形 式 化 方 法:

基于 B 方法的严格软件开发

(2) 抽象机和软件规范

裘宗燕

北京大学数学学院信息科学系

2010年春季

本部分概要

本部分讨论的内容

1. 了解 Atelier B 系统的基本使用
   - 建立工作空间、项目和组件（包括抽象机）
   - 抽象机的定义
   - 类型检查，证明义务的生成，自动证明
2. 看几个 B 抽象机实例
   - 银行的取号叫号机器
   - 抽象容器机器
   - 座位预定系统（157页）的几个不同版本
   - 客户管理系统

从中了解什么是软件的规范，如何用 B 方法做（描述）软件规范

这里是初步介绍，B 语言和 B 方法细节，以及其理论基础等放在后面

使用 Atelier B 系统 (1)

Atelier B 系统系统启动后显示一个常见形式的图形用户界面

开始工作，创建工作空间（workspace）

1. 在自己计算机的文件系统中选合适位置，准备创建一个 B 方法工作目录
2. 在 Atelier B 系统菜单中 new 项下创建一个 workspace，创建时指定这个 workspace 的目录
3. 这时可以看到这个 workspace

创建项目（project）

1. 通过具体 workspace 的右键菜单里 new 创建项目（也可以点亮相应 workspace 后从 Atelier B 系统菜单创建
2. 创建时给项目命名，一个 workspace 里的项目不重名

通过双击或项目名上的右键菜单 open 项打开项目，右键菜单 close 项关闭项目。可以同时打开多个项目，点亮的是当前项目（唯一）

使用 Atelier B 系统 (2)

创建组件（component, ）

- 通过具体项目（或项目里的 components 项）的右键菜单 new 项创建组件（也可以点亮相应项目后通过系统菜单）创建组件（component）
- 组件可以是 Machine，Refinement，或者 Implementation。现在只考虑 Machine
- 提供组件名，系统将建立相应组件的框架

在组件区点亮的是当前组件

组件的开发

- 在组件窗口区双击要编辑的组件，或通过组件名的右键菜单 edit 项打开专用编辑器
- 编辑器可以高亮关键字，提供运算符（B symbols）选单栏和自动格式功能
- 编辑完成后保存，系统自动做语法检查，在 Outline 区显示概要，指出语法错误

使用 Atelier B 系统 (3)

编辑好的抽象机通过了语法检查，就可以回到 Atelier B 系统窗口，对组件做进一步检查和处理

类型检查：

- 通过 TC 按钮（或组件区里组件名的右键菜单或 Atelier B 系统 component 菜单里的 Type Check 项）启动对组件的类型检查
- 在 Task 区可以看到检查的进展，发现的类型错误，检查结束报告
- 检查完成后，在组件区组件名下有相应标志

证明义务生成：

- 通过 PO 按钮（或上面所说菜单里的 Generate POs 项）启动
- 在 Task 区可以看到生成的进展情况和完成报告
- 检查完成后，在组件区组件名下标明生成的 PO 个数

使用 Atelier B 系统 (4)

自动证明：

- 通过 F0 或 F1 按钮，系统 component 菜单 proof 子菜单或组件的右键菜单的 automatic 项启动自动证明。等证明结束可以看到已通过的证明义务数
- 自动证明分若干证明强度，强度大可能证明更多定理，也需要更多时间
- 系统维护已做证明的情况。 如果再次点击要求证明，系统只考虑未通过的证明义务； 选择更大强度时也是如此

检查证明情况：

- 检查未通过的证明义务的情况，有助于考虑规范里是否有问题
- 通过系统 component 菜单的 proof 子菜单或具体组件的右键菜单里的 Interactive Proof 项启动交互式证明器
- 交互式证明器 Situation 区显示证明义务状态，红色标记未完成的证明义务，按操作分组， 双击打开分组，双击具体证明义务可以看到它的表述

