形 式 化 方 法:

基于 B 方法的严格软件开发

(9)

精化，数据精化

裘宗燕

北京大学数学学院信息科学系

2010年春季

概要

本次课讨论用 B 写出的系统规范的精化问题，内容包括：

- 为什么需要做精化
- 数据精化的概念
- 数据精化的一些相关问题
  - 简化信息的表示
  - 静态信息的继承和使用
  - 精化和结构化设施 **INCLUDES**, **SEES** 等的相互关系和作用

本次课主要讨论一些讨论和许多实例说明精化中的各种问题

有关精化的理论问题，在下次课讨论

精化

下面研究系统规范的精化（refinement）问题

精化是一种技术，也是软件形式化开发过程中的一项工作。其目标是将已经开发出的系统规范（抽象模型）变换到另一种更 “具体” 的模型

与原模型相比，这个新模型的“具体” 可能体现在两个方面：

1. 它可能包含更多与原来的系统需求和非形式化规范有关的细节
2. 它可能更接近系统的实现，其中的机制可能更接近实际计算机表示

六七十年代人们研究程序设计方法学，就提出了逐步求精的程序设计方法，实际上就是逐步精化。但在精化过程的前期阶段，许多描述是在自然语言或伪代码中给出的，没办法给出严格的语义 “维持性” 保证

形式化方法领域的 “精化” 概念可以看作是上述非形式化的 “精化” 的发展、系统化和严格化，其中特别严格地研究了 “语义保持性” 问题

在用 B 方法开发的过程中，被精化的是严格的抽象规范，精化的 “正确性”，即语义保持性，可以严格证明，为最后开发结果的正确性提供了坚实基础

抽象规范与计算机实现

基于 B 方法形式化开发的基本想法就是：先写出经过仔细检查和完全的形式化验证的规范，然后通过严格保证语义谓词的精化，最后做出有保证的系统

抽象机规范里通过一些数学结构描述了抽象机的状态信息，并把系统的功能性需求定义为与系统状态相关的一集操作

当抽象机的 “用户” 需要使用一部抽象机时，只需理解这一规范所描述的抽象机的行为，并不需要知道被操作的信息如何表示，与之相关的操作如何在计算机上实现。这也就是 “抽象机” 的 “抽象” 的主要意义

- 抽象机规范告诉我们这一机器将有什么样的行为
- 只要给相应操作正确的输入，就能导致所需的变换或得到所需的输出
- 只要一部实际机器符合这一规范，其具体实现方式与用户无关

抽象规范里使用了各种高级的数学结构（集合、关系、函数等），操作也是定义在这些数学结构上，常常是用高级的意义清晰的运算描述的

虽然这些结构意义清晰，表述严格，但与计算机实现层面的描述形式（程序设计语言的描述方式）相距甚远

抽象规范与计算机实现

首先，用户只关注功能，说明抽象规范里的各个操作都可以用完成同样工作的操作取代，用户不会有任何 “感觉"。所谓 “完成同样工作"，包括完成同样的状态转换，在同样状态下对同样输入参数产生同样的输出。这是 “操作精化"。操作精化还有其他作用，下次课考虑

另一方面，用户关心的功能，与抽象机器里信息的具体表示无关。我们完全可以（而且需要）用更接近计算机实际的表示形式代替抽象的数学形式，并相应修改操作的实现。这项工作就是本节课的主题

前面说 “只关注功能" 是用户的观点。程序员的观点与此不同，他们不仅关注系统的功能，还希望自己构造的系统能充分利用计算机的能力，而适合用户的描述往往不能有效映射到计算机实现。实现者要考虑：

- 将高级数学结构映射到计算机表示，需要做出许多设计决策，需要根据实际情况选择最合适的表示方式，一般而言这一过程无法自动化
- 与之对应，将抽象操作映射到计算机操作，也有实现方法的问题

同样抽象结构和抽象操作，可以有许许多多可能的精化和实现方式，需要在精化中考虑和选择，需要做出许多设计决策

数据精化（Data Refinement）

将抽象的数学结构变换为具体的计算机表示的过程（和工作）称为 “数据精化"。显然，这一变换过程还牵涉到相关操作的变换

在 B 方法里，数据表示（将抽象数据映射到具体的计算机表达形式）的工作也是通过抽象机的 “精化" 的方式进行的，每步精化实现一些设计决策，描述一些 “抽象" 结构如何表示为更具体的更接近实现的结构

为建立抽象规范与其精化后的规范之间的联系，需要用 “连接不变式"（linking invariant）建立抽象状态和精化后的状态之间的关联，还需要说明在新的数据表示形式上如何完成初始化和各种操作

一个精化机器的界面应该和被它精化的机器的界面完全一样，具有相同的操作集合，每个操作具有同样的输入和输出参数

在实际的复杂系统的开发中，这种精化可以做许多步，一步步地将软件的抽象规范落实到具体的计算机实现，并证明每步精化的连接不变式。这样就可能更好地保证了系统的最终实现能满足需要

下面考虑一个实例

实例：足球队管理

考虑比赛中一个足球队的队员管理。一个足球队有 22 名队员，足球赛中同时上场的为 11 人，但比赛中可以换人。现在要写一个规范描述这种过程

设球员的球衣编号为 1 到 22，用编号代表球员，球队的上场球员就是 1..22 的一个 11 个元素的子集。初始时假定 1..11 号球员上场，定义两个操作：

- substitute 是换人操作，下一个上一个
- query 查询某个球员是否在场上，用一个集合表示查询结果

描述这一抽象机，头部是

MACHINE
$Team$
SETS $ANSWER = \{in, out\}$
VARIABLES $team$
INVARIANT $team \subseteq 1..22 \wedge \mathsf{card}(team) = 11$
INITIALISATION $team := 1..11$

下面定义两个操作

足球队管理

OPERATIONS
$substitute(psn, rep) =$
PRE $psn \in 1..22 \wedge rep \in 1..22 \wedge psn \in team \wedge rep \notin team$
THEN $team := (team - \{psn\}) \cup \{rep\}$
END;
$ans \longleftarrow query(psn) =$
PRE $psn \in 1..22$
THEN IF $psn \in team$
THEN $ans := in$
ELSE $ans := out$
END
END
END

