Достижения в области параллельного программирования

Бертран Мейер

Ершовская Лекция

Новосибирск

11 апреля 2013 г.

Eiffel Software

ETH Chair of Software Engineering

ИТМО Санкт-Петербург

Андрей Петрович Ершов

Язык

english

Институт систем информатики
им. А.П. Ершова
СО РАН

Новости | Институт | Обучение | Проекты | Публикации | Конференции | Библиотека | Контакты

VII лекция по информатике

VII лекция, 2013
VI лекция, 2012
V лекция, 2011
IV лекция, 2010
III лекция, 2009
II лекция, 2008
I лекция, 2007

Если вы заметили ошибку или неработающую ссылку, пожалуйста, выделите текст мышью и нажмите Ctrl + Enter

По традиции, заложенной Андре… Андрея Петровича Ершова (19 апреля) наш… …м информатики СО РАН выступление известных учё… …иалистов в области информатики, программирования и вычислитель… …ки. С историей и записями предыдущих Ершовских лекций можно позна… …я на сайте .

В этом году наш Фонд при… …выступить с VII Ершовской лекцией известного специалиста в области п… …ммной инженерии Бертрана Мейера. В настоящее время профессор Мей… …зглавляет Chair of Software Engineering в Цюрихе, является Chief Technology Off… …компании Eiffel Software, а также возглавляет Кафедру программной инженерии и верификации про… …мм в Санкт-Петербургском национальном исследовательском университете ИТМО.

«…владеет русским языком…»

Профессор Б. Мейер владеет русским языком и имеет давние дружественные связи с российским программистским сообществом. Начинающим ученым он был приглашен А.П. Ершовым на стажировку в новосибирский Академгородок в 1977 г. А в 2003 г. Бертран Мейер в качестве приглашенного докладчика выступал в Новосибирске на пятой международной конференции, посвященной памяти А.П. Ершова Perspectives of System Informatics.

Ах...

говорит!

Наша лаборатория в ИТМО

Software Engineering Laboratory

(Лаборатория программной инженерии и верификации)

Часть Факультета информационных технологий и программирования

sel.ifmo.ru

Создана в 2011 г.

«Мегагрант» компании Mail.ru Group

О чем речь идет

SCOOP: модель программирования (расширение Eiffel), позволяющая контролировать параллельность

План этого доклада:

- 1. Почему параллельность это трудно
- 2. SCOOP: неформальная идея («трейлер»)
- 3. SCOOP модель: последовательные ограничения
- [4. Система типов SCOOP]
- 5. Открытые проблемы и текущая работа

SCOOP

Simple Concurrent Object-Oriented Programming

Цели и принципы:

- **Простота**
- Позволить «обычным» людям писать **надежные** параллельные программы
- Сохранить последовательные способы **рассуждения** (только одно новое ключевое слово)
- Освободить программистов от **рисков** параллелизма, особенно от состояния гонки (data race) и блокировки (deadlock)

SCOOP

Гонки нет

SCOOP

Гонки нет

SCOOP

Гонки нет

SCOOP

Гонки нет

SCOOP

Гонки нет

SCOOP

Гонки нет

SCOOP

Гонки нет

SCOOP

Гонки нет

Гонки нет

Гонки нет

Гонки нет

Гонки нет

Гонки нет

Гонки нет

Гонки нет

Гонки нет

Гонки нет

Гонки нет

Гонки нет

Гонки нет

Гонки нет

Гонки нет

Развитие SCOOP

Базовые идеи – из 90-х годов

Первое описание в книге «Объектно-ориентированное конструирование программных систем», 1997 (русский перевод, 2005)

Диссертация Пиотра Ниенальтовского, ETH, 2008

«Мегагрант» ERC (European Research Council), 2012-2017, 2.5 M €; цель: вывести параллельную технологию на следующий уровень

Здесь представлена работа многих людей

Eiffel Software: Emmanuel Stapf, Александр Когтенков, Ian King

ETH: Piotr Nienaltowski, Benjamin Morandi, Sebastian Nanz, Scott West

ITMO: Александр Когтенков

- 1 -

Параллельное программирование - трудно?

Параллельность: зачем?

1. Эффективность

Конец закона Мура

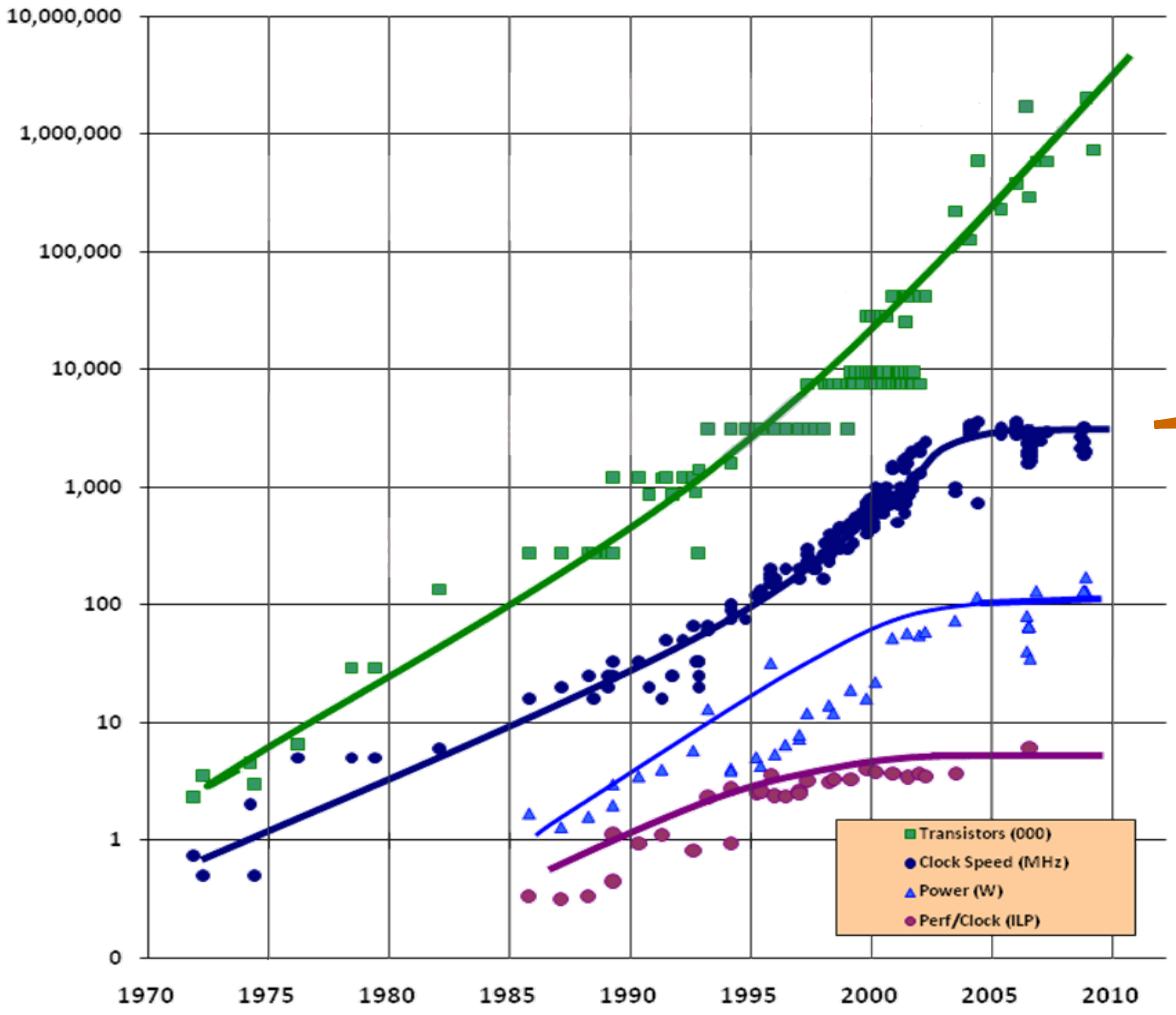
2. Удобство

Многопоточность (параллельность в рамке одной программы)

3. Моделирование

Мир – параллельный! (сети, встраиваемые системы, робототехника...)

Эффективность



Число транзисторов

Тактовая частота

Источник: Intel

New York Times, декабрь 2007 г.

The New York Times

Technology

WORLD | U.S. | N.Y. / REGION | BUSINESS | TECHNOLOGY | SCIENCE | HEALTH | SPORTS | OPINION

Search Tech News & 8,000+ Products Go

Browse Products -- Select a Product Category -- Go

Faster Chips Are Leaving Programmers in Their Dust

By JOHN MARKOFF
Published: December 17, 2007

REDMOND, Wash. — When he was chief executive of Intel in the 1990s, Andrew S. Grove would often talk about the "software spiral" — the interplay between ever-faster microprocessor chips and software that required ever more computing power.

Kevin P. Casey for The New York Times
At Microsoft, from top, Craig Mundie, Burton Smith and Dan Reed are working on the next generation of computing power.

The potential speed of chips is still climbing, but now the software they run is having trouble keeping up. Newer chips with multiple processors require dauntingly complex software that breaks up computing chores into chunks that can be processed at the same time.

The challenges have not dented the enthusiasm for the potential of the new parallel chips at Microsoft, where executives are betting that the arrival of manycore chips — processors with more than eight cores, possible as soon as 2010 — will transform the world of personal computing.

