The RAISE Method Group: АЛГЕБРАИЧЕСКОЕ ПРОЕКТИРОВАНИЕ КЛАССА

перевод: ПИСКУНОВ А.Г.

18 октября 2007 г.

АННОТАЦИЯ

Значительная часть документа представляет собой перевод разделов 2.2, 2.3, 2.5, 2.6 из [4] - монографии К.Джоржа (Chris George) и др. (The RAISE Development Group, далее RDG).

В документе рассматривается техника алгебраического проектирования абстрактного типа данных, широко используемая в методе формальных спецификацей RAISE. Одним из первых, кто предложил такой подход был John Guttag (см., например, [5] , http://sql.ru/forum/actualthread.aspx?tid=451704&pg=1#4400855), также см. Библиографические замечания к лекции Бертрана Мейера Абстрактные типы данных [2] . Более простой пример алгебраического подхода к проектированию стека можно посмотреть в [2] . Поскольку под классом можно понимать пару тип_данных + функции (как в работе [3]), а в рассматриваемой технике помимо проектирования собственно абстрактного типа данных выполняется проектирования фукнций работы с ним, то это эту технику можно считать техникой алгебраического проектирования классов (см. [1]). Далее, хотя Мейер, Гуттаг и RDG аксиомами описывают поведение функций класса, только RDG описывает конкретную процедуру выбора таких аксиом для

описания поведения пар функций, вырабатывающих значение типа (generator) и извлекающих данные из значения типа (observer).

Для обсуждаемого примера Harbor было разработано полноценное RDBMS приложение ([6]).

Содержание

CONTENTS **3**

1 Выбор стиля спецификаций **4**

2 Абстракность **4**

3 Первый пример: harbor **6**
- 3.1 Цели примера 6
- 3.2 Требования к функциональности системы 7
- 3.3 Начальная постановка задачи 7
- 3.4 Обзор метода 11
- 3.5 Модуль типов 13
- 3.6 Абстрактный аппликативный модуль 13
  - 3.6.1 Обсуждение аксиом 17
    - 3.6.1.1 Функции waiting, arrives. 18
    - 3.6.1.2 Функции waiting, docks. 18
    - 3.6.1.3 Функции waiting, leaves. 18
    - 3.6.1.4 Функции occupancy, arrive. 19
    - 3.6.1.5 Функции occupancy, docks. 19
    - 3.6.1.6 Функции occupancy, leaves. 19
    - 3.6.1.7 Условие целостности (consistent). 20

4 ASCII версии для символов RSL **21**

1 Выбор стиля спецификаций

В выборе стиля спецификаций существует четыре альтернативы:

- аппликативный последовательный - функциональное программирование без переменных и параллельных вычислений(concurrency);
- императивный последовательный - программирование с переменными, присваиванием, циклами и т.д., но без параллельных вычислений;
- аппликативынй конкурентный - функциональное программирование с параллельными вычислениями;
- императивный конкурентный - программирование с переменными, присваиванием, циклами и т.д., и с параллельными вычислениями;

Аппликативынй конкурентный стиль считается неподходящим как базис для программирования на языке реализации. Остается три альтернативы, которые будут далее называться аппликативная, императивная и конкурентная.

2 Абстракность

Также как и разделение между аппликативным и императивным; последовательным или конкурентным, отмечается разница между абстрактным и конкретным стилями. Под абстрактностью мы понимаем такое свойство спецификаций, при которых остается так много открытых альтернативных путей разработки как возможно. Другими словами, чем меньше проектных решений мы приняли при написании спецификации, тем более она обстрактна. Под проектными решениями мы понимаем решения вроде следующих

- решения как определить модуль;
- решения о конкретной структуре данных;
- решения о конкретных алгоритмов;

- решения о используемых переменных;
- решения о используемых каналах и образцах обмена информации для использования.

Противоположностью абстракности есть конкретность. Различия между ними не есть черно белыми, однако дают возможность охарактеризовать некоторый модуль, написанный в стиле одной из трех категорий, как более абстрактный или более конкретный.