J-R Abrial 认为，能自动确认的证明义务的百分比是规范质量的一个评判标准。 这一看法值得我们参考

几个实例

在深入讨论 B 方法的各方面细节之前，先看几个简单的机器，并做些试验。目的是帮助大家对 B 方法和 B 语言取得一些感性认识

在写这些抽象机时，可能遇到一些 B 语言细节。现在先给出朴素而简单的解释，更细致的语法语义规定和相关理论等在后面介绍

写抽象机的规范，从某种意义看，有些像是写程序，但常常是在一个更抽象的层面上

写抽象机之前也需要分析，但可能是基于一些抽象的数学概念，而不是基于具体的计算机层面的操作

抽象机实例一：银行取号叫号机 (1)

银行取号叫号机器有两个基本功能（非形式描述）

- 到银行要求服务的客户可以从本机器取得一个排队号
- 银行职员空闲时呼叫下一个等待的顾客

这里也可能有许多进一步的细节，例如

- 可能有多个职员，他们都可能启动叫号
- 可能一次叫号没有得到响应，该职员再次按号时，应该叫同一个号
- 多个职员可能“几乎同时”叫号，可能交错叫号和重新叫号
- 等等

下面只考虑一个简单的顺序实现，实际上假定了只有一个职员

叫号机应有的操作：取下一个号，叫下一个号

要能取到“下一个”，必须记录下一个号码，必须用变量记录机器状态

银行取号叫号机 (2)：状态和不变式

机器状态：

- 应该有变量记录将要取出的下一个号：$tnext$
- 应该有变量记录下一个接受服务的号：$snext$

不变式：

- 取号得到的号都应大于 0
- 允许叫的号应该是有人取过的号

自动机：状态和不变式：

MACHINE $enum\_machine$
VARIABLES $tnext, snext$
INVARIANT $tnext \in \mathsf{NAT}_1 \wedge snext \in \mathsf{NAT}_1 \wedge snext \leq tnext$
$\cdots\cdots$
END $/* 怎么初始化？注意\ snext = tnext\ 的意思 */$

银行取号叫号机 (3)：操作

INITIALISATION $tnext, snext := 1, 1$
OPERATIONS

$nn \longleftarrow takeNext =$ **BEGIN**
$nn, tnext := tnext, tnext + 1;$
END;

$nn \longleftarrow get =$ **IF**
$snext < tnext$
THEN
$nn, snext := snext, snext + 1$
ELSE
$nn := 0$
END

$reset =$ **BEGIN** $tnext, snext := 1, 1;$ **END;**

其中：

$nn, tnext := tnext, tnext + 1$

是多重代换（B 语言的术语），表示同时对多个变量做代换

银行取号叫号机 (4)：实际考虑

证明这一机器的不变式定理，还是会出现一个无法解决的证明义务

```
"‘Check that the invariant (tnext: NAT1) is preserved by the
operation - ref 3.4’" => tnext+1<=2147483647
```

修改这一机器，还是需要考虑不断 takeNext 可能溢出的问题

银行取号叫号机 (4)：实际考虑

证明这一机器的不变式定理，还是会出现一个无法解决的证明义务

```
"‘Check that the invariant (tnext: NAT1) is preserved by the
operation - ref 3.4’" => tnext+1<=2147483647
```

修改这一机器，还是需要考虑不断 $takeNext$ 可能溢出的问题

下面的实现考虑到实际情况：一个工作日服务的对象有限

- 一天86400秒，不可能服务超过 80000 人
- 将 $tnext$，$snext$ 定义为取值 1..80000 的整数
- 还需要修改 $takeNext$ 的定义，只在安全的范围内允许取下一个号。用一个 **IF**结构描述这个操作的代换