作为这一抽象机的精化，应该维护 $team$ 的信息变化轨迹，这样同样能执行换人和在场队员查询，也就是说，保持抽象机 $Team$ 的操作仍然有效

下面的想法是用一个数组 $teamr$ 代替集合 $team$ 保存上场队员的轨迹，其类型是 $teamr \in 1..11 \rightarrowtail 1..22$，做出的精化机器在下页给出

足球队管理：精化

REFINEMENT
$Team\_Ref$
REFINES
$Team$
VARIABLES $teamr$
INVARIANT $teamr \in 1..11 \rightarrowtail 1..22 \wedge \mathsf{ran}(teamr) = team$
INITIALISATION $teamr := \lambda\, psn \cdot (psn \in 1..11 \mid psn)$
OPERATIONS
$substitute(psn, rep) = teamr(teamr^{-1}(psn)) := rep$;

$ans \longleftarrow query(psn) =$
  IF $psn \in \mathsf{ran}(teamr)$
  THEN $ans := in$
  ELSE $ans := out$ **END**
END

$\mathsf{ran}(teamr) = team$ 是连接不变式，它建立其 "更具体的变量" $teamr$ 和抽象变量 $team$ 之间的联系。对集合 $team$ 的任何取值情况，$teamr$ 的值域总等于这个集合（精化机器的状态和被精化机器的状态严格对应）

足球队管理：精化

精化机器的初始化也要与原机器对应。对于原机器，初始化语句是

$$team := 1..11$$

新机器应该是一个将 $1..11$ 映射到 $1..11$ 的函数。前面描述中采用 λ 记法

$$teamr := \lambda\, psn \cdot (psn \in 1..11 \mid psn)$$

对应的正文形式是

```
teamr := % psn . (psn : 1..11 | psn)
```

显然这一初始化可以有许多不同写法，例如

$$teamr := \{i \mapsto i \mid i \in 1..11\}$$

或者更简单的

$$teamr := \mathsf{id}(1..11)$$

λ 写法是一种很常见也方便的函数描述方式，应该熟悉（一些今天流行的编程语言也引进了这种记法形式，如 Java 和 C#等）

抽象机的操作也需要精化，使之能在新的数据表示上正确操作

足球队管理：精化

原抽象机操作 $substitute$ 定义的主要部分

$$team := (team - \{psn\}) \cup \{rep\}$$

在精化后的抽象机操作里变成了

$$teamr(teamr^{-1}(psn)) := rep$$

另一个操作 $query$ 的情况与此类似，查看是否属于集合，精化后变成查看是否属于函数（数组）$teamr$ 的值域

精化操作取同样输入产生同样输出。如原操作修改状态，新操作与之对应

我们可以得到交换图：

$$\begin{array}{ccc} s_a & \overset{\sqsubseteq}{\longrightarrow} & s_r \\ {\scriptstyle op_a}\downarrow & & \downarrow{\scriptstyle op_r} \\ s'_a & \underset{\sqsubseteq}{\longrightarrow} & s'_r \end{array}$$

其中 $\sqsubseteq$ 表示精化关系

其中抽象操作 op_a 将抽象状态 s_a 变换到状态 s'_a，右边的 s_r 和 s'_r 表示与之对应的精化后操作 op_r 导致的相应状态变化。两对状态相互对应连接

足球队管理：另一种精化

精化得到的机器已比较接近可执行代码。但也可能还有些部分不能执行。上例里的 $psn \in ran(teamr)$ 和 $teamr^{-1}(psn)$ 都是比较典型的例子，可能需要用两个过程来实现它们，这可以通过进一步精化步骤来完成

一般而言，可能有许多不同的方式去精化一部抽象机。具体来说，前面给出的只是一种精化抽象机 $Team$ 的方式，还有许多可能方式

因此，每步精化都包含一个重要的设计选择，需要做出合理的设计决策

举例说，前面的 $Team$ 也完全可能采用其他的具体数据表示。下面考虑用另一种数据表示，用一个 22 元的数组 $teama$ 表示 $team$，其中每个元素的值取自集合 $\{in, out\}$，表示穿相应号码球衣的队员是否在场上

- 如果穿 psn 球衣的队员在场上，$teama(psn) = in$
- 如果穿 psn 球衣的队员不在场上，则 $teama(psn) = out$

现在连接不变式是 $\{in\}$ 的逆像等于 $team$，可以描述为

$$team = teama^{-1}[\{in\}] \quad 或者 \quad team = \mathsf{dom}(teama \rhd \{in\})$$

足球队管理：另一种精化

初始化子句可以写为

$$teama := \{i \mapsto in \mid i \in 1..11\} \cup \{j \mapsto out \mid j \in 12..22\} \text{ 或}$$

$$teama := (1..11) \times \{in\} \cup (12..22) \times \{out\}$$

按这一设计做出的精化如下：

REFINEMENT $TeamaRef$
REFINES $Team$
VARIABLES $teama$
INVARIANT $teama \in (1..22) \rightarrow ANSWER \wedge$
$dom(teama \rhd \{in\}) = team$
INITIALISATION $teama := (1..11) \times in \cup (12..22) \times out$
OPERATIONS
$substitute(psn, rep) =$
BEGIN $teama(psn) := out; teama(rep) := in$ **END;**
$ans \longleftarrow query(psn) = ans := teama(psn)$
END

操作实现直截了当，第二个操作非常简单，函数值就是表示球员的上场状态

简化信息表示

精化关系是不同抽象层次的机器之间的关系，通过精化得到的机器应维持足够的信息，使原规范的各种操作都能正常执行

但这并不要求原机器的所有状态都一一对应到精化后的机器里，因为外部看不到操作机器的状态，看到的只是操作。只要能保证操作的正确行为和效果，精化机器里完全可以用任何适当的方式表示内部信息