The company is mounting a major effort to improve the parallel computing capabilities in its software.

"Microsoft is doing the right thing in trying to develop parallel software," said Andrew Singer, a veteran software designer who is the co-founder of Rapport Inc., a parallel computing company based in Redwood City, Calif. "They could be roadkill if somebody else figures out how to do this first."

«Новые процессоры с несколькими ядрами требуют чрезвычайно сложного программного обеспечения»

Что говорят о параллельном программировании

Интел, 2006:

- *Многоядерные вычисления быстрым и захватывающим путем переводят индустрию на абсолютно новую территорию*

Рик Рашид, глава Microsoft Research, 2007:

- *Многоядерные процессоры представляют собой одну из крупнейших смен технологии, с глубокими следствиями в методах разработки программ*

Билл Гейтс:

- *Мы никогда не сталкивались с подобными задачами.* **Здесь нужен прорыв**.

Дэвид Паттерсон, Калифорнийский университет в Беркли, 2007:

- *Вся индустрия, в принципе, сделала отчаянный выбор. Она делает ставку на параллельные вычисления. Ставка сделана, но большая проблема - добиться выигрыша*

Что говорят о параллельном программировании

Национальная академия наук США (2011):

Героические программисты используют высокую степень параллелизма...

Однако ни одна из теперешних разработок близко не подходит к повсеместной поддержке программирования параллельного оборудования, которая требуется, чтобы убедиться, что влияние информационных технологий на общество в течение следующих двадцати лет будет настолько же ошеломительным, как и в последние полвека

Программирование: тогда и теперь

Последовательное программирование:

Бывало беспорядочным

Все еще трудно, но:

- Структурное программирование
- Абстракция данных & объектная технология
- Проектирование по Контракту
- Архитектурные шаблоны

Параллельное программирование:

Бывало беспорядочным

Таковым остается

Пример: потоковые модели в наиболее популярных подходах

Уровень разработки:

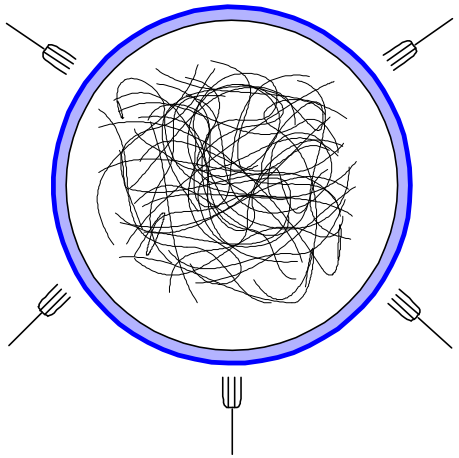
60-е/70-е годы

Операционное обоснование

Традиционная программа с семафорами

Listing 4.34: Tanenbaum's solution

```
1  def get_fork(i):
2      mutex.wait()
3      state[i] = 'hungry'
4      test(i)
5      mutex.signal()
6      sem[i].wait()
7
8  def put_fork(i):
9      mutex.wait()
10     state[i] = 'thinking'
11     test(right(i))
12     test(left(i))
13     mutex.signal()
14
15 def test(i):
16     if state[i] == 'hungry' and
17     state (left (i)) != 'eating' and
18     state (right (i)) != 'eating':
19         state[i] = 'eating'
20         sem[i].signal()
```



Операционное обоснование

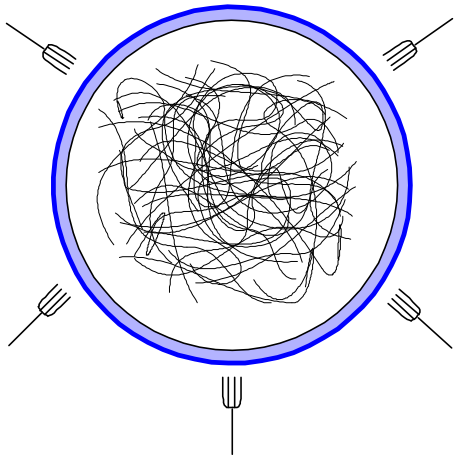
Пример: Goto

```
int main () {
    int a = 100;
    LOOP:do      {
                  if (a == 10) {
                        a = a + 1;
                        goto LOOP;}
                 printf("value of a: %d\n", a);
                 a++;
                 } while (a < 30 );
  return 0;}
```

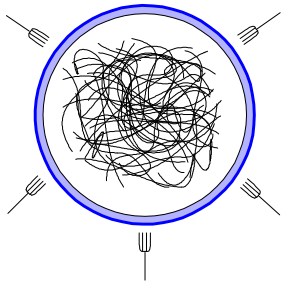
Традиционная программа с семафорами

Listing 4.34: Tanenbaum's solution

```
1  def get_fork(i):
2      mutex.wait()
3      state[i] = 'hungry'
4      test(i)
5      mutex.signal()
6      sem[i].wait()
7
8  def put_fork(i):
9      mutex.wait()
10     state[i] = 'thinking'
11     test(right(i))
12     test(left(i))
13     mutex.signal()
14
15 def test(i):
16     if state[i] == 'hungry' and
17     state (left (i)) != 'eating' and
18     state (right (i)) != 'eating':
19         state[i] = 'eating'
20         sem[i].signal()
```



Программирование на SCOOP



```
class ФИЛОСОФ feature
        левая, правая: ВИЛКА
        вставать do … end
        закончено : BOOLEAN

        жить
                do
                        from вставать until закончено loop
                                думать; обедать (левая, правая)
                        end
                end

        обедать ( л, п : separate ВИЛКА )
                        -- Есть, при наличии вилок л и п.
                do л.взять; п.взять ; … end
end
```

Почему так трудно?

```
transfer (source, target:                    ACCOUNT;
          amount: INTEGER)
        -- If enough funds, transfer amount from source to target.
    do
        if source.balance >= amount then
            source.withdraw  (amount)
            target.deposit   (amount)
        end
    end
```

Рассуждать на базе APIs

```
transfer (source, target:                    ACCOUNT;
          amount: INTEGER)
        -- Transfer amount from source to target.
    require
         source.balance >= amount
    do
        source.withdraw  (amount)
        target.deposit    (amount)
    ensure
        source.balance = old source.balance - amount
        target.balance = old target.balance + amount
    end
```

```
invariant
        balance >= 0
```

Рассуждать на базе APIs

```
transfer (source, target:                    ACCOUNT;
          amount: INTEGER)
        -- Transfer amount from source to target.
    require
        source.balance >= amount
    do
        source.withdraw  (amount)
        target.deposit   (amount)
    ensure
        source.balance = old source.balance - amount
        target.balance = old target.balance + amount
    end
```

Рассуждать о банковских счетах

```
if acc1.balance >= 100 then transfer (acc1, acc2, 100) end
if acc1.balance >= 100 then transfer (acc1, acc3, 100) end
```

```
invariant
        balance >= 0
```

```
transfer (source
            amo
        -- Trans
    require
        source.
    do
        source.w
        target.d
    ensure
        source.b
        target.balance = old target.balance + amount
    end
```

Дейкстра 1968*: *Goto considered harmful*

Наш разум приспособлен к представлению статических отношений, а не процессов, развивающихся во времени

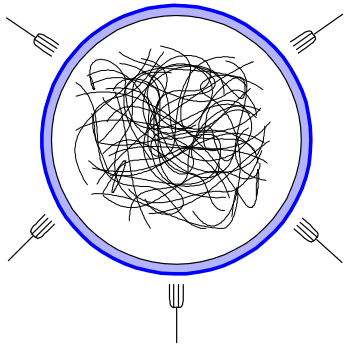
Поэтому мы должны стремиться сократить концептуальный разрыв между статическими программами (в пространстве текста) и динамическими процессами (во времени)

**Сокращено*

Параллельное программирование сегодня

Listing 4.34: Tanenbaum's solution

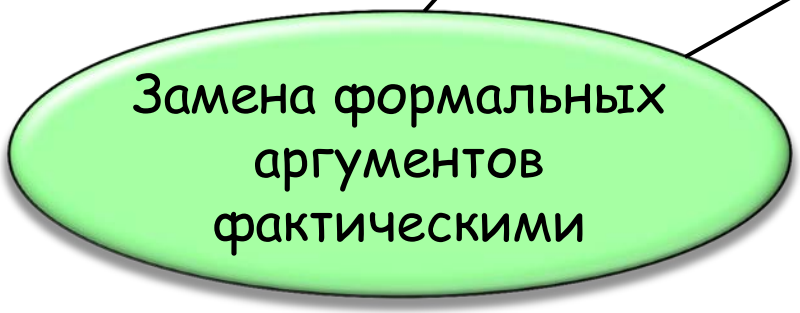
```
1  def get_fork(i):
2      mutex.wait()
3      state[i] - 'hungry'
4      test(i)
5      mutex.signal()
6      sem[i].wait()
7
8  def put_fork(i):
9      mutex.wait()
10     state[i] - 'thinking'
11     test(right(i))
12     test(left(i))
13     mutex.signal()
14
15 def test(i):
16     if state[i] -- 'hungry' and
17     state (left (i)) != 'eating' and
18     state (right (i)) != 'eating':
19         state[i] - 'eating'
20         sem[i].signal()
```



Формально: последовательное рассуждение

Если n экспортированных методов:
только n доказательств

$$\frac{\{\text{INV and } Pre_r\}\quad body_r\quad \{\text{INV and } Post_r\}}{\{Pre_r'\}\quad x.r\ (a)\quad \{Post_r'\}}$$



Рассуждение об объектах в параллельном контексте

Если n экспортированных методов: только n доказательств?