- абстрактный аппликативный - модуль содержит абстрактные типы и сигнатуры функций с аксиомами, а не явные определения функций;
- конкретный аппликативный - модуль содержит конкретные типы и явные определения функций;
- абстрактный императивный - модуль не определяет переменных, зато использует ключевое слово any в описании его доступа. Содержит аксиомы;
- конкретный императивный - модуль содержит определение переменных и явные определения функций;
- абстрактный конкурентный - модуль не содержит определения каналов, однако содержит ключевое слово any в описании доступа. Содержит аксиомы;
- конкретный конкурентный - модуль содержит явные определения переменных, каналов и функций;

И опять надо подчеркнуть, что эти различия более относительные, чем абсолютные. Модуль может быть абстрактный в одном смысле и конкретный в другом. И, естественно, вся спецификация будет содержать модули сочетающие все три стилей и две степени абстракции.

3 Первый пример: harbor

3.1 Цели примера

Пример является простой информационной системой, имеющая функции изменения данных, опрашивания данных и инвариантные свойства (условия целостности) которым данные должны удовлетворять. Не выставляется требований к конкуретному (параллельному) доступу.

First example: harbour 43

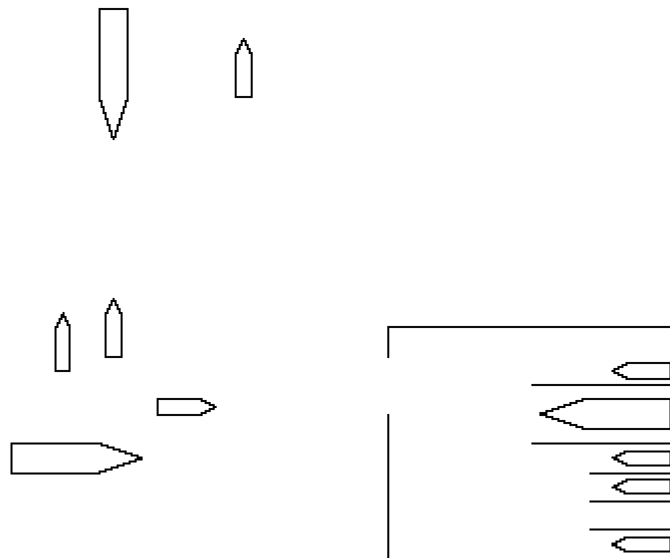

Figure 2.1: Harbour

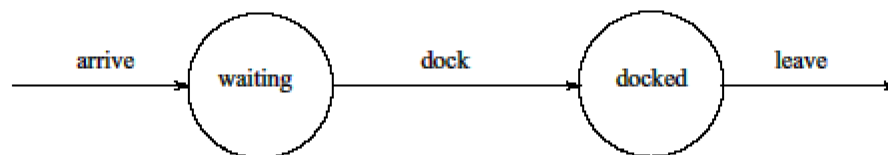

Figure 2.2: State transitions for ships

Рис. 1: Порт и изменение состояния судна

3.2 Требования к функциональности системы

Корабли прибывающие (arrive) в порт, будут распределяться по подходящим свободным причалам или ждать на рейде, до тех пор пока подходящий причал не станет свободен. Система должна поддерживать следующие функции позволяющие руководителю порта управлять движение судов в порту:

- arrive: зарегистрировать прибытие судна;
- dock: зарегистрировать причаливание судна к причалу;
- leave: зарегистрировать уход судна из порта.

Порт представлен на рисунке 1 . Мы предполагаем, что все корабли должны прибыть и ожидать на рейде (in the pool) прежде чем они могут причалить. Такое изменение состояния судна отражено на рисунке 1 .

3.3 Начальная постановка задачи

Сначала мы должны спросить что является объектами в системе? В требованиях к функциональности упоминались судна (ships), причалы (berths), рейд (pool) и порт (harbour). С нашей точки зрения можно считать, что порт является фиксированным списком причалов, в то время, как количество судов на рейде будет меняться. Схема взаимоотношений (связей) между всеми объектами системы отображены на рис. 2 .