在初始的抽象规范里，可能包含一些为使描述更加清晰易读易理解易验证性质的信息，但并不必要，精化中可以用更简单的方式来表示这些信息

改变或简化信息的表示方式的基本要求是保证抽象机的操作都能正确实现，这同样是设计选择和决策问题——抽象机里并不包含它能否简化的信息

下面通过一个例子说明这方面的情况

考虑一部记录考试成绩的抽象机，它能基于成绩记录做一些简单统计

成绩记录简单地用一个集合表示，抽象机的规范很简单（见下页，这里只是为了说明问题，并不是要做一个有用的系统）

成绩统计抽象机

MACHINE $Exam$
SETS $STUDENTS$
VARIABLES $scores$
INVARIANT $scores \in STUDENTS \pfun 0..100$
INITIALISATION $scores := \varnothing$
OPERATIONS
$enter(st, sc) =$
PRE $st \in STUDENTS \land st \notin \mathrm{dom}(scores) \land sc \in 0..100$
THEN $scores(st) := sc$
END;
$mean \longleftarrow average =$
PRE $scores \neq \varnothing$
THEN $mean := \Sigma st \cdot (st \in \mathrm{dom}(scores) | scores(st)) / \mathrm{card}(scores)$
END;
$num \longleftarrow number = num := \mathrm{card}(scores)$
END

成绩统计抽象机：精化

从功能看，这一抽象机只提供成绩输入，统计平均值和有成绩人数三个操作。因此，系统需要维持的信息只需保证这几个操作能正常完成，完全可以采用任何满足需要的信息记录和维护方式

分析这一系统规范，可以看到一些情况：

- 需要产生的输出并不涉及学生个人或标识
- 成绩输入操作的前条件要求被输入学生的标识没出现在记录中，这实际上要求操作的调用方保证调用时确实不出现这种情况。对于本抽象机是否保存了学生标识并没有要求

在这种情况下，这个抽象机里是否保存学生标识并不是必需的

如果只要求输入、求平均和总成绩项数，完全可以还一种简单表示方式：

- 用一个变量 $total$ 表示累积的成绩值
- 用一个变量 $numb$ 表示成绩项数

这些信息足以支持抽象机完成所需计算

成绩统计抽象机：精化

基于这些考虑做出精化机器如下：

REFINEMENT $ExamRef$
REFINES $Exam$
VARIABLES $total, numb$
INVARIANT $numb \in NAT \wedge total \in NAT \wedge numb = \mathsf{card}(scores)$
$total = \Sigma st \cdot (st \in \mathsf{dom}(scores) | scores(st))$
INITIALISATION $= total, numb := 0, 0$
OPERATIONS
$enter(st, sc) =$ **BEGIN** $total, numb := total + sc, numb + 1$ **END**;
$mean \longleftarrow average = mean := total/numb$;
$num \longleftarrow number = num := numb$
END

- 原规范里 $average$ 的前条件保证精化操作中不会出现除 0 的情况，如果没有输入过成绩，这一操作不会被调用
- 操作 $enter$ 的前条件保证调用方不会对同一学生重复调用 $enter$，模块实现者不必检查这一条件。如果没有这一前条件，这个精化就不正确了

成绩统计抽象机：精化

看看这个精化，它的一个状态对应着原规范的许许多多状态

例如状态 $total = 255, numb = 3$ 对应着原抽象机的各种各样的包含三个学生的成绩，且他们的成绩之和等于 255 的状态

对不同的抽象状态，操作 $enter$ 对不同的满足其前条件的输入参数会做不同计算，但对于这一抽象机的功能而言，抽象机器的状态差异并不重要。在精化机器上，它们表现为同样的行为

另一方面，为了考察 $enter$ 的前条件，精化机器的操作还需要追溯到抽象规范，去考虑 $enter$ 的前条件，这一前条件只有通过 $scores$ 才能说清楚。这一点也表现出抽象规范的基本作用 —— 帮助检查各种基本的一致性，也说明了具体变量（$total$ 和 $numb$）实际表示的是什么

用后来给出的精化抽象机作为原始规范实际上是不正确的：虽然它比较简单，但却没有完整地描述系统的需要和行为

最后：精化得到的这一机器还不是最终实现，其中的同时代换还需进一步精化为两个顺序代换，实现为两个顺序赋值

继承和使用静态信息

精化机器可以访问原抽象机的所有静态部分，包括参数、集合和常量，还可访问被原抽象机包含的机器里的这些部分，因为它们就是原抽象机的部分

但是精化抽象机不能访问原抽象机 **SEES** 的机器。如果真需要访问某些机器，精化机器里就必须明确用 **SEES** 子句说明。另一方面，精化机器不能用 **USES** 子句，因为 **USES** 就是为了建立抽象规范之间的状态关联

精化就是要说明用什么样的更具体的机器去满足抽象规范。从这一角度看，规范里的所有静态信息都应该是精化机器可用的，因为这些信息与机器的运行无关，只在处理（编译）时可用

但精化机器不应该去访问抽象机器的状态，因为精化机器需要

- 建立自己的状态去替代抽象规范的状态，说明两种状态之间的联系
- 在新的状态上建立自己的操作，说明它们怎样实现抽象的操作规范

下面通过一个例子，说明精化机器如何继承和使用抽象规范的静态信息

实例：熨衣店

假定熨衣店有一叠待熨衣物，新来的衣物放在上面，最上面的衣物先熨烫处理（一个栈）。假定衣物不能堆积太多，有一个 $limit$。写出的规范：

MACHINE $Ironing$
SETS $ITEM$
CONSTANTS $limit$
PROPERTIES $limit \in \mathbb{N}$
VARIABLES $pile$
INVARIANT $pile \in \mathsf{seq}(ITEM) \wedge \mathsf{size}(pile) \leq limit$
INITIALISATION $pile := \varnothing$
OPERATIONS
$put(it) =$
PRE $it \in ITEM \wedge \mathsf{size}(pile) < limit$ **THEN** $pile := pile \leftarrow it$ **END;**
$it \longleftarrow take =$
PRE $pile/ = [\,]$ **THEN** $pile := \mathsf{front}(pile) \parallel it := \mathsf{last}(pile)$ **END;**
$res \longleftarrow query(it) =$
PRE $it \in ITEM$ **THEN** $res := \mathsf{bool}(it \in \mathsf{ran}(pile))$ **END**
END