Параллельная версия этого правила – дальше!

$$\frac{\{\text{INV and } Pre_r\}\quad body_r\quad \{\text{INV and } Post_r\}}{\{Pre_r'\}\quad x.r\ (a)\quad \{Post_r'\}}$$

Обычные (и неправильные) предположения

"Параллельность – базовый случай, последовательное программирование – специальный случай"

- В принципе правильно, но мы **наилучше понимаем последовательное программирование**
- В реалистических системах, **параллельность – только одна часть программы**

"Объекты параллельны по самой своей природе" (Мильнер)

- Большинство попыток основаны на (внутренне противоречивом) понятии "Активных объектов"
- Часто ведут к "Аномалии наследования"

Может ли помочь объектная технология?

"*Объекты параллельны по самой своей природе*" (Milner)

Большинство попыток основаны на (внутренне противоречивом) понятии "Активных объектов"

Часто ведут к "Аномалии наследования"

Не являются широко принятыми

На практике: низкоуровневые механизмы, надстроенные над О-О языком

Подход

1. Прибавить параллельность **хорошо понятой последовательной модели**

2. Позволить программистам продолжать использование **техник рассуждения**, которые успешно применяются в последовательном программировании

3. Перенести трудности **в реализацию** модели (компилятор и scheduler)

Отправная точка

Объектно-ориентированная технология

- Абстракция данных
- Сокрытие информации
- Наследование
- Применяется обработке больших программ
- Widely used

Мы основываемся на варианте Eiffel:

- Проектирование по контрактам, так как мы можем **рассуждать** о программах

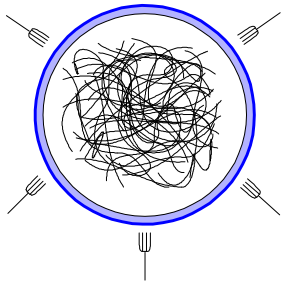
Фундаментальный вопрос

Можем ли мы привести параллельное программирование к такому же уровню **абстракции** и **удобства** как последовательное программирование?

- 2 -

SCOOP: трейлер

Пример 1: обедающие философы

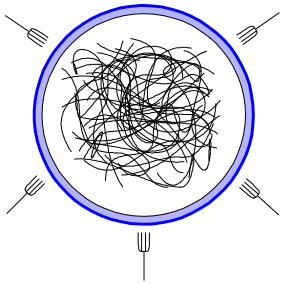


```
class ФИЛОСОФ feature
        левая, правая: ВИЛКА
        вставать do ... end
        закончено : BOOLEAN

        жить
                do
                        from вставать until закончено loop
                                думать;  обедать (левая, правая)
                        end
                end

        обедать ( л, п : separate ВИЛКА )
                        -- Есть, при наличии вилок л и п.
                do л.взять; п.взять ; ... end
end
```

С семафорами...

Listing 4.34: Tanenbaum's solution

```
1  def get_fork(i):
2      mutex.wait()
3      state[i] - 'hungry'
4      test(i)
5      mutex.signal()
6      sem[i].wait()
7
8  def put_fork(i):
9      mutex.wait()
10     state[i] - 'thinking'
11     test(right(i))
12     test(left(i))
13     mutex.signal()
14
15 def test(i):
16     if state[i] -- 'hungry' and
17     state (left (i)) != 'eating' and
18     state (right (i)) != 'eating':
19         state[i] - 'eating'
20         sem[i].signal()
```



Пример 2: перевод денег

```
if acc1.balance >= 100 then transfer (acc1, acc2, 100) end
if acc1.balance >= 100 then transfer (acc1, acc3, 100) end
```

```
invariant
        balance >= 0
```

```
transfer (source
            amo
        -- Trans
    require
        source.
    do
        source.w
        target.d
    ensure
        source.b
        target.balance = old target.balance + amount
    end
```

Перевод денег в SCOOP

```
transfer (source, target:                    ACCOUNT;
          amount: INTEGER)
        -- Transfer amount from source to target.
    require
         source.balance >= amount
    do
        source.withdraw  (amount)
        target.deposit    (amount)
    ensure
        source.balance = old source.balance – amount
        target.balance = old target.balance + amount
    end
```

separate

Пример 3: Рассуждать с APIs: очередь

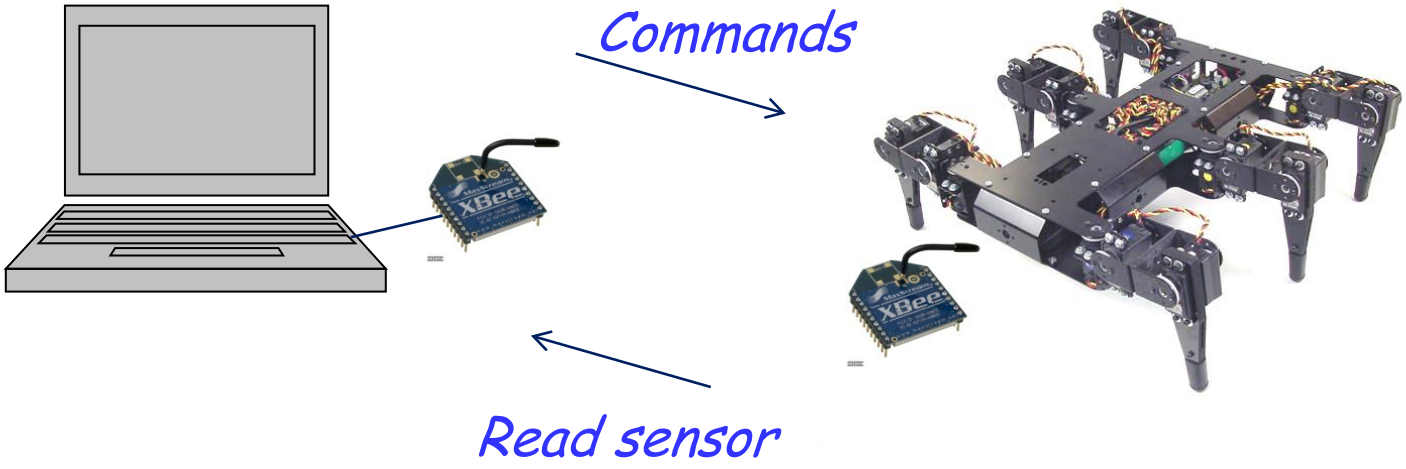
```
if not buf.is_full then put (buff, v1) end
if not buf.is_full then put (buff, v2) end
```

separate

```
put (buffer: ACCOUNT; v: G)
        -- Add v to buffer.
    require
        not buffer.is_full
    do
        ...
    ensure
        ...
    end
```

Пример 4: шестиногий робот

Шестиногий робот



Контролная SCOOP-программа исполняется на PC, передает команды бортовому сервоконтроллеру, опрашивает входы об информации с датчиков

Правила для координации шестиногого

R1: Protraction can start only if partner group on ground:

- **R2.1**: Protraction starts on completion of retraction
- **R2.2**: Retraction starts on completion of protraction

R3: Retraction can start only when partner group raised

R4: Protraction can end only when partner group retracted

Dürr, Schmitz, Cruse: *Behavior-based modeling of hexapod locomotion: linking biology & technical application*, in *Arthropod Structure & Development*, 2004

R1: Затягивание может начаться только тогда, когда партнер-группа на земле

- **R2.1**: Затягивание начинается по завершении отвода
- **R2.2**: Отвод начинается по окончании затягивания

R3: Отвод может начаться только тогда, когда партнер-группа подвышена

R4: Затягивание может закончиться только тогда, когда партнер-группа отведена

Последовательная реализация

```
TripodLeg lead = tripodA;
TripodLeg lag = tripodB;
while (true)
{
        lead.Raise();
        lag .Retract();
        lead.Swing();
        lead.Drop();

        TripodLeg temp = lead;
        lead= lag;
        lag= temp;
}
```

