形 式 化 方 法:

基于 B 方法的严格软件开发

(7)

抽象机组织—INCLUDES

裘宗燕

北京大学数学学院信息科学系

2010年春季

概要

需要开发的软件系统可能非常复杂，为了描述它们而写出的抽象机的规范也可能任意复杂，包含复杂的状态和不变式，大量操作，规范非常长

这时我们面临常规复杂软件开发同样的问题：需要控制描述的复杂性

过于冗长而复杂的规范（和冗长复杂的程序一样）将不容易写正确，不容易理解，也不容易修改。同样难以证明其正确性

解决复杂问题的唯一途径就是设法分解其复杂性。B 方法提供了一些机制，使规范的开发者可以把一部复杂抽象机描述为一些较简单抽象机的组合

利用这些机制，可以把一部 B 抽象机的状态分解到一些相互独立的（子）机器里，让每个机器在整个状态的一部分上操作

要使这些机器最终能组合为一个整体的抽象机，B 方法还提供了一些方式来描述作为一部机器的组成部分的不同机器之间的关系

抽象机：分解和构造

允许将一部抽象机描述为一些独立部分的组合，带来许多益处：

- 一个规范中概念上独立的不同部分可以分别描述和理解，这些描述和理解可在集成到整体规范描述之前完成
- 独立的成员抽象机的内在一致性可以独立地证明
- 有利于规范的重用，已开发好并经过验证的抽象机有可能用于其他规范
- 如果规范的分解做得好（模块化做得好），还可能减少需要验证的证明义务，减少规范开发的工作量

规范的分解方式（规范的组织结构）不一定要反映抽象机最终实现的方式（可能在很多情况下，规范结构确实反映了实现的结构，但不必如此）

在考虑规范的结构时，最重要的考虑应该是：

- 通过分解和构造，使与一个复杂软件相关的大量信息更容易理解
- 设法分解一个复杂规范里的不同方面的考虑

抽象机：分解和构造

B 方法规范的组合构造机制实现的是一种 “半隐藏” 机制，允许在使用成员抽象机的规范时引用其中的一些内部细节

如果使用了抽象机的内部细节，所定义的抽象机的正确性就会依赖于被使用机器的内部信息，以及那些机器的操作的正确性

B 方法提供了一些抽象机的组织构造机制。本节讨论 **INCLUDES** 机制，其作用就是把已有的抽象机规范作为整体包含到更大的抽象机里

INCLUDES 的基本用途是构造在被定义的抽象机内部使用的 “私用” 机器，被 **INCLUDES** 的抽象机完全隐藏在包含它的抽象机内部，外界完全看不到

为了实际需要，B 方法提供了一种 **PROMOTES** 子句，用它可以把被包含抽象机的一部分操作 “提升” 为外围抽象机的接口操作。这样就可以把被包含抽象机的一部分接口暴露出来

还有一种特殊形式的 **INCLUDES** 称为 **EXTENDS**，它将被 **EXTENDS** 的抽象机的整个接口暴露给外部，可以看作是一种功能扩充，类似于 OO 的继承

包含－INCLUDES（7.2.3节）

一般说，一部抽象机描述了一个复杂系统的状态中的一个自足部分，描述了

- 这个部分的行为（静态行为和动态行为）
- 这一部分与其环境的交互接口（操作集合）
- 在各种交互中该系统部分的动态演化进展

抽象机的操作说明了它的环境能访问和改变其状态的所有可能方式

抽象机的证明义务保证：无论其环境如何使用它的操作（通过满足其前条件要求的使用），导致的机器状态变化都不会破坏机器的不变式

抽象机包含操作牵涉到两部抽象机。假设现在已有一部抽象机 M_1，通过在另一抽象机 M_2 的规范里写子句

INCLUDES M_1

就可以将抽象机 M_1 包含进来作为 M_2 的一个部分

这时说 "M_2 包含了 M_1"

包含－INCLUDES

如果 M_2 包含了 M_1，那么我们认为

- M_1 实际上是 M_2 的描述的一部分
- M_1 的状态是 M_2 的实际状态的一部分
- M_1 的描述提供的所有信息看作是 "被包含" 在 M_2 的信息里，与 M_2 本身的各子句中描述的 "本地" 信息有所不同

