возможных конструкторов
  Constructor[] con=c.getConstructors();

  // создание экземпляра класса
  Object[] args={};
  Object o=con[0].newInstance(args);

  // отображение диалога
  Method show=c.getMethod("show",null);
  Boolean res=(Boolean)show.invoke(o,null);
  if (res.booleanValue())
  { // получение параметров
   Method getparam=c.getMethod("getParameters",null);
   param=(Parameters)getparam.invoke(o,null);
  }
 }
 catch (ClassNotFoundException e)    {} // Не найден класс
 catch (NoSuchMethodException e)     {} // Не найден метод
 catch (IllegalAccessException e)    {} // Класс или метод недоступен
 catch (InstantiationException e)    {} // Создание экземпляра абстрактного
                                        // класса или интерфейса
 catch (InvocationTargetException e) {} // Исключение в вызываемом
                                        // конструкторе или методе
 return param;
}
```

Java 1.1 развивает этот подход в JavaBeans API, позволяющий создавать т. н. Bean-объекты — модули многократного использования. Программы, задействующие Bean-объекты, могут устанавливать и использовать свойства, методы и события, определенные в них.

В заключение рассмотрим одну из распространенных и важных задач сервера — организацию работы с базой данных. Java предоставляет для этой цели специальный программный интерфейс JDBC (хотя JDBC и не является аббревиатурой, под этим словом обычно понимается «Java Database Connectivity»), предоставляемый в пакете java.sql. Этот интерфейс позволяет обращаться совершенно одинаковым образом практически к любым реляционным СУБД, использующим язык SQL.

Сложность состоит в том, что производители СУБД обычно в той или иной мере отходят от стандарта SQL в своих реализациях, предоставляя свои специфичные расширения. Поэтому JDBC позволяет, во-первых, передавать драйверу СУБД запрос «как есть», что делает возможным использовать любую СУБД за счет потери универсальности. Во-вторых, допускается использование стандартных *управляющих последовательностей* (escape clause) для ряда часто встречающихся задач. И в-третьих, JDBC позволяет получать метаинформацию о СУБД, ее требованиях и возможностях.

Для начала работы с базой данных требуется загрузить соответствующий драйвер и установить соединение с СУБД. Для идентификации СУБД используется URL, где устанавливается протокол «jdbc», нужный подпротокол и указывается база данных. Полный вид URL зависит от используемой СУБД и драйвера. Например, для установления соединения с базой данных «db» под управлением Postgres 95 на машине «host.domain» под пользователем «username» с паролем «password» может применяться следующий фрагмент кода:

```
Class.forName("postgresql.Driver");
Connection con=DriverManager.getConnection(
  "jdbc:postgresql://host.domain/db","username","password");
```

JDBC предоставляет возможность использовать три разные формы запросов — обычные (Statement), прекомпилированные (PreparedStatement) и вызовы хранимых процедур (CallableStatement). Прекомпилированные запросы допускают параметризацию и используются для повышения эффективности в случае их неоднократного применения. Вызовы хранимых процедур вынесены в отдельную форму запроса для того, чтобы предоставить единый способ обращения к хранимым процедурам для всех СУБД.

Запросы, результатом которых является таблица, выполняются с помощью метода executeQuery(), возвращающего объект типа ResultSet. Такие запросы, как INSERT, UPDATE, DELETE, выполняются с помощью метода executeUpdate().

Приведем пример метода, который служит для идентификации пользователей. Информация о зарегистрированных пользователях хранится в базе данных в таблице «users». Метод возвращает true, если пароли совпадают, и false, если указанного пользователя нет в таблице или пароль указан неверно.

```
public boolean checkAuthority(String username, String password)
{ Connection con=null;
  boolean check=false;
```

```
try
{ Class.forName("postgresql.Driver");
  con=DriverManager.getConnection(
    "jdbc:postgresql://host.domain/db","username","password");

  Statement st=con.createStatement();
  ResultSet rs=st.executeQuery(
    "SELECT password FROM users WHERE username='"+username+"'");
  if (rs.next())
  { String pw=rs.getString(1);
    check=(password.equals(pw));
  }
}
catch (SQLException e) {}          // ошибка в запросе
catch (ClassNotFoundException e) {} // не найден драйвер