Затем попытаемся идентифицировать атрибуты объектов и посмотрим, какие из них будут менятся динамически. Единственное атрибут к судно упомянутый в требованиях - судно может соответствовать или не соответствовать (fit) причалу. Можно ввести в рассмотрение такой атбирут как размер (size), однако достоверно не известно можно ли через размер определить соответствие судна причалу. Таким образом заметим, что мы возможно будем нуждаться в функции

```
---- File:fits.r


  fits : Ship >< Berth -> Bool

---- End Of File:fits.r
```

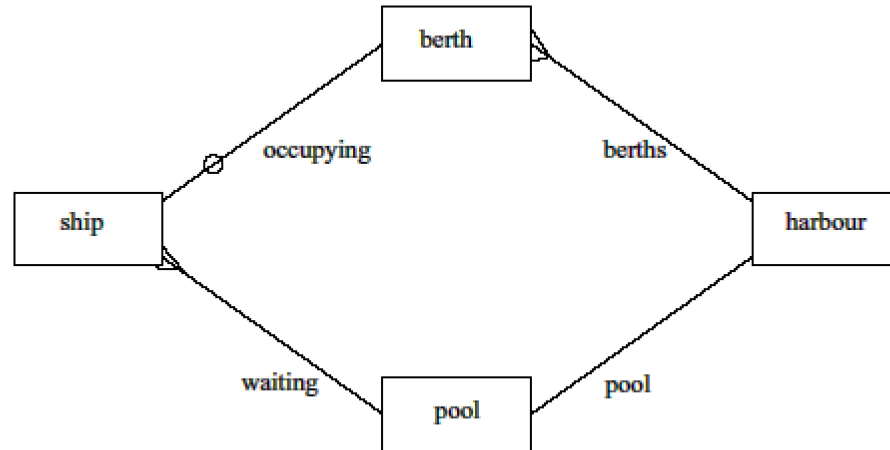

Figure 2.3: Entity relationships for harbour

Рис. 2: Схема взаимоотношений между объектами порта

Эта функция мы оставим неспецифицированной, по крайней мере до тех пор, пока мы не обсудим с заказчиком, что она может означать.

Причалы изменяют свой атрибут в том смысле, что они могут быть свободными в один момент времени и заняты некоторым судном в другой. Таким образом мы может определить термин заполненность (occupancy) как динамический атрибут. Это предполагает что причал будет императивным объектом RSL с функциями которые меняют состояние объекта - например, причалить (enter) и покинуть (leave).

Сам порт, похоже, является множеством причалов. Количество элементов этого множества очевидно фиксированно. Таким образом можно рассматривать его как массив.

Рейд (pool) с ожидающими суднами будет изменятся динамически, по мере того, как судна будут прибывать (arrive) и причаливать (dock). Таким образом тут тоже рекомендуется императивный объект RSL с изменяющими его состояние функциям- например, enter и leave.

Далее, существует выбор того, что мы можем трактовать как атри-

бут. Мы можем иметь динамический атрибут положение (location) судна, которое может быть гдето, ожидать на рейде или быть причаленным (docked) на причале K. Мы могли бы сделать судна императивным объектом RSL чтобы промоделировать это. Тогда, в случае если система содержит динамические причалы и рейд, мы будем иметь дублирование информации. Это вызвало бы лишние накладные расходы при изменение обоих объектов согласованно (This would cause extra overhead in changing both objects consistently). Некоторые системы проектируются таким образом - обычно когда количество информации большое, запросы делаются часто и должны быть быстрыми, а изменения состояния редкие. Одноко это опасная практика и для этой системы более подходящим подходом будет рассматривать систему как причал и множество ждущих на рейде кораблей и вычислять положение судна тогда, когды мы будем в этом нуждаться.

Далее обсудим инварианты (свойства, которые всегда выполняются) данных. Такими свойствами будут следующие возможности:

- судно не может быть в двух местах одновременно;
- на причале не может быть более одного судна;
- судно может причалить только к подходящему причалу (fits).

Существует два пути чтобы работать с такими инвариантами. Если возможно, мы встроим их в модель. Если занятость (occupancy) причала моделируется либо как свободный (vacant) либо как занят (occupied_by(s), где s - судно), в модели становится недопустимой возможность причаливания более судна причалят к причалов, и таким образом будет гарантирован второй инвариант. (Необходимо также заметить, что мы не должны пытаться причалить судно к причалу, если причал уже занят, однако такая ситуация будет обрабатываться отдельно.) Мы уже решили что строим модель, в которой считается, что множество причалов не меняется. Это условие также рассматривается как инвариант.