这样就得到了一个更实际的抽象机规范

注意：操作之间用分号分隔

银行取号叫号机 (5)：实际考虑（计算机形式）

```
MACHINE
    enum_machine
VARIABLES
    tnext, snext
INVARIANT
    tnext : 1..80000 & snext : 1..80000 & snext <= tnext
INITIALISATION
    tnext, snext := 1, 1
OPERATIONS
    nn <-- takeNaxt = IF tnext < 80000
      THEN nn, tnext := tnext, tnext + 1
      ELSE nn := 0 END;
    nn <-- serveNext = IF snext < tnext
      THEN nn, snext := snext, snext + 1
      ELSE nn := 0 END;
    reset = BEGIN tnext, snext := 1, 1 END
END
```


抽象机实例二：容器抽象机 (1)

考虑一类容器，其中可以存放某种东西的一组实例

抽象机可以有参数，可以有表示类型（集合）的参数。容器抽象机：

MACHINE

$$container(ELEM)$$

.....

END

其中 $ELEM$ 表示某个（元素）集合

容器抽象机状态就是容器里当前当前保存的元素集合，它总是 $ELEM$ 的子集，或说是 $ELEM$ 密集的元素。因此它的类型是 $\mathbb{P}\, ELEM$

VARIABLES

$$elems$$

INVARIANT

$$elems : \mathbb{P}\, ELEM$$

INITIALISATION

$$elems := \emptyset$$

容器抽象机 (2)

假定容器是集合，重复存入元素不重复保存

存入元素操作需要有参数，需要用前条件描述对参数的限制

$insert(ex) =$
PRE
$ex : ELEM$
THEN
$elems := elems \cup \{ex\}$
END

窥视容器元素的操作 $peek$

$ex \longleftarrow peek =$
PRE
$elems \neq \emptyset$
THEN
$ex :\in elems$
END

这里看到的是当前容器里的某个元素，非确定选取

容器抽象机 (3)

取出元素既要得到容器里的元素，又要修改容器状态

下面操作 $take$ 用了一个纵向写法的多重代换：

$ex \longleftarrow take =$
PRE
$elems \neq \emptyset$
THEN
$ex :\in elems \;||$
$elems := elems - \{ex\}$
END

座位预定系统 (1)：需求和考虑

假定要为一项活动建立一个座位预定系统

这里简化问题，只考虑座位的数量。也就是说，认为所有座位都一样

（扩充的模型也可以定义，作为练习。考虑状态，不变式，操作）

这样，抽象机的状态就可以用一个变量描述，例如用 $seat$

MACHINE
$booking$
VARIABLES
$seat$
$\ldots\ldots$
END

考虑通用性，抽象机最好有一个自然数参数，表示初始的座位数

这样定义的抽象机可以适合多种需要

座位预定系统 (2)：抽象机参数和约束

抽象机可以有基本数据类型的参数，这时需要用一个 **CONSTRAINTS**子句说明对参数的要求

下面定义里要求参数 max_seat 必须是一个自然数

MACHINE
$booking(max\_seat)$
CONSTRAINTS
$max\_seat \in$ **NAT**
VARIABLES
$seat$
INVARIANT
$seat \in 0..max\_seat$
INITIALISATION
$seat := max\_seat$
OPERATIONS
$\ldots\ldots$
END

下面考虑这个系统的操作

座位预定系统 (3)：操作一

考虑定义如下几个操作：

- 预定一个座位和取消一个座位预定的操作
- 预定 n 个座位和取消 n 个座位预定的操作
- 查询剩余座位数的操作

前两个操作很简单，定义如下：

$book =$
PRE $0 < seat$
THEN
    $seat := seat - 1$
END;

$cancel =$
PRE $seat < max\_seat$
THEN
    $seat := seat + 1$
END

座位预定系统 (4)：操作二

我们可能允许一次预定若干座位，也允许一次退回若干座位， 当然不应该出现退回后总座位数超过初始总座位数的情况

两个操作的定义如下

$bookn(sn) =$
PRE
    $sn \in \mathbf{NAT1} \wedge sn \leq seat$
THEN
    $seat := seat - sn$
END;