实例：熨衣店

考虑上面机器的精化：

- 用一个数组 $pilearr$（代替序列）保存待熨的衣物
- 用一个变量 $counter$ 记录堆中的衣物件数
- 总把衣物存放在 $pilearr$ 里位置 $1..counter$
- $counter$ 随着衣物的加入和取出而增大或减小，以 $limit$ 为限

这以工作实际上就是用数组来实现一个栈

连接不变式：数组 $pilearr$ 里的最前面一段（位置为 $1..counter$ 的一段）就等于原来的序列 $pile$

操作实现为修改 $counter$ 值和数组元素的存取

实例：熨衣店

精化的机器：

REFINEMENT $IroningRef$
REFINES $Ironing$
VARIABLES $pilearr, counter$
INVARIANT $pilearr \in 1..limit \pfun ITEM \land counter \in 0..limit \land$
$(1..counter \domres pilearr) = pile \land counter = \mathsf{size}(pile)$
INITIALISATION $pilearr := [\,] \parallel counter := 0$ /* [ ] 相当于 $\varnothing$ */
OPERATIONS
$put(it) =$
BEGIN $counter := counter + 1; pilearr(counter) := it$ **END;**
$it \longleftarrow take =$
BEGIN $it := pilearr(counter); counter := counter - 1$ **END;**
$res \longleftarrow query(it) =$
IF $it \in ITEM \land it \in pilearr[1..counter]$
THEN $res :=$ **TRUE ELSE** $res :=$ **FALSE END**
END

这里的连接不变式对 $1..counter$ 之外的数组元素没有任何限制

INCLUDES、SEES 和精化

精化机器也可以用 B 方法的各种结构化机制构造，包括 **INCLUDES**、**SEES**等。完全可以 **INCLUDES** 或者 **SEES** 一些已经独立开发和验证过的机器

构造中只允许包含抽象机器（不允许包含其他精化机器），因为抽象机器仅仅是需要实现的行为的抽象规范，并不包含如何实现这些行为的描述

在精化机器里，可以 **INCLUDES** 或 **EXTENDS** 抽象机器，可以 **PROMOTES** 抽象机器的操作，可以 **SEES** 抽象机器

但精化机器里不允许写 **USES** 子句。这样规定的原因是，**USES** 要建立两个机器之间的状态联系，在这里不适用

精化机器里的连接不变式需要连接它的所有状态变量与抽象状态里的变量，包括被 **INCLUDES** 或 **EXTENDS** 的机器的变量。被 **SEES** 的机器的状态不需要连接，因为它不是本精化机器的一部分

在精化机器里，同样可以直接读被它 **INCLUDES**、**EXTENDS** 或 **SEES** 的机器的状态，在操作描述里使用这些状态，也同样不能修改这些状态

实例：港口系统

考虑一个港口管理系统，它维持一批货轮的到达、等待和入/出码头的情况

- 有一集货轮 $SHIP$，它们到达港口后排在一个序列 $waiting$ 里等待入港
- 港口有一集码头 $QUAY$，每个码头每次可以接一条货轮
- 操作包括到港、进码头和离开等
- 另用一个函数 $docked$ 表示所有在码头作业的货轮

基本抽象机 $Port$ 的静态部分可以描述为：

MACHINE
$Port$**SETS** $SHIP; QUAY$
VARIABLES $waiting, docked$
INVARIANT $waiting \in \mathrm{iseq}(SHIP) \wedge docked \in QUAY \rightarrowtail\!\!\!\!\!\!\!\!+\ SHIP \wedge$
$\mathrm{ran}(waiting) \cap \mathrm{ran}(docked) = \varnothing$
INITIALISATION $waiting := [\,] \,||\, docked := \varnothing$

不变式说明，一条货轮不会同时正在等待和码头作业

港口抽象机：操作

港口抽象机的操作很容易定义：

OPERATIONS

$arrive(sp) =$ /* 到港的货轮不会是已经在等待或港口作业的 */
  PRE $sp \in SHIP \wedge sp \notin ran(waiting) \wedge sp \notin ran(docked)$
  THEN $waiting := waiting \leftarrow sp$ **END;**

$dock(qy) =$ /* 有等待货轮且码头 qy 空闲，才能进码头一只船 */
  PRE $waiting \neq [\,] \wedge qy \in QUAY \wedge qy \notin dom(docked)$
  THEN
    $docked(qy) := first(waiting) \parallel waiting := tail(waiting)$
  END;

$qy \longleftarrow leave(sp) =$
  PRE $sp \in SHIP \wedge sp \in \mathsf{ran}(docked)$
  THEN $docked := docked \rhd \{sp\} \parallel qy := docked^{-1}(sp)$ **END;**

$num \longleftarrow numberwaiting = num := \mathsf{size}(waiting)$

END

这些操作的意义都很简单

港口系统：通用模块 Queue

下面考虑 $Port$ 的精化，这里准备先开发两个通用的抽象机，基于它们去精化 $Port$。首先是通用抽象机 $Queue$，它实现一个队列数据结构

- 支持以 "先进性出" 的方式保存一批 "数据元素"。为此给抽象机引入一个集合参数 $ELEMENT$，使之成为一个通用的模块，可以用到任何需要这种服务的系统规范里
- 在 $Port$ 的实现里（精化里），用 $Queue$ 的实例保存正在等待的货轮（$SHIP$ 类型的元素），支持对等待货轮的操作