Первый инвариант задается императивным предикатом

```
---- File:inv.r
```

```
all s: Ship :- ~(waiting(s) /\ is_docked(s)) /\
   (all b1, b2: Berth :-
       occupancy(b1) = occupied_by(s) /\
       occupancy(b2) = occupied_by(s) =>
       b1 = b2
   ) /\

   (all b: Berth :- occupancy(b) =  occupied_by(s) => fits(s,b) )

---- End Of File:inv.r
```

Мы ожидаем что в начальной спецификации будет использован абстрактный тип для порта (harbor). Чтобы указать инвариантное свойство отраженное предикатом consistent, можно использовать подтип как в следующем примере

```
---- File:hrb.r


type
  Harbour_base,
  Harbour = {| h: Harbour_base :- consistent(h) |}

---- End Of File:hrb.r
```

Мы можем использовать эту возможность, однако это потребует от нас генерирование условий доверия (confidence conditions) (см. секцию 4.1.2 [4]) для конкретной аппликативной спецификации (когда будет построен некоторый конкретный тип Harbour_base). С другой стороны очень легко создать конкретную аппликативную спецификацию, которая выдержит проверку соответствия (implementation check) абстрактной, но не будет поддерживать выполнения инвариантов, и таким образом не будут выполнятся условия целостности (inconsistent). Это общее правило что подтипы абстрактных типов не должны использоваться до тех пор, пока не будут сгенерированны и проверены условия доверия (confidence conditions) для конкретных модулей.

Вместо этого, свойство "изменяющие состояние функции поддерживают инварианты"мы выразим набором аксиом. Такой подход делает свойства более видимыми и будет заставлять нас доказываать что они

выполняются при проверки реализации (which makes the property more visible and will force us to justify it when we justify implementation). Может для этого примера это не кажется слишком важным, но в друге примере - The Lift - мы обнаружим, что свойства безопасности выглядят как инварианты.

Например, если arrive это изменяющая состояние фукнция и consistent предикат выражающий инвариант, мы можем записать аксиому

```
---- File:axiom.r


axiom
  [arrive_consistent]
  all s: Ship :-
   arrive (s)  post consistent()
               pre consistent() /\ can_arrive(s)

---- End Of File:axiom.r
```

где can_arrive это предикат выражающий предусловия для arrive.

Теперь мы имеем некую картину объектов в системе. Мы можем изобрасить их как схеме 3 причем там показаны только изменяющие состояние функции. Хотя мы легко могли бы выразить на RSL сначала написав схемы POOL, BERTH и BERTHS, воощe говоря хорошей идеей является попробовать сначала одну схему без компонент которая наиболее полно удовлетворяет требованиям. А затем произвести декомпозицию модели.

Как альтернативный путь, можно сначала выполнить декомпозицию системы, чтобы лучше понять, как система будет работать вместе и, возможно, создав абстракцию позже. Как мы отмечали раньше, опыт использования RAISE советует что конструирование более конкретной и структурированной системы, вообще говоря, является более успешной техникой.

3.4 Обзор метода

Мы будем двигаться от аппликативного к императивному описанию. Таким образом практически метод будет использоваться как показано ниже:

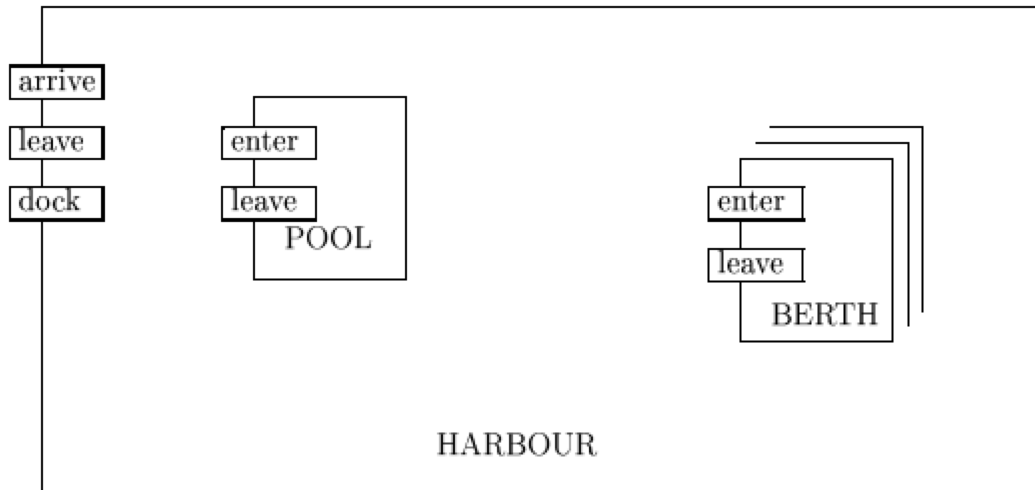

Figure 2.4: Harbour objects

Рис. 3: Объекты порта

- Определить схему TYPES содержащую типы и атрибуты для не динамических сущностей и сделать глобальный объект T для этой схемы;
- Определить абстрактный аппликативный модуль A_HARBOUR0 содержащий функции высшего уровня, аксиомы, связывающие их и инварианты;
- Разработать последовательность конкретных аппликативных модулей, A_HARBOUR1, A_HARBOUR2, и т.д., которые будут вводить аппликативный модуля для компонент "pool"и "berths";
- Разработать набор соответствующих императивных модулей начиная с последних аппликативных;
- Продумать любые улучшения производительности, которые могут быть сделаны в императивных модулях;
- Транслировать императивные модули в язык программирования.

Этот эскиз метода для конкретных приложений мы будем назвать планом разработки (development plan). На практике такие планы будут включать некоторое число других действий под документированию, тестированию, проверке качества (quality assurance), и т.д.

3.5 Модуль типов

Из наших начальных размышлений мы формулируем модуль TYPES:
scheme TYPES =
  class
    type
      Ship,
      Berth,
      Occupancy == empty | occupied_by(occupant : Ship),
      Index = $\{|\ \ i : \mathbf{Int} \bullet i \geq min \wedge max \geq i\ |\}$

    value
      min, max : **Int**,
      fits : Ship × Berth → **Bool**,
      indx : Berth → Index

    axiom
      [index_not_empty]
        max ≥ min,
      [berths_indexable]
        ∀ b1, b2 : Berth • indx(b1) = indx(b2) ⇒ b1 = b2
  end

Затем из модуля TYPES обьявляем глобальный объект : TYPES

object T : TYPES

3.6 Абстрактный аппликативный модуль

Чтобы создать модель системы мы следуем методу, который будет подробно изложен в секции 2.8.4.1 в [4]. Но так как мы описываем систе-

му, а не тип данных вроде очереди, мы будем формулировать свойства системы которые мы сможем специфицировать в текущий момент. Мы также сформулируем понятие целостности типа, которое выразим в виде предиката. Метод вкратце:

- Определить название проектируемого типа (типа интереса);
- Определить сигнатуры необходимых функций;
- Определить эти функции как функции перехода (generator), ([3]) если тип интереса (или тип зависящий от него) появляется в типе результата; или как функции выхода (observer) в другом случае. (Забегая вперед, скажем, что императивная версия функций состояния есть функция, которая меняет состояние). Мы обнаружили что имеем три генератора: arrives, docks, leaves; и указали два обсервера: wainting и occupancy;
- Сформулировать предусловия (predcondition) которые должны выполняться для каждой частичной функции. Все три генератора нашего примера являются частичными: существуют ситуации, когда их нельзя применять. Для этих ситуаций зададим функции защиты (guard) чтобы выразить это предусловия : can_arrive, etc. Все функции защиты должны быть выведены (то есть, им можно дать конкретное определение в терминах функций выхода) из функций выхода;
- Определить функцию целостности (consistent) типа, делая ее еще одной выводимой функций состояния;
- Для каждой возможной пары невыводимой функции состояния и невыводимой функции выхода определить аксиому, выражающую отношение между ними. Так как мы имеем три невыводимых функции состояния и две невыводимых функции выхода, то получаем шесть таких аксиом;
- добавить аксиомы выражающие представление о том, что невыводимые функции перехода поддерживают целостность типа. Мы имеем три такие аксиомы;
- Инкапсулировать функцию целостности типа (consistent) и все то, в чем не нуждаются клиенты модуля.