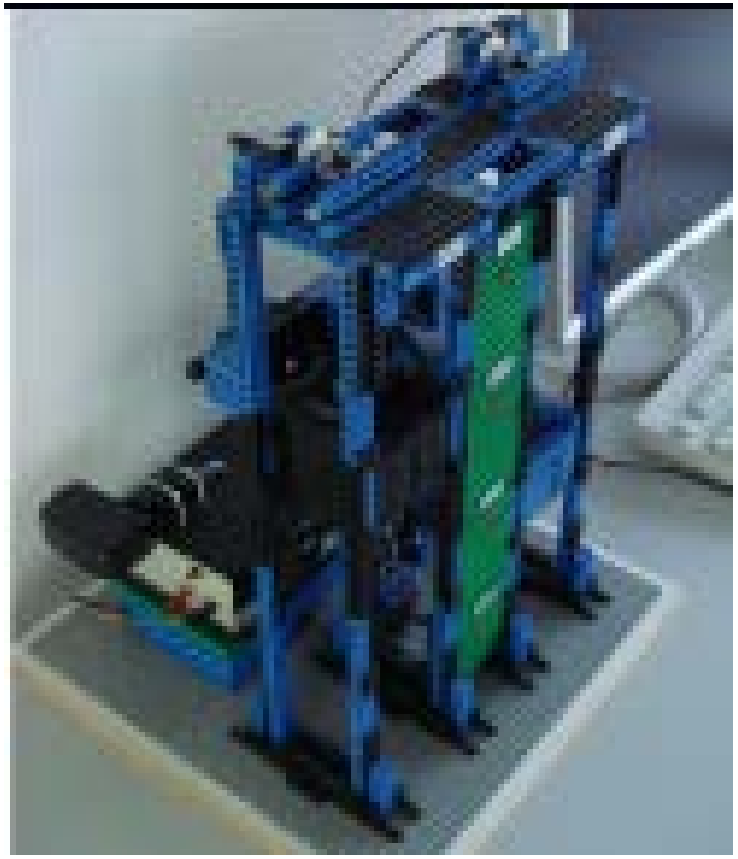
Классическая мнотопоточная реализация

```
private void ThreadProcWalk(object obj)
{
  TripodLeg leg = obj as TripodLeg;
  while (Thread.CurrentThread.ThreadState != ThreadState, AbortRequested)
  {
    //Waiting for protraction lock
    lock (m _protractionLock)
    {
      // Waiting for partner leg drop
      leg .Partner.DroppedEvent.WaitOne();
      leg .Raise();
    }
    leg.Swing();

    // Waiting for partner retraction
    leg.Partner.RetractedEvent.WaitOne();
    leg.Drop();

    // Waiting for partner raise
    leg .Partner.RaisedEvent.WaitOne():
    leg .Retract
  }
}
```

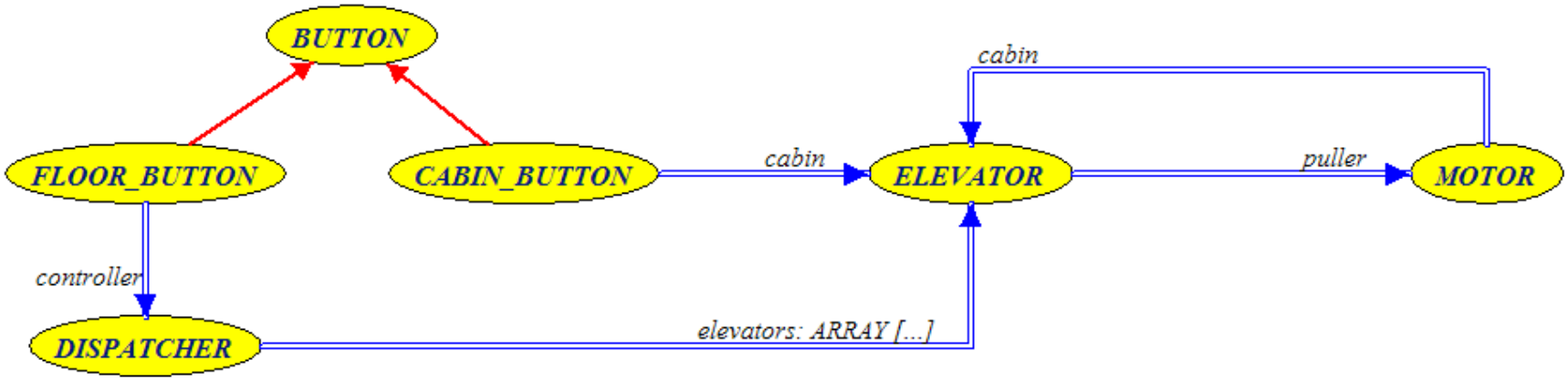
Версия SCOOP

```
begin_protraction (partner, me: separate LEG_GROUP)
        require
                me.legs_retracted
                partner.legs_down
                not partner.protraction_pending
        do
                tripod.lift
                me.set_protraction_pending
        end
```

Пример 5: Система управления лифтом



Система управления лифтом



From: *Object-Oriented Software Construction*

«Ошеломительно параллельная» система
("Embarrassingly parallel")
«Ошеломительно О-О»

Как написать SCOOP-программу

1. Проектируйте О-О программу как обычно: наилучшая модель системы

   Убедитесь, что включили все нужные **контракты**

2. Анализируйте, какие элементы должны быть **separate**

- 3 -

SCOOP: принципы и техники

Разработка модели SCOOP

Для того, чтобы достичь предыдущих целей, SCOOP применяет модели параллельности ряд **ограничений**

Остальная часть доклада представляет и **объясняет** эти ограничения

Цель – дать программистам способность **рассуждать** о своих программах: **"Concurrency Made Easy"**

SCOOP - не полная переделка программирования, а дополнение основной схемы ОО: **одно новое ключевое слово**

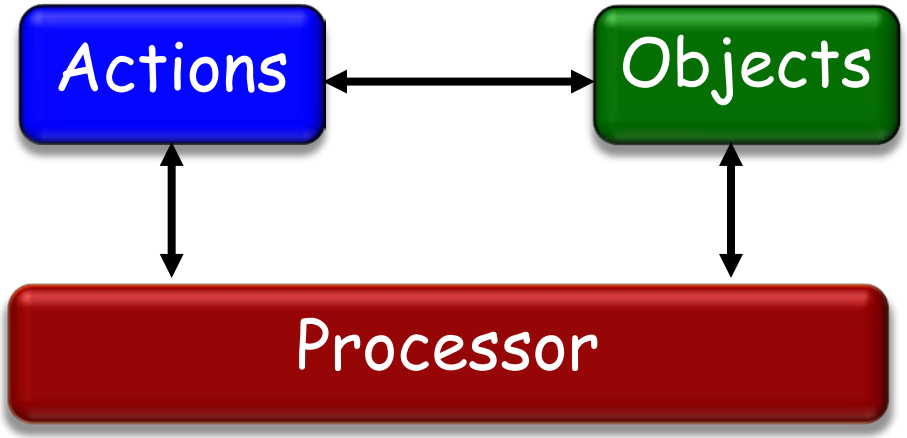
Шесть понятий

- **(A)** «Процессоры»
- **(B)** Области
- **(C)** Синхронные против асинхронные вызовы
- **(D)** Новая семантика передачи аргументов
- **(E)** Новая семантика ресинхронизации (ленивое ожидание – lazy wait)
- **(F)** Новая семантика предусловий

(A): Процессоры

Выполнение вычислений состоит в следующем:

- Применить некоторые действия
- К некоторым объектам
- Используя некоторые процессоры

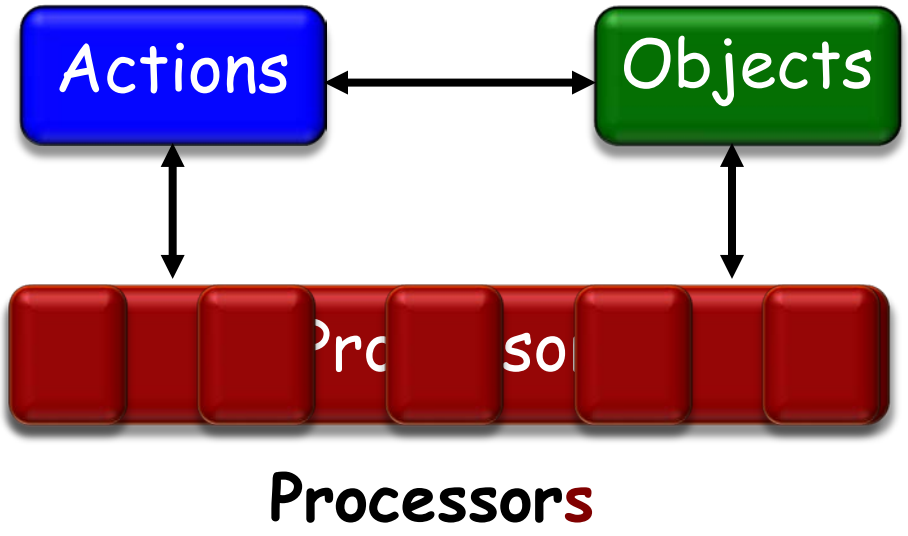


Последовательное исполнение: один процессор

(A): Процессоры

Выполнение вычислений состоит в следующем:

- Применить некоторые действия
- К некоторым объектам
- Используя некоторые процессоры



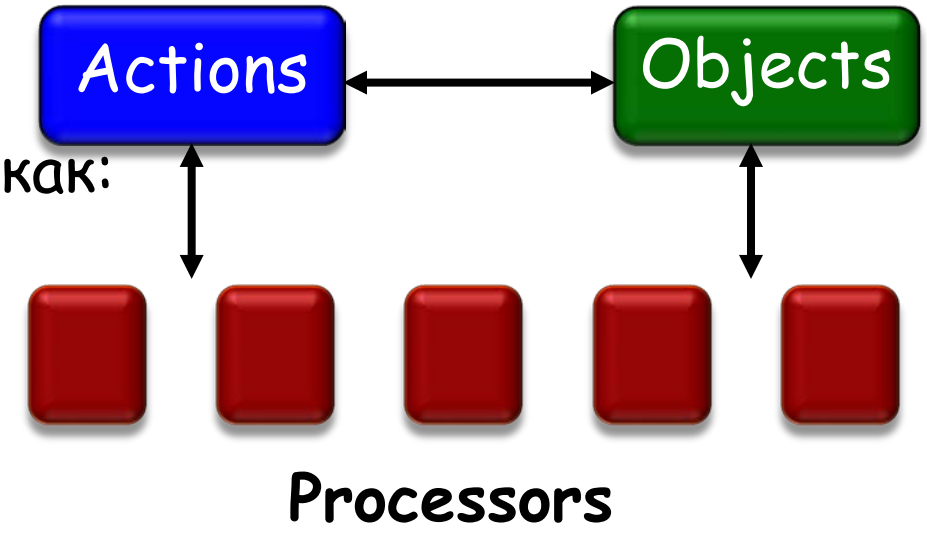
Последовательное исполнение: один процессор

Параллельное исполнение: любое число процессоров

Что делает приложение параллельным?