MACHINE $Queue(ELEMENT)$
VARIABLES $list$
INVARIANT $list \in \mathsf{seq}(ELEMENT)$
INITIALISATION $list := [\,]$
OPERATIONS
  $add(elem) =$ **PRE** $elem \in ELEMENT$ **THEN** $list := list \leftarrow elem$ **END;**
  $el \longleftarrow take =$
    PRE $list \neq [\,]$ **THEN** $el := \mathsf{first}(list) \parallel list := \mathsf{tail}(list)$ **END**
END

港口系统：通用模块 Map

定义通用抽象机 Map，这是一个通用的 "字典" 抽象机

- 为它定义（集合）参数 $INDEX$ 和 $ITEM$，表示字典的 "索引" 类型和 "数据项" 类型。这一抽象机也可以在许多规范中使用
- 在 $Port$ 的实现里（精化里），用 Map 的实例记录码头和正在码头作业的货轮之间的对应关系

MACHINE $Map(INDEX, ITEM)$
VARIABLES $func$
INVARIANT $func \in INDEX \pfun ITEM$
INITIALISATION $func := \varnothing$
OPERATIONS
$insert(ind, itm) =$
PRE $ind \in INDEX \land itm \in ITEM$ **THEN** $func(ind) := itm$ **END;**
$remove(ind) =$ **PRE** $ind \in INDEX$ **THEN** $func := ind \domsub func$ **END;**
$it \longleftarrow query(ind) =$
PRE $ind \in INDEX \land ind \in dom(func)$ **THEN** $it := func(ind)$ **END**
END

港口系统：精化

现在考虑 $Port$ 的精化，其静态部分见下：

REFINEMENT $PortRef$
REFINES $Port$
INCLUDES $qu.Queue(SHIP), mp.Map(SHIP, QUAY)$
VARIABLES $number$
INVARIANT $waiting = qu.list \land docked^{-1} = mp.func \land$
$number = \mathsf{size}(waiting)$
INITIALISATION $number := 0$

这里 **INCLUDES** 两个抽象机 $Queue$ 和 Map，但没有 **PROMOTES** 它们的操作，这就是说，将这两个抽象机作为实现自己功能的内部构件

注意：在 **INCLUDES** 两个组件时做了重命名。这并不必要，直接 **INCLUDES** 也不会出现名字冲突。这里重命名是为提高规范的清晰性和可读性

重命名后再用组件里的成分，描述麻烦了，都需要加前缀，但

- 每次写前缀迫使我们想清楚要写的是什么，可以帮助避免无意识地犯错
- 读规范的人得到了实惠，他们更容易看清规范里引用的是哪里的成分

港口系统：精化

精化里的新变量 $number$，需要在不变式里描述其类型，这里是通过 size 的返回值 "隐式" 说明了它的类型

不变式还需要连接精化机器的变量和被精化的抽象机器的变量。由于这里 **INCLUDES** 了 $Queue$ 和 Map，它们的变量也是精化机器里的变量

不变式：

$waiting = qu.list \land docked^{-1} = mp.func \land number = \mathsf{size}(waiting)$

描述了不同抽象层次的规范里的变量之间的所有关系

由于函数 $leave$ 需要从停泊的货轮找到它所在的码头，这里把用的 Map 实例 mp 定义为从 $SHIP$ 到 $QUAY$ 的函数，以方便这一查找

由于已将 $docked$ 定义为单射，这就保证了 mp 里的 $list$ 一定是函数。这里也不必再将其定义为单射（因为它有逆函数，自然是），有关的性质已经由抽象规范保证了，精化机器里的变量更新一定保证抽象规范里描述的性质

这里定义 $number$ 等于抽象的序列变量 $waiting$ 的长度，完全可以将它定义为等于 $qu.list$ 的长度，没有什么差别

港口系统：精化

精化机器的操作也是基于组件操作定义的：

OPERATIONS

$arrive(sp) =$
  BEGIN $qu.add(sp); number := number + 1$ **END;**
$dock(qy) =$
  BEGIN
    VAR sp **IN** $sp \longleftarrow qu.take; mp.insert(sp, qy)$ **END;**
    $number := number - 1$
  END;
$qy \longleftarrow leave(sp) =$
  BEGIN $qy \longleftarrow mp.query(sp); mp.remove(sp)$ **END;**
$num \longleftarrow numberwaiting = num := number$

END

重命名使人更容易看懂这个规范。如果抽象机规模更大，**INCLUDES** 许多抽象机，我们更应该仔细为每个组件抽象机引进合适的名字

这是从程序设计和软件工程实践中学来的。在写复杂规范时，写得尽可能清晰易读也很重要，也应注意引进必要的名字，选择适当的名字

港口系统：精化

在这个精化的多处用了顺序代换（代换的顺序复合）。有一处是必须的：

$$\textbf{VAR } sp \textbf{ IN } sp \longleftarrow qu.take; mp.insert(sp, qy) \textbf{ END}$$

其余都可以写成并行代换

精化是系统开发的后续步骤。此时人们已不再强调操作的原子性，开始认为一个操作可能通过一系列顺序步骤完成

在 $leave$ 的新定义里可以明显看到这种情况：其中两次调用 Map 的操作：首先用 $query$ 取得一个元素，而后用 $remove$ 删除一项

由于前面的 $Queue$ 和 Map 还不是实现，这个 $PortRef$ 也还不是可执行的规范描述。进一步开发就要求精化整个规范（包括通过 **INCLUDES** 包含进来的部分），而且要将它作为一个整体来考虑

实际中人们通常不这样做，而是采用另外的方法

后面讨论 implementation 时会介绍 B 方法的其他支持结构化开发的机制，利用它们可以独立地精化被 **INCLUDES** 的各抽象机。为此要引进一些新约束

实例：城际交通

下面用一个实例来展示一步步精化的开发过程。这里用的实例很简单，是城际交通连接的一个抽象模型，实际上是维护一个无向图的连接关系

下面开发的抽象机有一个集合参数 $CITY$，表示系统关注的城市集合，系统里维护的状态 $roads$ 表示连接城市的道路

抽象地看，道路就是 $CITY$ 到自身的一个关系 $roads \in CITY \leftrightarrow CITY$。初始时没有城市之间的连接，$roads$ 取值空集