$canceln(sn) =$
PRE
    $sn \in \mathbf{NAT1} \wedge seat + sn \leq max\_seat$
THEN
    $seat := seat + sn$
END

还可以考虑，例如防止倒票黄牛，限制一次购票的最大张数等

座位预定系统 (5)

最后一个操作是查询剩余座位数，非常简单：

$num \longleftarrow nseat =$
BEGIN
    $value := seat$
END

显然这里有许多不尽人意的地方，请大家考虑，例如：

- 用数的集合取代整数
- 允许选座位
- 设法保证退还的座位是前面预定的（这个比较困难）
- 等等

座位预定系统 (6)

如果采用座位编号集合，可能采用下面的状态定义：

VARIABLES
    $seats$
INVARIANT
    $seats \in \mathbb{P}\,\mathbf{NAT} \wedge seats \subseteq \{sn \mid sn \in \mathbf{1}..max\_seat\}$
INITIALISATION
    $seats := \{sn \mid sn \in \mathbf{1}..max\_seat\}$
OPERATIONS
    .....
    $bo \longleftarrow have\_seat =$
    IF $seats \neq \emptyset$
    THEN $bo :=$ **TRUE**
    ELSE $bo :=$ **FALSE**
    END
END

请大家在上面框架的基础上把这个抽象机定义好，并试验一下， 看看是否能确认定义好的抽象机所有证明义务（下次课交流）

规范和模型

我们已经看到了几个规范，它们描述了功能差异很大的不同软件系统或部件

- ***counter*** 是一个简单的软件模块的规范
- ***enum_machine*** 是一个简单的计算机化应用系统的模型
- ***container*** 是一类软件模块的抽象，一般的栈/队列等可能看作它的精化
- ***booking*** 和后面的扩充版本都可以看作一个实际的复杂计算机系统的简化版本， 我们有可能进一步扩充其功能，去开发更实际的系统

这些例子也显示了我们前面讲的一些问题：

写规范就是构造模型，建立模型。

此时最关键的是构造出一个清晰易理解的模型， 而不是考虑软件的细节实现问题（例如，具体数据表示方式，算法和效率等）

下面讨论一些写规范时的一般性问题

规范的宽松风格和防御风格（164页）

写规范有两种基本风格：宽松式风格和防御式风格

注意：抽象机提供了一组操作，供抽象机的外部调用。 现在说的宽松或防御，都是指抽象机与其外部的关系，对其外部的基本看法

宽松式风格： 宽松风格的规范对使用方提出一些正确使用方面的要求（通过操作的前条件）， 采用这种描述，实际上是“相信”外部调用这些操作时会负责任，会遵循自己提出的前条件

防御式风格： 写规范时并不相信外部能提供完全的保证， 规范里的动作都基于自己操作中所做的检查和对各方面情况的了解

对于同一个抽象机，可以按照宽松风格描述，也可以按防御式风格描述

虽然采取不同风格都能写出“正确”的规范， 但不同规范在设计的复杂程度，需要考虑问题的多少， 将来实现的效率，以及（最重要的）完成后系统的坚固性（抵御外来错误的能力）等方面都会有差异

规范的宽松风格

以前面定义的 $booking$ 抽象机为例，那里采用的是宽松风格：

- 一些操作有（类型要求之外的）前条件，只有在前条件满足的情况下执行操作，才能保证不变式的维持
- 这实际上要求用户调用代码必须关注前条件，保证用满足前条件的实参调用这些操作。 组件的正确性实际上依赖于用户的所有使用的正确性
- 只有以某种方式“证明”了所有用户调用都正确，才能保证这一组件的行为确实正确。 增加一个调用，实际上需要增加一个“证明”
- 注意：编程语言可保证调用时类型正确，但不能保证其他前条件得到满足