  return check;
}
```

Все описанные технологии создания приложений на языке Java применялись при разработке системы анализа динамических процессов.

В текущей реализации клиентская часть представляет собой апплет, выполняющийся любым современным браузером, серверная часть — сервлет, работающий под управлением web-сервера Apache с модулем jserv. Клиент-серверный обмен происходит с помощью объектных потоков, доступ к СУБД (Postgres 95) осуществляется через интерфейс JDBC. Вычислительные модули используют интерфейс CGI, для ввода пользователем параметров используется Reflection API.

Литература.

1. **Л. Н. Королев, А. М. Попов, Н. Н. Попова, Е. В. Рогов.**
2. Реализация системы анализа динамических процессов в сети Интернет. — В кн.: Научный сервис в сети Интернет: Тезисы докладов Всероссийской научной конференции (20—25 сентября 1999 г., г. Новороссийск). — М.: Изд-во МГУ, 1999, с. 162—167.
3. **Дэвид Флэнаген.** Java in a Nutshell: Пер. с англ. — К.: Издательская группа BHV, 1998.
4. **JDBC Database Access from Java:** A Tutorial and Annotated Reference. — Addison-Wesley Publishing Company, 1997.
5. **Г. Буч.** Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд./Пер. с англ. — М.: «Издательство Бином», СПб.: «Невский диалект», 1999 г.

Аннотация

Королев Л.Н., Смелянский Р.Л. Проблемы преподавания программирования в классическом университете. // Программные системы и инструменты. Тематический сборник № 1, М.: Изд-во факультета ВМиК МГУ, 2000.

Рассматриваются проблемы преподавания цикла программистских дисциплин в классическом университете. Намечаются пути решения данных проблем.

Ил.: нет. Библиогр.: нет.

Брусенцов Н. П. Блуждание в трех соснах (Приключения диалектики в информатике). // Программные системы и инструменты. Тематический сборник № 1, М.: Изд-во факультета ВМиК МГУ, 2000.

Цель памфлета - обратить внимание на недиалектичность и несподручность здравому рассуждению преобладающей в современной информатике формальной логики, унаследованные ею будто бы от Аристотеля. Указана возможность и актуальная необходимость воссоздать в информатике подлинный дух аристотелева Органона.

Табл.: 2. Библиогр.: 13 назв.

Доложено на Ломоносовских чтениях 24 апреля 2000 г. на факультете ВМиК МГУ.

Смелянский Р.Л., Чистолинов М.В., Бахмуров А.Г., Захаров В.А. О Международном проекте в области проверки правильности программного обеспечения встроенных систем. // Программные системы и инструменты. Тематический сборник № 1, М.: Изд-во факультета ВМиК МГУ, 2000.

В статье рассмотрены опыт и итоги проекта по сравнительному анализу и интеграции средств проектирования и спецификации распределеных систем (проект EC INCO-Copernicus № 977 020). Предложена схема интеграции средств, созданных участниками проекта. Рассматривается концепция комплексного подхода к проектированию встроенных систем.

Ил.: 1. Библ.: 14 назв.

Костенко В. А. Задачи синтеза архитектур: формализация, особенности и возможности различных методов для их решения. // Москва, 2000. // Программные системы и инструменты. Тематический сборник № 1, М.: Изд-во факультета ВМиК МГУ, 2000.

Рассматривается проблема синтеза архитектур вычислительных систем под заданное приложение: формальные модели описания функционирования вычислительных систем, позволяющие формализовать задачу синтеза архитектур как экстремальную многопараметрическую задачу с ограничениями, проводится анализ возможностей использования существующих методов решения, рассмотрен ряд методов решения задач синтеза архитектур, учитывающих особенности рассматриваемой предметной области, и приведены результаты их исследования.

Ил.: 1. Библиогр.: 9 назв.

Чистолинов М.В., Бахмуров А.Г. Среда моделирования многопроцессорных вычислительных систем. // Программные системы и инструменты. Тематический сборник № 1, М.: Изд-во факультета ВМиК МГУ, 2000.

Рассмотрены особенности методов анализа функционирования многопроцессорных вычислительных систем, архитектура среды для моделирования таких систем, опыт ее применения и преспективы развития.

Ил.: 1. Библиогр.: 8 назв.

Афанасьев М.К., Дмитриев П.А. Подходы к исследованию поведения генетических алгоритмов с использованием конструктора NS Galaxy. // Программные системы и инструменты. Тематический сборник № 1, М.: Изд-во факультета ВМиК МГУ, 2000.

Описаны основные характеристики конструктора генетических алгоритмов NS Galaxy, способы задания параметров и интерфейс с пользователем.

Ил.: 8. Библиогр.: 12.

Буряк Д.Ю. Особенности применения нейронных сетей к задаче распознования символов. // Программные системы и инструменты. Тематический сборник № 1, М.: Изд-во факультета ВМиК МГУ, 2000.

Приведены результаты сравнительного анализа применения нейронных сетей и методов классификации по критерию минимального расстояния к задаче распознавания символов. Рассмотрены " быстрые" нейронные сети и сети с отброшенными связями.