抽象机的操作包括加入新的道路，以及查询两个城市之间是否有道路连通

加入新路就是修改城市之间的连接关系 $roads$。城市之间的直接或间接连通关系可以用复杂的数学关系表示：

$$ct1 = ct2 \vee (ct1, ct2) \in (roads \cup roads^{-1})^*$$

这里 $(roads \cup roads^{-1})^*$ 表示关系 $(roads \cup roads^{-1})$ 的自反传递闭包。直观说，从城市 $ct1$ 可到达 $ct2$，如果存在经过一些城市的路径连接它们

基于这些想法，可以写出系统的初始抽象规范

实例：城际交通

MACHINE $Cities(CITY)$
SETS $ANSWER = connected, notconnected$
VARIABLES $roads$
INVARIANT $roads \in CITY \leftrightarrow CITY$
INITIALISATION $roads := \varnothing$
OPERATIONS
$link(ct1, ct2) =$
PRE $ct1 \in CITY \wedge ct2 \in CITY$
THEN $roads := roads \cup \{ct1 \mapsto ct2\}$ **END;**
$ans \longleftarrow connectedQ(ct1, ct2) =$
PRE $ct1 \in CITY \wedge ct2 \in CITY$
THEN IF $ct1 = ct2 \vee ct1 \mapsto ct2 \in (roads \cup roads^{-1})^*$
THEN $ans := connected$
ELSE $ans := notconnected$
END
END
END

城际交通：第一步精化

操作 $link$ 非常简单，复杂的就是操作 $connectedQ$ 里 **IF** 的条件

$$ct1 = ct2 \vee (ct1, ct2) \in (roads \cup roads^{-1})^*$$

概念清晰而正确，但其中用到复杂的 "自反传递闭包" 和意义深刻的 $(\bullet)^*$ 操作（"Kleen 星号" 操作）。这一结构显然不是可执行的

下面的精化就是想消去这个复杂概念，用更面向计算的概念代替它

第一步数据精化的想法：在系统的每个状态，城市集合 $CITY$ 都被道路连接关系划分为一些部分，形成 $CITY$ 的一个划分（partition），$CITY$ 的每个元素属于且仅属于其中的一个部分（图论里称为 "连通分支"）

第一步精化里想用一对变量来表示系统中 $CITY$ 集合的划分状态：

- 变量 $partition$ 是 $CITY$ 的子集的集合，其任何时刻的值就表示 $CITY$ 的一个划分，属于 $partition$ 的同一个元素（$CITY$ 的一个子集）的城市相互连通，属于不同元素的城市互不可达
- 变量 $class$ 是从各城市到它所属的 $partition$ 元素的映射，根据它可以方便地确定任何一个城市属于哪个连通部分

城际交通：第一步精化

例：设城市用整数标号，共 9 个城市，当前情况是其中的 1, 3, 8 互通，2, 4, 5, 9 互通，6, 7 互通，表示这个状态的变量取值是：

$$
\begin{aligned}
partition &= \{\{1,3,8\},\{2,4,5,9\},\{6,7\}\} \\
class &= \{(1,\{1,3,8\}),(3,\{1,3,8\}),(8,\{1,3,8\}),\\
&\quad (2,\{2,4,5,9\}),(4,\{2,4,5,9\}),(5,\{2,4,5,9\}),\\
&\quad (9,\{2,4,5,9\}),(6,\{6,7\}),(7,\{6,7\})\}
\end{aligned}
$$

采用这种新的状态表示以后，判断两个城市间是否互通就非常简单了：只需将 $class$ 应用于这两个城市，看得到的是不是同一个子集

link 的实现复杂一些，因为要维护两个状态变量（下面给出并解释）

系统的初始状态是什么？不难看到，情况应该是

- $partition$ 是 $CITY$ 的所有单元素子集的集合
- $class$ 将 $CITY$ 里的每个城市映射到只包含它自身的那个单元素集

下面先看精化抽象机的静态部分

城际交通：精化的静态部分

REFINEMENT $\boldsymbol{CitiesRef}(CITY)$ /* 精化也要有同样参数 */
REFINES $\boldsymbol{Cities}$
VARIABLES $partition, class$
INVARIANT $partition \subseteq \mathbb{P}(CITY) \wedge class \in CITY \rightarrow \mathbb{P}(CITY) \wedge$
$\bigcup cs.(cs \in partition | cs) = CITY \wedge$
$\forall (cs, ds).(cs \in partition \wedge ds \in partition \Rightarrow (cs = ds \vee cs \cap ds = \varnothing)) \wedge$
$\forall ct.(ct \in CITY \Rightarrow \mathsf{closure}(roads \cup roads^{-1} \cup \mathsf{id}(CITY))[\{ct\}] \in partition) \wedge$
$\forall ct.(ct \in CITY \Rightarrow ct \in class(ct)) \wedge \mathsf{ran}(class) = partition \wedge$
$\forall cs.(cs \in partition \Rightarrow class^{-1}[\{cs\}] = cs)$
INITIALISATION $partition := \{cs | cs \in \mathbb{P}(CITY) \wedge \mathsf{card}(cs) = 1\} \;||$
$class := \{ct, cs | ct \in CITY \wedge cs = \{ct\}\}$

INVARIANT 的

- 第 2, 3 行说明 $partition$ 是 $CITY$ 的划分
- 第 4 行说每个 $ct \in CITY$ 经过 $roads$ 的自反传递闭包是 $partition$ 的元素
- 第 4, 6 行描述 $partition$ 和 $class$ 的关系

注意初始化的写法，它正确建立了两个变量的初值

城际交通：精化的操作

OPERATIONS

$\boldsymbol{link}(ct1, ct2) =$

IF $class(ct1) \neq class(ct2)$

THEN

$partition := partition - \{class(ct1), class(ct2)\}$
$\cup \{class(ct1) \cup class(ct2)\};$

$class := class \Leftarrow ((class(ct1) \cup class(ct2)) *$
$\{class(ct1) \cup class(ct2)\})$

END;