Процессор:

Поток управления, поддерживающий последовательное выполнение инструкций над одним или несколькими объектами



Может быть реализован как:

- Компьютер ЦПУ
- Процесс
- Поток
- …

Модель SCOOP – абстрактная, не определяет соответсвенность между процессорами и ресурсами

ОО программирование

Ключевой механизм – вызов подпрограммы («метода»):

```
x.r (args)
```

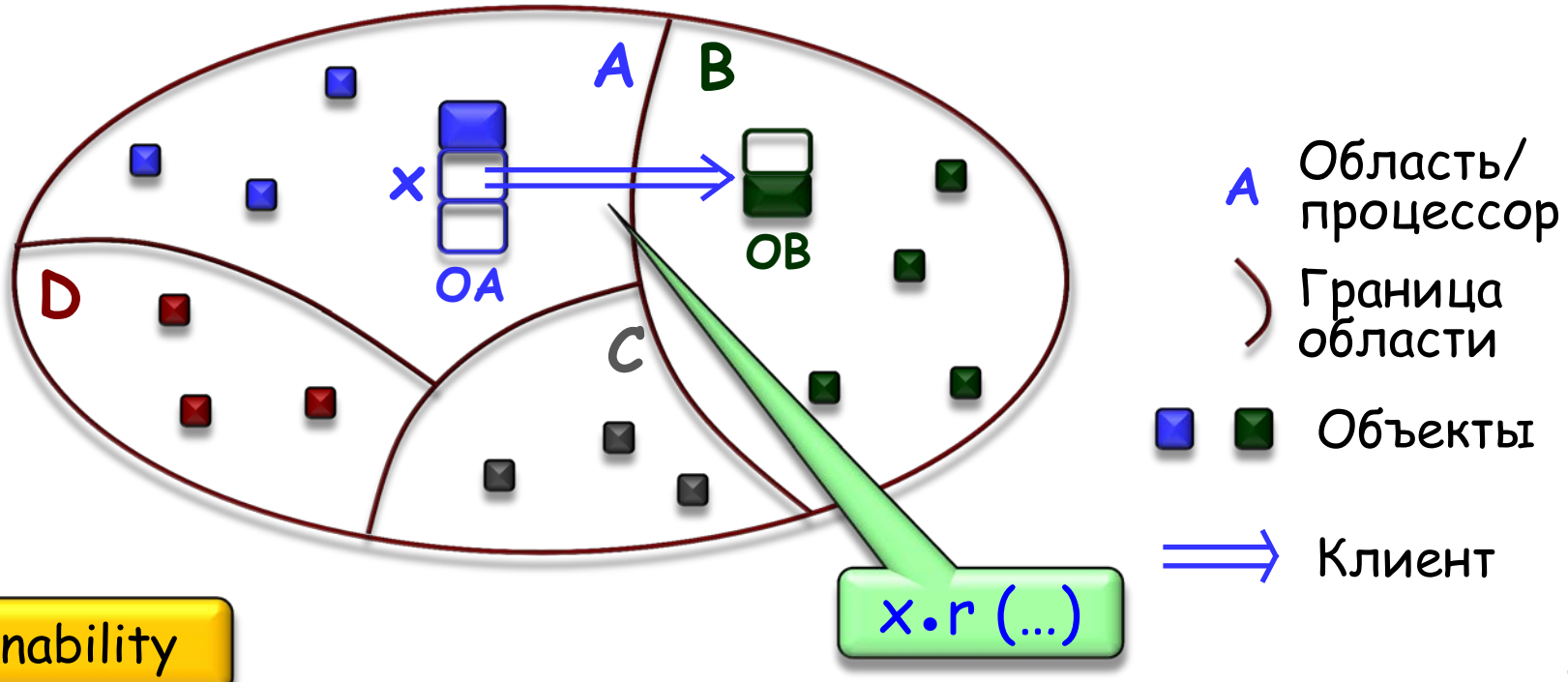
где x, **цель** вызова, обозначает объект, которому вызов применит подпрограмму r

В параллельном контексте, какой процессор ответственный за исполнение вызова?

(B): Области

Все вызовы с данной целей исполняет один процессор

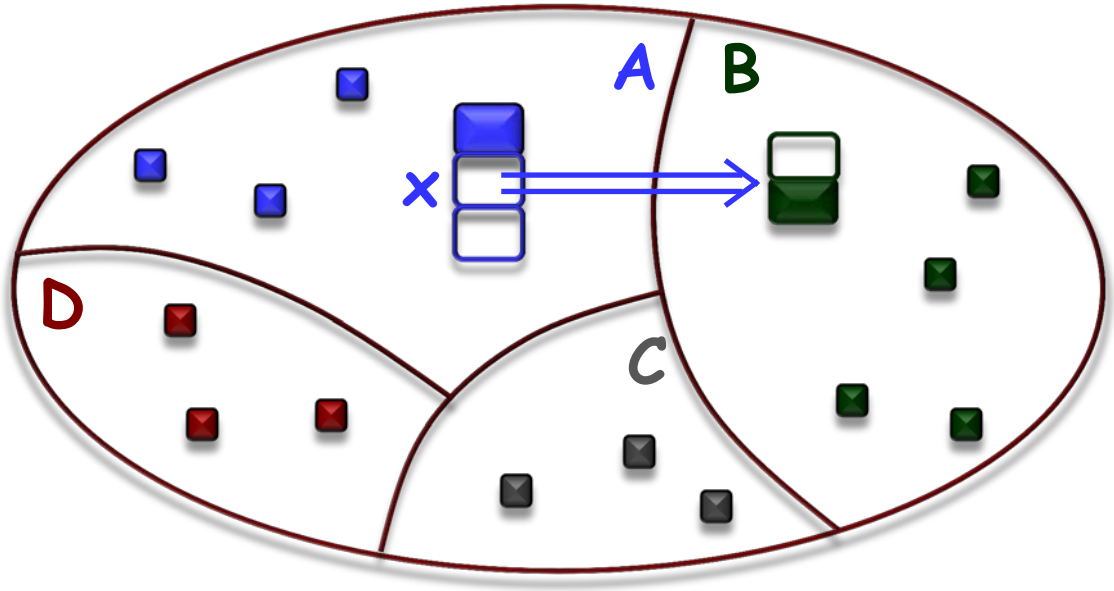
- Этот процессор называем ***обработчиком*** (*handler*) данного объекта
- Множество объектов с данным обработчиком называем *область (region)*



Reasonability

Области

Сушествует точное соответствие между процессорами и областями



Ограничение: один обработчик для каждого объекта

- В любой момент, на данном объекте, не больше одной операции (подпрограммы) может быть активной