例如

- $canceln$ 操作有前条件 $seat + sn \leq max\_seat$
- $bookn$ 操作有前条件 $sn \leq seat$

系统的证明义务只要求在这些前条件成立的情况下维持不变式， 对调用时参数不满足前条件的情况没提出任何要求

规范的防御风格

防御式风格的基本考虑是不相信操作的调用方能永远正确使用本模块的操作

实际策略：

- 总是写不依赖于机器的具体状态和参数情况的操作（规范）
- 通过操作内部的检查区分不同情况，只在满足条件时才做实际操作
- 其他情况下适当地合理处理（具体情况具体分析）

具体技术是把前条件中的非类型要求转到一个条件语句里。如

$bookn(sn) =$
PRE
  $sn \in \mathsf{NAT}_1$
THEN
  IF $sn \leq seat$
  THEN $\cdots$ /* 正确情况下的处理 */
  ELSE $\cdots$ /* 错误情况下的处理 */
  END
END /* 这个操作定义只要求参数类型正确 */

规范的宽松风格和防御风格

貌似这种转换有些机械，但很难合理地自动将前条件转换为控制流中的条件判断。 因为错之后的合理处理都是具体的，一般而言不能自动生成

宽松风格和防御风格不是写规范中特殊的东西。 《程序设计实践》的作者是搞实际的软件工程师，他们也谈到防御式风格，举了一个例子：

> 美海军约克敦号导弹巡洋舰在海上航行中， 有位船员不适当地从键盘输入了一个 0。由于没有良好的防御性设计， 这个错误数据冲入系统，最后关闭的船的推进系统，导致约克敦号在海上飘了几个小时

防御性处理也需适当合理

> 阿里亚娜五型火箭首飞爆炸是不适当的防御性设计的例子。 由于加速度大导致一个浮点变量溢出，系统自动关闭了导航系统，最后火箭自毁

防御性设计是保证系统的安全性的重要技术， 其基本想法是让系统里的各个部分能保护自己不受外来错误的影响

基本落脚点：每个模块都更加强健了，整个系统也会更加强健

防御风格和系统安全性

从整个系统的角度看，防御性设计，就是要保证系统对任何外来信息都能合理处理， 目前的高可信系统都特别强调这个方面

- 过去设计系统，较多考虑的是在合法输入下的功能正确
- 目前增加的要求：对不合法数据也能恰当处理，而且保证系统遇到不合法数据时还能继续工作

这些时软件设计观点的变化

一个系统的安全性设计需要有一个全局规划：在哪些模块的哪些接口函数采用防御式设计， 以保证运行中出现的错误不会流窜到系统的关键部分

采用防御风格，还需要设计好发现错误时的处理

- 退出正在执行的操作？
- 向用户报告问题？
- 状态的修改或维护？

机器booking的另一种设计

修改 $booking$ 的设计，需要在参数出错时报告。下面设计用 **BOOL**值报告，**TRUE**表示操作成功，**FALSE**表示操作出错

$report \longleftarrow bookn(sn) =$
PRE sn **:** NAT_1
THEN
    IF $sn \leq seat$
    THEN $report, seat :=$ **TRUE**, $seat - sn$
    ELSE $report :=$ **FALSE**
    END
END;
$report \longleftarrow canceln(sn) =$
BEGIN
    IF $seat + sn \leq max\_seat$
    THEN $report, seat :=$ **TRUE**, $seat + sn$
    ELSE $report :=$ **FALSE**
    END
END