$\boldsymbol{ans} \longleftarrow \boldsymbol{connectedQ}(ct1, ct2) =$

IF $class(ct1) = class(ct2)$

THEN $\boldsymbol{ans := connected}$

ELSE $\boldsymbol{ans := notconnected}$ **END**

END

操作 $\boldsymbol{link}$ 维护两个变量的值

- 对 $partition$，去掉 $ct1$ 和 $ct1$ 所在的子集，加上它们的并集
- 对 $class$，让 $ct1$ 和 $ct1$ 所在的子集里的元素都映射到它们的并集

城际交通：划分集到代表元

经过一步精化，复杂的判断条件移到不变式里，不需要实现了。但现在操作还不可执行：$\boldsymbol{link}$ 里有复杂的集合操作，$\boldsymbol{connectedQ}$ 要判断集合相等

下面考虑进一步改变机器的状态表示，不再维护一组集合（即 $partition$）以及城市与集合的关系（$class$），而是考虑另一做法：

- 在划分的每个子集合里选一个（确定的）代表元
- 维护一个映射，把每个元素映射到它所在的划分类的代表元

要实现这种基本想法，需要处理下面两个问题：

- 在连接两个城市时，需要维护代表元关系，为新形成的集合中的所有元素设置统一的代表元，为此问题找到一种有效方法
- 要检查两个城市是否互通，只需检查它们关联到的代表元是不是同一个元素。这一工作现在变得非常简单

作为初始状态，每个城市的代表元是自己

城际交通：代表元

新的精化机器的静态描述部分是

REFINEMENT
$\boldsymbol{CitiesRefR}(CITY)$
REFINES
$\boldsymbol{CitiesRef}$
VARIABLES $\boldsymbol{rep}$
INVARIANT $\boldsymbol{rep} \in CITY \rightarrow CITY \wedge$
$\forall (\boldsymbol{ct1}, \boldsymbol{ct2}) \cdot (\boldsymbol{ct1} \in CITY \wedge \boldsymbol{ct2} \in CITY \Rightarrow$
$((class(\boldsymbol{ct1}) = class(\boldsymbol{ct1})) \Leftrightarrow (\boldsymbol{rep}(\boldsymbol{ct1}) = \boldsymbol{rep}(\boldsymbol{ct2}))))$
INITIALISATION $\boldsymbol{rep} := \mathsf{id}(CITY)$

注意这里的连接不变式，它说的是：如果两个城市属于同一个划分类（在前一精化机器里），那么它们的代表元相同（在这个机器里）

操作实现的关键是 $\boldsymbol{link}$ 两个城市时做相关城市的代表元更新。想法是：两个类合并，让合并后的类里的所有城市都以原来一个类的代表元作为代表元

城际交通：代表元

OPERATIONS
$\boldsymbol{link}(\boldsymbol{ct1}, \boldsymbol{ct2}) =$
IF $\boldsymbol{rep}(\boldsymbol{ct1}) \neq \boldsymbol{rep}(\boldsymbol{ct2})$
THEN $\boldsymbol{rep} := \boldsymbol{rep} \Leftarrow \boldsymbol{rep}^{-1}[\{\boldsymbol{rep}(\boldsymbol{ct1})\}] \times \{\boldsymbol{rep}(\boldsymbol{ct2})\}$
END;
$\boldsymbol{ans} \longleftarrow \boldsymbol{connectedQ}(\boldsymbol{ct1}, \boldsymbol{ct2}) =$
IF $\boldsymbol{rep}(\boldsymbol{ct1}) = \boldsymbol{rep}(\boldsymbol{ct2})$
THEN $\boldsymbol{ans} := \boldsymbol{connected}$
ELSE $\boldsymbol{ans} := \boldsymbol{notconnected}$
END
END

注意 $\boldsymbol{link}$ 里最关键的一行：

- $\boldsymbol{rep}^{-1}[\{\boldsymbol{rep}(\boldsymbol{ct1})\}]$ 表示所有与 $\boldsymbol{ct1}$ 的代表元相同的城市的集合
- $\boldsymbol{rep}^{-1}[\{\boldsymbol{rep}(\boldsymbol{ct1})\}] \times \{\boldsymbol{rep}(\boldsymbol{ct2})\}$ 是把上述城市都映射到 $\boldsymbol{ct2}$ 的代表元的函数
- 覆盖 $\boldsymbol{rep}$ 使合并后的集合的所有城市都以 $\boldsymbol{rep}(\boldsymbol{ct2})$ 为代表元

城际交通：再精化

上面基于代表元想法的精化已经可以直接映射到计算机实现了（请自己想想如何实现，用什么数据结构，怎样实现操作）

但这一规范还可以改进，提高效率。对于软件的设计合实现，效率问题也是一个关键问题，在形式化开发的精化阶段也应该考虑它

这里的低效在于 $\boldsymbol{link}$ 操作，当需要合并两个集合时，实际上需要检查数组 $\boldsymbol{rep}$ 里的每个元素，考虑是否需要更新代表元

有一种称为 Galler-Fischer 的数据结构，可以有效地表示像这里遇到的这样的数据更新。其基本想法是：

- 不是将集合里的元素映射到一个代表元，而是将每个集合表示为元素的一棵树，以树根作为代表元
- 这样，一个元素的代表元，就是它所在的树的树根

树的结构基于从子结点向上的 $parent \in CITY \rightarrow CITY$ 关系

在这种集合表示里，合并两个集合时，只需将一个集合（是一棵树）的根的 $parent$ 设置为另一集合（也是一棵树）的根

城际交通：树型集合表示

在这种表示里，我们把树根的 $parent$ 设置为它自身

变量 $parent$ 的初始状态就是 id($CITY$)，以此做初始化

下面问题是互通判断。由于不能一步直接找到代表元，我们需要顺着 $parent$ 迭代上行，直到找到一个城市，其 $parent$ 等于自身

下面精化里不想做这种迭代和检查，因此采用了另一种方式。B 里的 $\boldsymbol{R^n}$ 表示某集合 $\boldsymbol{S}$ 上的关系 $\boldsymbol{R} \in \boldsymbol{S} \leftrightarrow \boldsymbol{S}$ 的 $\boldsymbol{n}$ 次复合