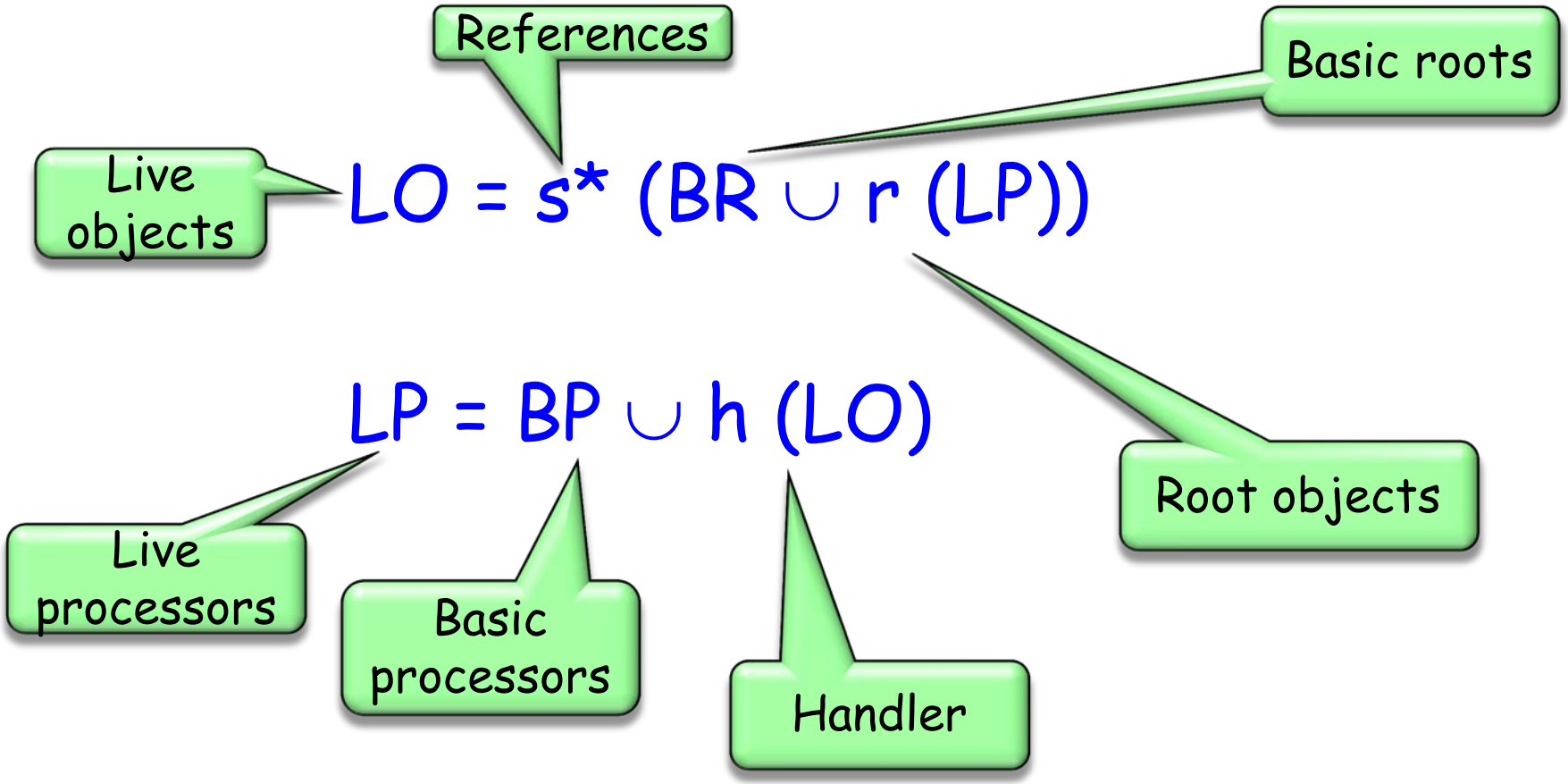
Reasonability

Уборщик мусора для процессоров

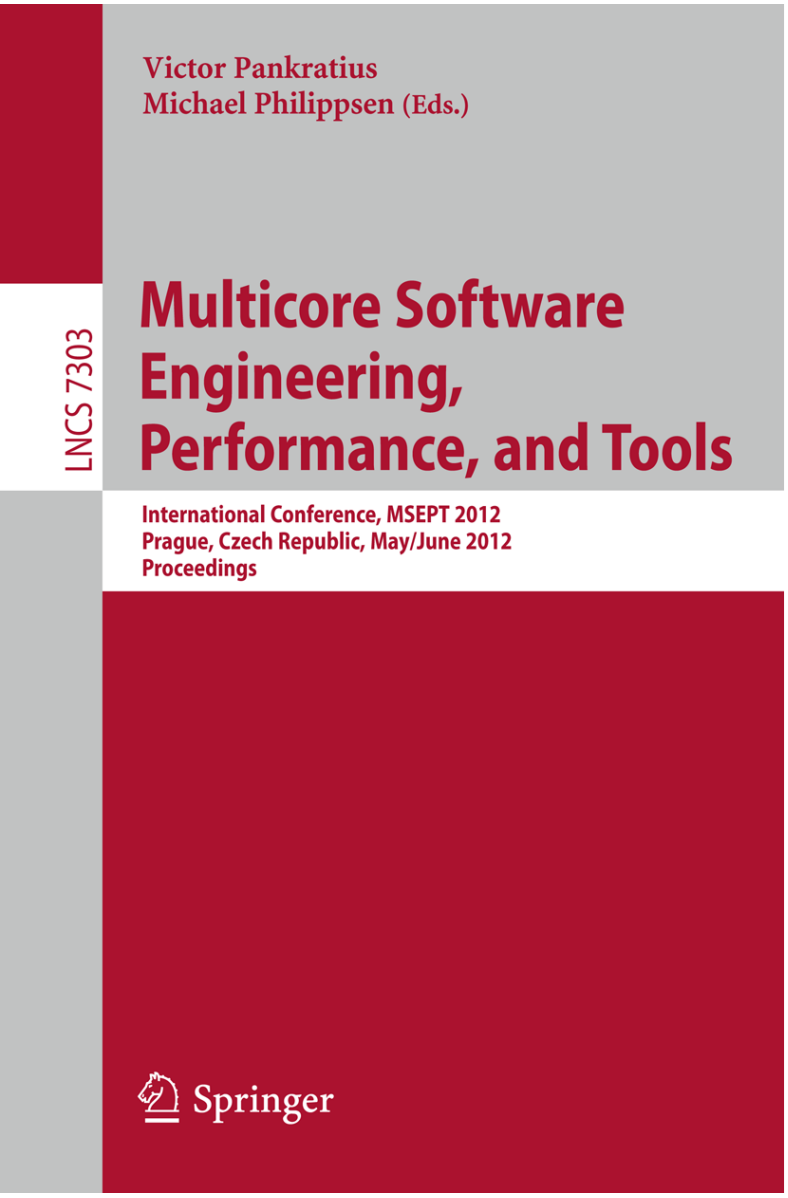
Александп Когтенков, MSEPT 2012

Как объекты, так и процессоры могут становиться недостижимым!

Сбор мусора для объектов и процессоров - переплетенный



MSEPT 2012



Processors and Their Collection

Bertrand Meyer[1,2,3], Alexander Kogtenkov[2,3], and Anton Akhi[3]

[1] ETH Zurich, Switzerland
[2] ITMO National Research University, Saint Petersburg, Russia
[3] Eiffel Software, Santa Barbara, California
se.ethz.ch, eiffel.com, sel.ifmo.ru

Abstract. In a flexible approach to concurrent computation, "processors" (computational resources such as threads) are allocated dynamically, just as objects are; but then, just as objects, they can become unused, leading to performance degradation or worse. We generalized the notion of garbage collection (GC), traditionally applied to objects, so that it also handles collecting unused processors.

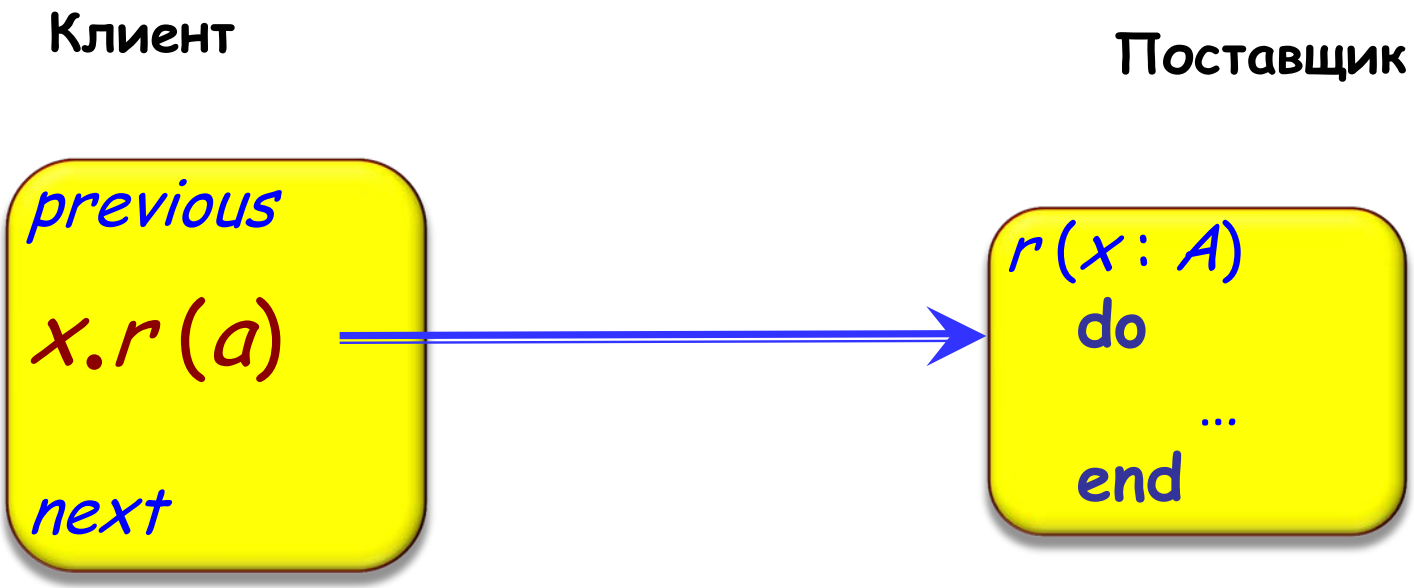
The paper describes the processor collection problem, formalizes it as a set of fixpoint equations, introduces the resulting objects-and-processor GC algorithm implemented as part of concurrency support (the SCOOP model) in the latest version of EiffelStudio, and presents benchmarks results showing that the new technique introduces no overhead as compared to traditional objects-only GC, and in fact improves its execution time slightly in some cases.