Ил.: 8. Библиогр.: 7.

Царьков Д.В. Использование модульности при верификации распределенных программ// Москва, 2000. // Программные системы и инструменты. Тематический сборник № 1, М.: Изд-во факультета ВМиК МГУ, 2000.

Рассматривается задача формальной верификации распределенных программ. Приводится алгоритм использования модульной структуры программы для упрощения процесса верификации. Предлагаются механизмы модульной редукции программы. Доказывается корректность приведенных алгоритмов.

Библиогр.: 7 назв.

Рамиль Альварес Х. Статистический анализ результатов контрольных работ и тестов в МСО "Наставник". // Программные системы и инструменты. Тематический сборник № 1, М.: Изд-во факультета ВМиК МГУ, 2000.

Рассмотрен 25-летний опыт работы системы " Наставник", используемый на факультете ВМиК МГУ для тестирования и проведения контрольных работ.

Ил.: 3. Библиогр.: 4.

Сидоров С.А. Стандарт Open Firmware - критический анализ. // Программные системы и инструменты. Тематический сборник № 1, М.: Изд-во факультета ВМиК МГУ, 2000.

Рассматривается стандарт на внутреннее программное обеспечение (ПО) «IEEE Std 1275-1994, Standard for boot (Initialization Configuration) Firmware: Core Requirements and Practices», сокращенно называемый «Open Firmware». Вводятся основные понятия, описываются функции внутреннего ПО, главные компоненты системы внутреннего ПО. Анализируются содержание и слабые стороны стандарта.

Доложено на Ломоносовских чтениях 24 апреля 2000 г. на факультете ВМиК МГУ

Рогов Е.В. Технологии создания web-приложений на языке Java. // Программные системы и инструменты. Тематический сборник № 1, М.: Изд-во факультета ВМиК МГУ, 2000.

Предлагаются технологии разработки систем клиент-сервер в сети Internet. Приведены типовые программные компоненты на языке Java.

Библиогр.: 5.

Шчепанович С., Леонтьев Д.В., Мратов А.А. Вопросы определения калибровочных параметров имитационных моделей серверных приложений // Москва, 2000. // Программные системы и инструменты. Тематический сборник № 1, М.: Изд-во факультета ВМиК МГУ, 2000.

Рассматриваются общие принципы калибровки имитационных моделей сетевых компонентов и предлагается методика калибровки моделей файл-сервера и сервера базы данных.

Ил.: 2. Библиогр.: 6 назв.

Вознесенская Т. В. Математическая модель алгоритмов синхронизации времени для распределенного имитационного моделирования// Москва, 2000. // Программные системы и инструменты. Тематический сборник № 1, М.: Изд-во факультета ВМиК МГУ, 2000.

Рассмотрена проблема синхронизации времени в распределенном имитационном моделировании. Построены математические модели функционирования приложения (имитационной модели) под управлением строго синхронного, консервативного и оптимистических алгоритмов синхронизации (АС) модельного времени. Разработан метод сравнения различных АС на основе этих моделей.

Илл.: 1 рис., 1 табл. Библиогр.: 4 назв.

Новиков М.Д., Павлов Б.М. Программный инструментарий идентификации динамических режимов нелинейных систем уравнений. // Программные системы и инструменты. Тематический сборник № 1, М.: Изд-во факультета ВМиК МГУ, 2000.

Описывается структура и возможности интерактивного пакета прикладных программ "Нелинейные динамические системы".

Ил.: 4. Библиогр.: 6 назв.

Рогов Е.В. Интернет-организация. Системы Анализа Динамических Процессов. // Программные системы и инструменты. Тематический сборник № 1, М.: Изд-во факультета ВМиК МГУ, 2000.

Описывается архитектура Системы Анализа Динамических процессов, предназначенная для анализа многопараметрических временных рядов.

Ил.: 2. Библиогр. : 2 назв.

Эдельман Л.В. Система объектно-ориентированных баз данных на основе технологии COM. // Программные системы и инструменты. Тематический сборник № 1, М.: Изд-во факультета ВМиК МГУ, 2000.

Предложен вариант реализации объектно-ориентированных баз данных на основе технологии COM, который позволит решить ряд проблем СООБД.

Ил.: нет. Библиогр.: 10 назв.

Маркин М.И. Об одном методе повышения эффективности обучения нейронной сети прямого распространения// Москва, 2000. // Программные системы и инструменты. Тематический сборник № 1, М.: Изд-во факультета ВМиК МГУ, 2000.

Работа посвящена задаче выбора начальных значений весов нейронной сети при ее обучении с помощью методов локальной оптимизации. Предлагаемый алгоритм позволяет выбирать "хорошие" начальные приближения, что значительно сокращает время работы алгоритма локальной оптимизации и позволяет отсекать начальные приближения, находящиеся в зоне притяжения неудовлетворительных локальных минимумов. Результаты экспериментов подтверждают эффективность предлагаемого алгоритма.

Библиогр.: 5 назв.