用机器读的正文形式，$\boldsymbol{R^n}$ 表示为（在 Atelier B 里需要这样写）

```
iterate(R, n)
```

下面抽象机里维持了一个表示可能最大迭代次数的变量 num，对城市 $\boldsymbol{c}$ 求 $parent^{num}(\boldsymbol{c})$ 必定得到其代表元（$\boldsymbol{c}$ 所在树的根）

关系 $parent^{num}$ 可能通过简单的代码实现

城际交通：树型集合表示

REFINEMENT
$\boldsymbol{CitiesRefRR}(CITY)$
REFINES
$\boldsymbol{CitiesRefR}$
VARIABLES $parent, num$
INVARIANT $parent \in CITY \rightarrow CITY \wedge num \in \mathbb{N} \wedge$
$rep = parent^{num} \wedge$
$\forall \boldsymbol{ct} \cdot (\boldsymbol{ct} \in CITY \Rightarrow parent(rep(\boldsymbol{ct})) = rep(\boldsymbol{ct}))$
INITIALISATION $num := \mathbf{0} \parallel parent := \mathsf{id}(CITY)$

这里的连接不变式是

$$\boldsymbol{rep} = parent^{num} \wedge \forall \boldsymbol{ct} \cdot (\boldsymbol{ct} \in CITY \Rightarrow parent(rep(\boldsymbol{ct})) = rep(\boldsymbol{ct}))$$

也就是说，关系 $parent$ 的 num 次复合一定等于 $\boldsymbol{rep}$，而且 "任一个代表元的 $parent$ 还是这个代表元自己"

这一简单实现是否保证工作得很好？自己想是否可能出问题？（如果是）会出什么问题，能否改进实现。请想想，能保证的最好结果是什么？

城际交通：树型集合表示

这一机器的操作集合：

OPERATIONS
$\boldsymbol{link}(\boldsymbol{ct1}, \boldsymbol{ct2}) =$
VAR $rep1, rep2$
IN
$rep1 := parent^{num}(\boldsymbol{ct1});$
$rep2 := parent^{num}(\boldsymbol{ct2});$
IF $rep1 \neq rep2$
THEN $parent(rep1) := rep2; num := num + \mathbf{1}$ **END**
END;
$\boldsymbol{ans} \longleftarrow \boldsymbol{connectedQ}(\boldsymbol{ct1}, \boldsymbol{ct2}) =$
IF $parent^{num}(\boldsymbol{ct1}) = parent^{num}(\boldsymbol{ct2})$
THEN $\boldsymbol{ans} := \boldsymbol{connected}$
ELSE $\boldsymbol{ans} := \boldsymbol{notconnected}$
END
END

精化的结构总结

现在总结一下精化的各方面问题

1. 一部精化机器总是另一部已有机器（下面称为 “原机器”）的精化，其基本结构与前面介绍的抽象机一样，只是多了有关精化关系的说明子句

2. 精化机器的头部是

   REFINEMENT $name$

   说明这是一个精化，并给出精化机器的名字。人们的习惯是用原来机器的名字加上一个表示精化的后缀

   原机器有参数，精化机器也应有同样参数（不同工具要求可能不同）

   在 Atelier B 里，在同一 project 里建立新 component，选择 Refinement 就是建立精化机器。创建时还应给定原机器的名字。Aterial B 还会要求说明是不是把原机器中一些部分的描述拷贝过来

3. 在 **REFINES** mmm 子句里描述本精化机器是哪个机器的精化。被精化的可以是表示抽象规范的机器，也可以是另一个已做好的精化机器

   原机器的 **SETS** 和 **CONSTANTS** 都可以直接引可用，包括被原机器所 **INCLUDES** 或 **EXTENDS** 的那些机器里的这些部分

   但不包括被原机器 **SEES** 或者 **USES** 的机器里的集合和常量（前面说过，**SEES** 和 **USES** 关系不传递）。如果精化机器需要 **SEES** 某个被它精化的机器所 **SEES** 的机器，它就必须显式地 **SEES** 该机器

4. 精化机器可以利用 B 的结构化机制使用已有的抽象机，可以 **INCLUDES** 或 **EXTENDS** 它们，或者 **SEES** 它们。但不能用 **USES**

   注意：被 **INCLUDES** 的只能是抽象机器 **MACHINE**，不能是精化机器 **REFINEMENT**（对其他结构化操作，都有同样规定），原因是：机器组合构造只关心需要什么功能，不应关心这些功能应该如何实现

   USES 功能只能用在抽象规范的层次，不能用在精化里

5. 精化机器不但可以使用原机器的集合和常量，也可以根据需要定义自己的集合和常量，同样在 **SETS** 或者 **CONSTANTS** 子句里定义

6. 精化机器经常需要定义自己的变量

精化：总结

7. 不变式里的新情况是描述精化机器里的变量与原机器里的变量之间的关系的 “连接不变式”。被原机器 **INCLUDES** 或 **EXTENDS** 的机器里的变量也需要连接，也要在不变式里描述

8. 描述变量的初始化，这样建立的一个初始状态应该通过连接不变式关联于原抽象机的某个初始状态

9. 精化机器里的操作应该与原机器里的操作一样，同样名字，具有相同的输入和输出参数

   这些操作通常不必描述前条件（被精化机器里已经有描述），其操作总是在原机器的前条件下执行

   当某个状态使原机器的某操作的前条件 P 成立时，与之通过连接不变式关联的精化机器的状态也应使这一机器里的相应操作的前条件成立（如果精化机器的操作没写前条件，这一条自然成立）

   在对应状态下用同样输入调用操作，得到的输出应与原机器用同样输入得到的输出相同

精化：总结

下面课程中还要讨论精化的理论，主要讨论精化中的证明义务

在讨论精化的理论之前，还要讨论通过精化减少系统的非确定性的问题