客户管理系统 (1)：问题

假设我们要开发一个超级市场销售管理系统。下面只考虑客户管理

根据情况将客户分为三类：

- 正常客户，优惠为0，默认信用卡消费限额为1000元
- VIP 客户，优惠10%，有较高的信用卡消费限额
- 可疑客户，优惠为0，有较低的信用卡消费限额

系统提供的功能可能包括：

- 加入客户，开始加入的客户都是正常客户
- 转换客户类别
- 为客户设定新的信用卡消费限额
- 等等

下面考虑这样一个系统的设计

客户管理系统 (2)：基本集合

要建立客户管理系统的状态，需要几个基本集合：

- 客户集合
- 客户类别的集合

B 抽象机里可以定义集合。集合分为两种：

- 待定集合（延期集合），用一个标识符表示
- 枚举集合，用一个标识符，后跟一个枚举项列表

这一抽象机开头是：

MACHINE
$customer$
SETS
$Client$;
$Category = \{vip, normal, dubious\}$
$\cdots\cdots$

客户管理系统 (3)：状态

分类优惠是一套固定的计算方法，是系统“常量”

B 抽象机里可以定义常量，并通过 **PROPERTIES**子句给定值

这里定义常量 $discount$，它是一个从分类到 0..100 的有限函数

CONSTANTS
$discount$
PROPERTIES
$discount : Category \rightarrow 0..100 \wedge$
$discount = \{vip \mapsto 90, normal \mapsto 100, dubious \mapsto 100\}$

关于复杂类型的描述等问题，下面将详细讨论

客户管理系统 (4)：状态

现在考虑变量。需要有变量表示

- 目前的客户集合，是 $Client$ 的子集
- 每个客户的分类，是从客户到 $Category$ 的函数
- 每个客户允许的信用卡限额，是从客户到某段自然数的函数

假设我们确定任何客户的信用卡购物不能超过 2000 元，可以如下定义变量、不变式和变量初始化：

VARIABLES

$client, category, allowance$

INVARIANT

$client : \mathbb{P}(Client) \wedge$
$category : client \rightarrow Category \wedge$
$allowance : client \rightarrow 1..2000 \wedge$

INITIALISATION

$client, category, allowance := \emptyset, \emptyset, \emptyset$

客户管理系统 (5)：操作示例

现在考虑抽象机的操作。这里只列出两个操作作为示例

首先是增加客户的操作

只有还能增加客户的情况下操作才能进行，（为其分配一个客户记录）

加入客户需要为其指定类别和购物限额：

OPERATIONS

$clt \longleftarrow addClient =$
IF
$client \neq Client$
THEN
$clt :: Client - client \;||$
$client := client \cup \{clt\} \;||$
$category := category \cup \{clt \mapsto normal\} \;||$
$allowance := allowance \cup \{clt \mapsto 1000\}$
END;

客户管理系统 (6)：操作示例

考虑修改客户限额的操作

- 只有非 $normal$ 客户才能（才需要）修改限额
- vip 客户只能赋予更大的限额；$dubious$ 客户只能赋予更小的限额

$$
\begin{array}{l}
modifyAllowance(clt, allow) = \\
\textbf{PRE} \\
\quad clt : Client \land allow : 1..2000 \\
\textbf{THEN} \\
\quad \textbf{IF} \\
\quad\quad category(clt) \neq normal \land \\
\quad\quad category(clt) = vip \Rightarrow allow \geq 1000 \land \\
\quad\quad category(clt) = dubious \Rightarrow allow \leq 1000 \\
\quad \textbf{THEN} \\
\quad\quad allowance := allowance \lhd\!\!+ \{clt \mapsto allow\} \\
\quad \textbf{END} \\
\textbf{END} \\
\textbf{END}
\end{array}
$$

总结

本节主要讨论了几个问题：

- 介绍了 Aterial B 系统和如何在其中开发验证规范
- 通过几个例子，介绍了 B 语言和抽象机的一些概念
- 通过例子说明了写 B 规范需要考虑的问题，基本问题是：
  - 抽象机状态的设计（常量/变量/不变式/初始化等）
  - 抽象机操作的设计
- 抽象机设计依赖于对要描述问题的分析，但也有不同的考虑：
  - 宽松风格的规范
  - 防御风格的规范

下面将结合软件规范描述，介绍 B 语言的一些基本概念和理论