T

```
scheme A_HARBOUR0 =
  hide consistent in
  class
    type Harbour

    value
      /∗ generators ∗/
      arrives : T. Ship × Harbour ~→ Harbour,
      docks : T. Ship × T. Berth × Harbour ~→ Harbour,
      leaves : T. Ship × T. Berth × Harbour ~→ Harbour,
      /∗ observers ∗/
      waiting : T. Ship × Harbour → Bool,
      occupancy : T. Berth × Harbour → T. Occupancy,

      /∗ derived ∗/
      consistent : Harbour → Bool
      consistent(h) ≡
        (∀ s : T. Ship •
          ∼ (waiting(s, h) ∧ is_docked(s, h)) ∧
          (∀ b1, b2 : T. Berth •
              occupancy(b1, h) = T. occupied_by(s) ∧
              occupancy(b2, h) = T. occupied_by(s) ⇒ b1 = b2) ∧
          (∀ b : T. Berth •
              occupancy(b, h) = T. occupied_by(s) ⇒
                T. fits(s, b))),

      is_docked : T. Ship × Harbour → Bool
      is_docked(s, h) ≡
        (∃ b : T. Berth •
          occupancy(b, h) = T. occupied_by(s)),

      /∗ guards ∗/
      can_arrive : T. Ship × Harbour → Bool
      can_arrive(s, h) ≡
        ∼ waiting(s, h) ∧ ∼ is_docked(s, h),
```

```
can_dock : T. Ship × T. Berth × Harbour → Bool
can_dock(s, b, h) ≡
  waiting(s, h) ∧ ∼ is_docked(s, h) ∧
  occupancy(b, h) = T. empty ∧ T. fits(s, b),

can_leave : T. Ship × T. Berth × Harbour → Bool
can_leave(s, b, h) ≡
  occupancy(b, h) = T. occupied_by(s)

axiom
  [ waiting_arrives ]
    ∀ h : Harbour, s1, s2 : T. Ship •
      waiting(s2, arrives(s1, h)) ≡
        s1 = s2 ∨ waiting(s2, h)
      pre can_arrive(s1, h),
  [ waiting_docks ]
    ∀ h : Harbour, s1, s2 : T. Ship, b : T. Berth •
      waiting(s2, docks(s1, b, h)) ≡
        s1 ≠ s2 ∧ waiting(s2, h)
      pre can_dock(s1, b, h),
  [ waiting_leaves ]
    ∀ h : Harbour, s1, s2 : T. Ship, b : T. Berth •
      waiting(s2, leaves(s1, b, h)) ≡ waiting(s2, h)
      pre can_leave(s1, b, h),
  [ occupancy_arrives ]
    ∀ h : Harbour, s : T. Ship, b : T. Berth •
      occupancy(b, arrives(s, h)) ≡ occupancy(b, h)
      pre can_arrive(s, h),
  [ occupancy_docks ]
    ∀ h : Harbour, s : T. Ship, b1, b2 : T. Berth •
      occupancy(b2, docks(s, b1, h)) ≡
        if b1 = b2 then T. occupied_by(s)
        else occupancy(b2, h)
        end
      pre can_dock(s, b1, h),
  [ occupancy_leaves ]
    ∀ h : Harbour, s : T. Ship, b1, b2 : T. Berth •
      occupancy(b2, leaves(s, b1, h)) ≡
```

```
        if b1 = b2 then T. empty else occupancy(b2, h) end
      pre can_leave(s, b1, h),
  [ consistent_arrives ]
    ∀ h : Harbour, s : T. Ship •
      arrives(s, h) as h′ post consistent(h′ )
      pre consistent(h) ∧ can_arrive(s, h),
  [ consistent_docks ]
    ∀ h : Harbour, s : T. Ship, b : T. Berth •
      docks(s, b, h) as h′ post consistent(h′ )
      pre consistent(h) ∧ can_dock(s, b, h),
  [ consistent_leaves ]
    ∀ h : Harbour, s : T. Ship, b : T. Berth •
      leaves(s, b, h) as h′ post consistent(h′ )
      pre consistent(h) ∧ can_leave(s, b, h)
end
```