1 Overview

Few issues are more pressing today, in the entire field of information technology, than providing a safe and convenient way to program concurrent architectures. The SCOOP approach to concurrent computation [5] [6] [7] [8] [9] , devised in its basic form as a small extension to Eiffel, is a comprehensive effort to make concurrent

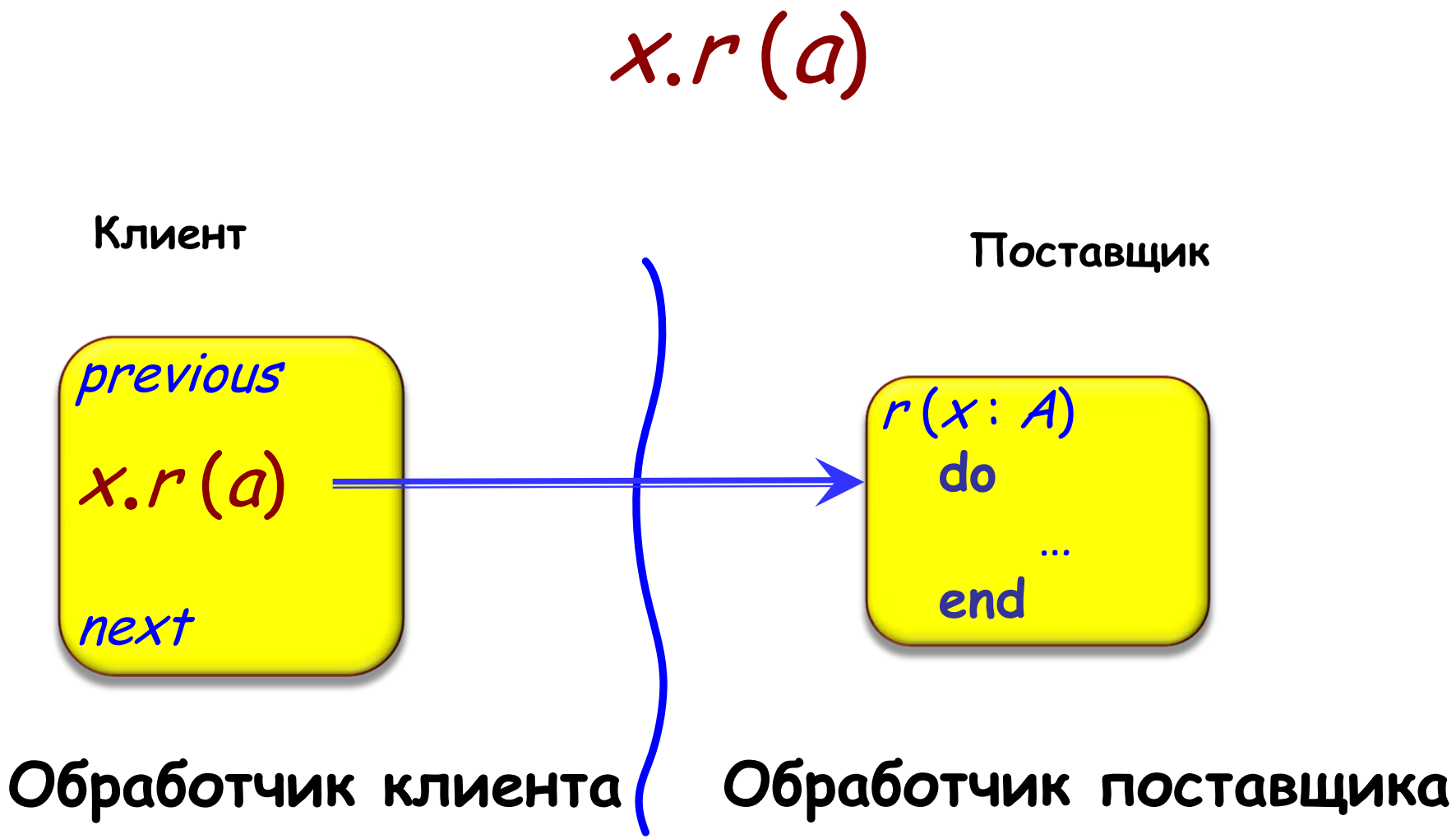
(C) Семантика вызовов: синхронный случай

$$x.r(a)$$



Процессор

(C) Семантика вызовов: асинхронный случай



Два типа вызова

Ждать или не ждать:

- Если тот же процессор, синхронный
- Если разные процессоры, асинхронный

Reasonability

Различия должны появляться в синтаксисе:

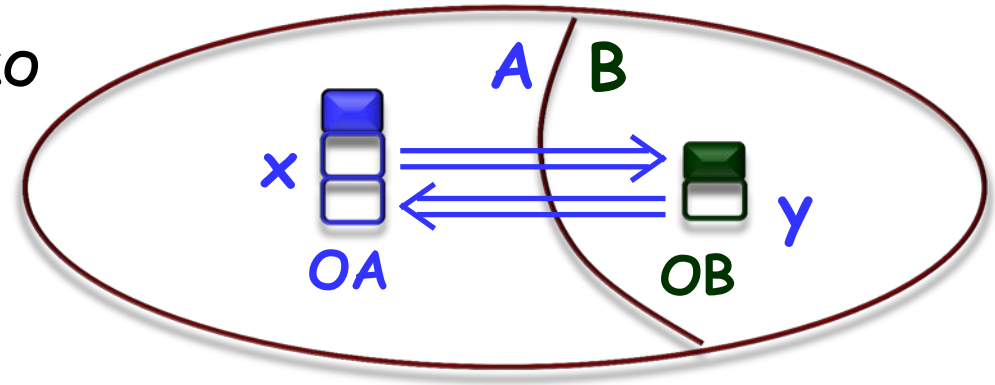
- x: T
- x: **separate** T -- потенциально другой процессор

Базовое семантическое правило: вызов *x.r (a)*

- Ждет, если *x* - не-separate
- Не ждет, если *x* - separate

Почему *потенциально* separate?

separate объявление только утверждает, что объект *мог бы* иметь другой обработчик



- В классе *A*: x: **separate** *B*
- В классе *B*: y: **separate** *A*
- В *A*, какой тип для x.y?

В каком-то исполнении, значение x мог бы указателем к объекту в той же области

Вызов - не то же самое, что применение

Асинхронность нас заставляет различать между **feature вызовом** и **применением** подпрограммы

Вызов - исполнение инструкции

```
x . r (...)
```

и не ждет в случае синхронности (клиент только регистрирует вызов и проходит дальше)

Действительное исполнение тела подпрограммы r происходит позже, и называется её **применением**

Правила согласованности: как уничтожить предателей

nonsep: *T*

sep: **separate** *T*

nonsep := *sep*

nonsep.*p*(*a*)

Предателя!

Система типов предотвращает предателей!

Reasonability

(D) Политика контроля доступа

Так как separate вызовы асинхронные, есть настоящий риск путания

Пример:

```
st : separate STACK [T]
...
st .put (a)
... Instructions not affecting the stack...
y := st.item
```

Reasonability

(D) Политика контроля доступа

SCOOP требует, что цель separate вызова – **формальный аргумент** ограждающей подпрограммы:

```
put (s: separate STACK [T]; value : T)
        -- Store value into s.
    do
        s.put (value)
    end
```

Чтобы использоваться separate объект:

```
st: separate STACK [INTEGER]
create st
put (st,  10)
```

Reasonability

(D) Правило separate аргументов

Цель separate вызова должен быть аргументом окружающей подпрограммы

Separate вызов: $x.r(...)$, где x - separate

(D) Политика контроля доступа

Делаем предыдущий пример правильным:

```
handle (s : separate STACK[T]                    )
        do
                ...
                s.put (a)
                ... Instructions not affecting the stack...
                y := my_stack.item
        end
```

(D) Правило эксклюзивного доступа

Вызов гарантирует эксклюзивный доступ обработчикам (процессорам) всех separate аргументов

r (*a*, *b*, *sep_c*, *sep_d*, *sep_e*)

Эксклюзивный доступ к to *sep_c*, *sep_d*, *sep_e* в теле *r*

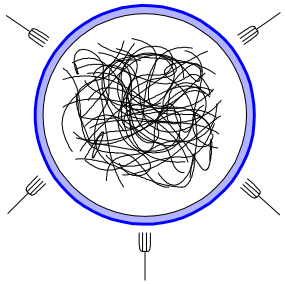
Reasonability

An example: from sequential to concurrent

```
transfer (source, target: separate ACCOUNT;
          amount: INTEGER)
    -- Transfer amount, if available, from source to target.
  do
    if source.balance >= amount then
      source.withdraw  (amount)
      target.deposit   (amount)
    end
  end
```

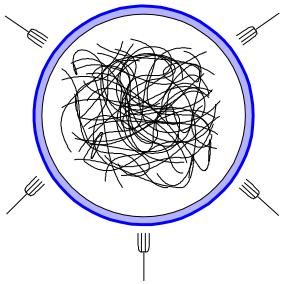
Reasonability

Обедающие философы на SCOOP (1)



```
class ФИЛОСОФ feature
        левая, правая: ВИЛКА
        вставать do … end
        закончено : BOOLEAN

        жить
                do
                        from вставать until закончено loop
                                думать;  обедать (левая, правая)
                        end
                end

        обедать ( л, п : separate ВИЛКА )
                        -- Есть, при наличии вилок л и п.
                do л.взять; п.взять ; … end
end
```