На практике обычно требуется несколько итераций прежде чем спецификация может быть разработана. В частности, в то время как функции переходов (generators) достаточно понятны из требований (судно может прибыть, причалить и т.д.) менее понятно, чем должны быть хорошие функции выхода (observers). Например, надо ли нам иметь функцию выхода для множество судов, находящихся на рейде? Если попробовать использовать такую функцию выхода, то вскоре станет понятно, что она может быть легко определенена как выведенная из простой функции выхода waiting, которую мы уже используем.

Мы также опустили такую константу типа Harbor как empty. Частично это произошло изза того, что требования молчали относительно начальных условий. На практике возможность инициализировать систему (и рестартировать ее) является требованием, таким образом константа empty должна понадобиться. Добавление константы empty (и соответствующие изменения в требованиях разработки) оставлено как упражнение для читателя.

3.6.1 Обсуждение аксиом

Этот раздел является не переводом, а комментариями переводчика.

3.6.1.1 Функции waiting, arrives.

Для любого состояния порта h и любых кораблей s1, s2: s2 находится на рейде после прибытия судна s1 (arrives(s1, h)) тогда и только тогда, когда s1 и s2 одно и тоже судно или s2 уже находился на рейде до прибытия судна s1. При условии, что s1 вообще может прибыть в порт.

```
---- File:waiting_arrives.r

  [waiting_arrives]
    all h : Harbour, s1, s2 : T.Ship :-
      waiting(s2, arrives(s1, h)) is s1 = s2 \/ waiting(s2, h)
        pre can_arrive(s1, h),

---- End Of File:waiting_arrives.r
```

3.6.1.2 Функции waiting, docks.

Для любого состояния порта h, причала b и любых кораблей s1, s2: s2 находится на рейде после причаливания судна s1 (docks(s1, h)) тогда и только тогда, когда s1 и s2 разные судна или s2 уже находился на рейде до причаливания судна s1.

```
---- File:waiting_docks.r

    [waiting_docks]
      all h : Harbour, s1, s2 : T.Ship, b : T.Berth :-
        waiting(s2, docks(s1, b, h)) is s1 ~= s2 /\ waiting(s2, h)
          pre can_dock(s1, b, h),

---- End Of File:waiting_docks.r
```

3.6.1.3 Функции waiting, leaves.

Для любого состояния порта h, причала b и любых кораблей s1, s2: уход судна s1 от причала b не изменяет факт наличия или отсутствия судна s2 на рейде.

```
---- File:waiting_leaves.r

          [waiting_leaves]
            all h : Harbour, s1, s2 : T.Ship, b : T.Berth :-
              waiting(s2, leaves(s1, b, h)) is waiting(s2, h)
                pre can_leave(s1, b, h),

---- End Of File:waiting_leaves.r
```

3.6.1.4 Функции occupancy, arrive.

Для любого состояния порта h, причала b и любого судна s: Прибытие судна на рейд не менят занятость причала h.

```
---- File:occupancy_arrive.r


   [occupancy_arrives]
     all h : Harbour, s : T.Ship, b : T.Berth :-
       occupancy(b, arrives(s, h)) is occupancy(b, h)
         pre can_arrive(s, h),

---- End Of File:occupancy_arrive.r
```

3.6.1.5 Функции occupancy, docks.

Для любых состояния порта h, причалов b1, и2 и судна s: после причаливания судна s к причалу b1 причал b2 занят судном s (если b1 = b2) или его состояние не изменилось.

```
---- File:occupancy_docks.r


   [occupancy_docks]
     all h : Harbour, s : T.Ship, b1, b2 : T.Berth :-
       occupancy(b2, docks(s, b1, h)) is
         if b1 = b2 then T.occupied_by(s) else occupancy(b2, h) end
         pre can_dock(s, b1, h),

---- End Of File:occupancy_docks.r
```

3.6.1.6 Функции occupancy, leaves.

Для любых состояния порта h, причалов b1, и2 и судна s: после ухода судно s от причала b1 причал b2 становится свободным (если b1 = b2) или его состояние не меняется.

```
---- File:occupancy_leaves.r


   [occupancy_leaves]
     all h : Harbour, s : T.Ship, b1, b2 : T.Berth :-
       occupancy(b2, leaves(s, b1, h)) is
         if b1 = b2 then T.empty else occupancy(b2, h) end
         pre can_leave(s, b1, h),

---- End Of File:occupancy_leaves.r
```

3.6.1.7 Условие целостности (consistent). Для порта h, находящегося в состоянии, удовлетворяющем условиям целостности consistent(s), прибытие, причаливание и уход судна переводит порт в новое состояние h', которое тоже удовлетворяет условиям целостности.

```
---- File:consistent.r


    [consistent_arrives]
      all h : Harbour, s : T.Ship :-
        arrives(s, h) as h'
          post consistent(h')
          pre consistent(h) /\ can_arrive(s, h),

   [consistent_docks]
     all h : Harbour, s : T.Ship, b : T.Berth :-
       docks(s, b, h) as h'
         post consistent(h')
         pre consistent(h) /\ can_dock(s, b, h),

   [consistent_leaves]
     all h : Harbour, s : T.Ship, b : T.Berth :-
       leaves(s, b, h) as h'
         post consistent(h')
         pre consistent(h) /\ can_leave(s, b, h)

---- End Of File:consistent.r
```

4 ASCII версии для символов RSL

Sym	ASCII	Sym	ASCII	Sym	ASCII
$\times$	><	*	-list	$^{\omega}$	-inflist
$\rightarrow$	->	$\overset{\sim}{\rightarrow}$	-~->	$\underset{m}{\rightarrow}$	-m->
$\overset{\sim}{\underset{m}{\rightarrow}}$	-~m->	$\leftrightarrow$	<->		
$\wedge$	/\	$\vee$	\/	$\Rightarrow$	=>
$\forall$	all	$\exists$	exists	$\bullet$	:-
$\Box$	always	$\equiv$	is	$\neq$	~=
$\leq$	<=	$\geq$	>=	$\uparrow$	**
$\in$	isin	$\notin$	~isin	$\subset$	<<
$\subseteq$	<<=	$\supset$	>>	$\supseteq$	>>=
$\cup$	union	$\cap$	inter	$\dagger$	!!
$\langle$	<.	$\rangle$	.>	$\mapsto$	+>
$\parallel$	||	$\#$	++	$[\,]$	|=|
$\sqcap$	|^|	λ	-\	$\circ$	#
$\vdash$	|-	$\preceq$	{=	$\sqsubseteq$	[=

Рис. 4: ASCII версия для символов RSL

Предметный указатель

axiom.r, 11

consistent.r, 20

fits.r, 7

generator , 2

hrb.r, 10

inv.r, 9

observer, 2
occupancy_arrive.r, 19
occupancy_docks.r, 19
occupancy_leaves.r, 19

waiting_arrives.r , 18
waiting_docks.r , 18
waiting_leaves.r , 18

ССЫЛКИ

[1] Бертран Мейер. Основы объектно-ориентированного программирования. Статические структуры: классы. http://www.intuit.ru/department/se/oopbases/7/. 1

[2] Бертран Мейер. Основы объектно-ориентированного программирования. Абстрактные типы данных (АТД). http://www.intuit.ru/department/se/oopbases/6/10.html. 1

[3] А.Г. Пискунов. Формализация парадигмы обьектно-ориентированного программирования: критика определения Гради Буча, 2007. http://i.com.ua/~agp1/ru/oopFormalizm.html. 1, 14

[4] The RAISE Method Group. The RAISE Development Method, 1999. http://users.iptelecom.net.ua/~agp1/arts/book.pdf. 1, 10, 13

[5] John Guttag. Abstract Data Types and the Development of Data Structures, 1977. Bern, June 1997 2001. 1

[6] A.G. Piskunov. Harbor: использование метода формальных спецификаций raise для разработки rdbms приложения. http://users.iptelecom.net.ua/~agp1/ru/hrb.html. 2