Dining philosophers in SCOOP (2)



```
class PHILOSOPHER inherit
        REPEATABLE
                rename
                        setup as getup
                redefine step end

feature {BUTLER}
        step
                do
                        think ;   eat (left, right)
                end

        eat (l, r: separate FORK)
                        -- Eat, having grabbed l and r.
                do ... end
end
```

A library class describing processes

SCOOP integrates inheritance and other O-O techniques with concurrency, seamlessly and without conflicts ("inheritance anomaly")

No need for built-in notion of **active object**: it is **programmed** through a library class such as :

```
class REPEATABLE feature
    setup do end
    step do end
    over : BOOLEAN
    tear_down do end
    live
            do
                    from setup until over loop step end
                    tear_down
            end
    end
end
```

(D) Что действительно значит правило эксклюзивного доступа

Бить самого опасного врага параллельности: нарушения атомарности

- Гонки (data races)
- Неправильное чередование вызовов

В SCOOP, эти ошибки не могут происходить

(D) Exclusive access rule

A call guarantees exclusive access to the handlers (processors) of all separate arguments

r (*a*, *b*, *sep_c*, *sep_d*, *sep_e*)

Exclusive access to *sep_c*, *sep_d*, *sep_e* within *r*

Semantics vs implementation

Older SCOOP literature says that feature application "waits" until all the separate arguments' handlers are available

This is not necessary!

What matters is **exclusive access**: implementation does not have to wait unless semantically necessary

The implementation performs some of these optimizations

```
r (a, b, c : separate T)
        do
                something_else
                a.r
                b.s
        end
```

No need to wait for a and b until here

No need to wait for c!

Implementation techniques

The literature on transactional memory (Herlihy) criticizes approaches of the form

- Lock everything first
- Then think

Conceptually, SCOOP is of this kind

But it can be implemented in a TM-like style

(E) Снова синхронизировать

Как снова синхронизировать после асинхронных вызовов?

Нет специального механизма!

Клиент будет ждать тогда, и только тогда, необходимо:

```
x.f (...)
x.g (...)
x.h (...)
...
value := x.some_query
```

Ждать здесь!

Ленивое ожидание (lazy wait)

(также: wait by necessity)

Reasonability

(E) Синхронность против асинхронность

Для separate цели *x* :

- *x.command* (...) – асинхронный
- *v* := *x.query* (...) - синхронный

Exercise

If we do want to resynchronize explicitly, what do we do?

(F) Контракты

Что случится с контрактами, особенно с предусловиями, в параллельном контексте?

(F) Контракты

```
put (b: separate QUEUE[INTEGER]; v: INTEGER)
            -- Store v into b.
        require
            not b.is_full

        do
            b.put (v)
        ensure
            not b.is_empty
            ...
        end

...
```

В клиенте:

```
put (my_buffer, 10 )
```

(F) Контракты

```
put (b: separate QUEUE [INTEGER]; v: INTEGER)
            -- Insert v into buffer b.
        require
            not b.is_full

        do
            b.put (v)
        ensure
            not b.is_empty
        end
```

В клиенте:

```
put (my_buffer, 10 )
```

Предусловие становится **условием ожидания**

.

Bank transfer

```
transfer (source, target: separate ACCOUNT;
              amount: INTEGER)
        -- Transfer amount from source to target.
    require
         source.balance >= amount
    do
        source.withdraw  (amount)
        target.deposit     (amount)
    ensure
        source.balance = old source.balance - amount
        target.balance = old target.balance + amount
    end
```

(F) Полное правило синхронизации

Вызов с separate аргументами:

- Имеет эксклюзивный доступ ко всем соотвествующим объектам
- Ждет до того, как все их separate предусловия выполняются

"Separate предусловие":

x.some_property -- где *x* - separate

Reasonability

Какая семантика применяется?

```
put (buf: separate QUEUE [INTEGER]; i: INTEGER)
        require
                not buf.is_full
                i > 0
        do
                buf.put (i)
        end
```

Условие ожидания

Условие коррекности

```
my_buffer: separate QUEUE [INTEGER]
put (my_buffer, 10)
```

Рассуждать об объектах: в последовательном контексте

$$\frac{\{\mathrm{INV}\ \textbf{and}\ \mathrm{Pre}_r\}\quad \mathrm{body}_r\quad \{\mathrm{INV}\ \textbf{and}\ \mathrm{Post}_r\}}{\{\mathrm{Pre}_r'\}\quad x.r\,(a)\quad \{\mathrm{Post}_r'\}}$$

Только n доказательств,
если n exпортированных методов!

Рассуждать об объектах: в SCOOP

$$\frac{\{INV \wedge Pre_r(x)\}\ body_r\ \{INV \wedge Post_r(x)\}}{\{Pre_r(a^{cont})\}\ \ e.r(a)\ \ \{Post_r(a^{cont})\}}$$

Мы сохраняем последовательное рассуждение в стиле Хоор

Контролируемое выражение – выражение, на который обработчик имеет доступ:

- Либо не-separate
- Либо уже резервировано

- 4 -

SCOOP: система типов

Предатель

Предатель – переменное, которое

- Статически объявляется не-separate
- Во время одного возможного исполнения, обозначает separate объект

Consistency Rules: First Attempt

Original model (Object-Oriented Software Construction, chapter 30) defines four consistency rules that eliminate traitors

Written in English

Easy to understand by programmers

Sound? Complete?

Original consistency rules: example

Separateness Consistency Rule (1)

If the source of an attachment (assignment or argument passing) is separate, its target must be separate too.

```
r (buf: separate BUFFER [T]; x: T )
    local
        buf1: separate BUFFER [T]
        buf2: BUFFER [T]
        x2: separate T
    do
        buf1 := buf          -- Valid
        buf2 := buf1             -- Invalid

        r (buf1, x2)         -- Invalid
    end
```

A type system for SCOOP

Piotr Nienaltowski, 2008

Goal: prevent traitors through static (compile-time) checks

Simplifies, refines and formalizes SCOOP rules

Integrates expanded types and agents with SCOOP

Tool for reasoning about concurrent programs

- May serve as basis for future extensions, e.g. for deadlock prevention schemes

Three components of a type

Notation:

Γ |- x :: (γ, α, C)

Under the binding Γ, x has the type(γ, α, C)

1. Attached/detachable: $\gamma \in \{!, ?\}$

Some processor (top)

x: **separate** U

Current processor

2. Processor tag $\alpha \in \{\bullet, \top, \bot, \langle p \rangle, \langle a.handler \rangle\}$

3. Ordinary (class) type C

No processor (bottom)

Result type combinator

What is the type T_{result} of a query call `x.f (...)`?

$T_{result} = T_{target} * T_f$

$= (\alpha x, px, TX) * (\alpha f, pf, TF)$

$= (\alpha f, \mathbf{pr}, TF)$

pr

px \ pf	•	T	<q>
•	•	T	T
T	T	T	T
<p>	<p>	T	T

Argument type combinator

What is the expected actual argument type in x.f (a)?

$T_{actual} = T_{target} \otimes T_{formal}$

$= (\alpha x, px, TX) \otimes (\alpha f, pf, TF)$

$= (\alpha f, \mathbf{pa}, D)$

pa

px \ pf	•	⊤	⟨q⟩
•	•	⊤	⊥
⊤	⊥	⊤	⊥
⟨p⟩	⟨p⟩	⊤	⊥

- 5 -

SCOOP: Статус и будущее

Настоящее состояние и следующие шаги

Все предыдущие механизмы – в EiffelStudio (http://eiffel.com)

Текущая работа:

- Эффективость
- Импротирование объектов
- Улучшение механизма исключений

Открытые проблемы

- Множественные читатели
- Избежание блокировки (deadlock prevention and detection)

Некоторые темы исследования

- Библиотеки и шаблони
- Полное семантическое определение, верификация (EVE)
- Миграция объектов
- Распределенные вычисления
- Realtime
- SIMD-вычисления

Что может SCOOP делать для нас?

Бить самого опасного врага в мире параллельности: нарушения атомарности

- Гонки (Data races)
- Неправильное чередование вызовов

Основные понятия SCOOP

- Тесная связь с моделированием О-О
- Естественное пользование такими механизнами О-О как наследование
- Гонок нет
- Очень легко резервировать любое число ресурсов вместе
- Встроенная справедливость (Built-in fairness)
- Освобождает программиста от многих забот
- Поддерживает разнообразные формы параллельности
- Сохраняет хорошо понятые формы рассуждения
- Легко выучить и использовать

Reasonability

Наша лаборатория в ИТМО

Software Engineering Laboratory

(Лаборатория программной инженерии и верификации)

sel.ifmo.ru

Наш подход («*под защитой чуждых крыл…*»)

Наша лаборатория работает в узком сотрудничестве с моей кафедрой (Chair of Software Engineering) в ETH Zurich (и Eiffel Software в Санта-Барбаре)

Члены лаборатории регулярно участвуют в нашей летней школе «LASER» на острове Эльба в сентябре и в других мероприятиях

Темы сотрудничества:

- Техники верификации, особенно инварианты
- Параллелизм
- Эмпирический анализ т.н. «гибких методов разработки» (“agile methods”)

Андрей Петрович Ершов