一般而言，**INCLUDES** 用于包含抽象机的部件，通过这种包含，从简单的抽象机组合出更为复杂的抽象机

一部抽象机可以 **INCLUDES** 任意多部抽象机；被 **INCLUDES** 的抽象机里完全可以已经包含了其他的抽象机（支持任意深度的分层构造）

下面先考虑抽象机 M_2 里只包含一部抽象机 M_1 的情况。其他更一般的情况都与此类似，是这种简单情况的直接推论

在下面的讨论中，我们假定抽象机 M_2 包含了抽象机 M_1。给出包含中的一些细节情况和规定

抽象机包含 (1)：参数，集合和常量

1) 如果 $M_1(x)$ 是带参数的抽象机

- M_2 包含它时必须实例化这些参数。被包含的总是 $M_1(x)$ 的具体实例
- 这种实例化产生一个证明义务：要求证明 M_2 包含 M_1 时的实际参数符合 M_1 的规范对参数的要求（由 M_1 的 **CONSTRAINTS** 子句描述）

2) M_1 里的集合和常量在 M_2 可见，就像是在 M_2 里定义

- 它们是被 M_2 包含进来的集合和常量，M_2 可以像引用其本地集合和常量一样引用被包含抽象机的集合和常量
- M_2 的 **PROPERTIES** 里的谓词可以涉及被包含的集合和常量，特别是可以描述被包含集合和常量与本地描述的集合和常量之间的关系
- 注意：**PROPERTIES** 的基本作用是描述本地集合和常量的类型和性质。上面说法意味着，如果需要，可以建立本地集合和常量与被包含的抽象机的集合和常量之间的关联

抽象机包含 (2)：状态

3) M_1 的状态成为 M_2 的状态的一部分

- M_2 的不变式可以描述对 M_1 的状态变量的要求，也可以描述 M_2 的本地变量与 M_1 的变量之间的关系
- M_1 的状态在 M_2 的规范描述直接可见，因此在 M_2 操作的前条件 **PRE** 里和操作体里都可以以读的方式访问这些变量，但不能修改
- M_2 的一致性要求其操作维持不变式，包括使用了 M_1 的操作的 M_2 操作
- M_2 的不变式包含 M_1 的不变式，M_1 的不变式能为 M_2 的证明义务提供信息。由于 M_2 操作不直接修改 M_1 状态，因此无须重证明 M_1 的不变式

4) 由于要保证被包含机器 M_1 的不变式，因此

- 在 M_2 的操作和初始化里都不能给 M_1 的变量赋值，因为这种赋值可能打破 M_1 的不变式，破坏证明的模块化
- 要保证 M_1 的自足性质、模块化和通过增量方式构造大型规范，在 M_1 的外部只能通过调用它的操作的方式改变其状态

抽象机包含 (3)：状态和初始化

5) M_2 的本地不变式可能关注 M_1 的状态，该状态是 M_2 状态的一部分，要保证自己的本地不变式成立，M_2 必须完全控制自己的状态变化。这意味着

- M_1 的操作应该只能在 M_2 里可用，这一 M_1 实例不能被同一软件开发项目里除 M_2 之外的其他机器包含
- M_2 可以通过调用 M_1 的操作，以受控方式更新 M_1 的状态
- 在做这种调用时，必须要保证被调用操作的前条件（由其 **PRE** 描述）成立。这种保证是 M_2 的证明义务中的一部分

6) 抽象机 M_2 的初始化过程是：

- 首先做被 M_2 包含的所有机器的 **INITIALISATION** 描述的初始化工作，然后在做 M_2 自己的本地初始化子句
- 这种情况说明，在 M_2 的本地初始化子句里可以引用 M_1 的状态，可以根据 M_1 的实际初始化情况做 M_2 的本地初始化
- 如前所述，M_2 的初始化子句里不能修改 M_1 的状态

包含与被包含抽象机的关系（附录 D，可见性）

总结：假设抽象机 M_2 包含了抽象机 M_1，那么

- 在 M_2 的 **PROPERTIES** 子句里，可以以读的方式访问
  - 本地的集合和常量；M_1 里的集合和常量
- 在 M_2 的 **INVARIANT** 子句里，可以以读的方式访问
  - 本地的集合、常量和变量；M_1 里的集合、常量和变量
- 在 M_2 的 **INITIALISATION** 子句里，可以
  - 以读的方式引用 M_1 里的集合、常量和变量
  - 读本地的集合和常量，设置本地的变量
- 在 M_2 的 **OPERATIONS** 子句的操作里，可以
  - 以读的方式引用 M_1 里的集合、常量和变量
  - 读本地的集合和常量，读和更新本地的变量

M_1 应已有定义，显然它不可能访问 M_2 的状态

PROMOTES

可以把被包含抽象机 M_1 的操作提升为抽象机 M_2 的操作。假设要提升 M_1 的操作 *operation1*，写（不需要指明 M_1）

PROMOTES *operation1*

提升后 *operation1* 就成为 M_2 的接口的一部分

显然 *operation1* 这样的操作只会改变 M_2 里 M_1 的那部分状态。但是

- 由于现在 *operation1* 也要作为 M_2 的接口的一部分，因此就必须证明它也能维持 M_2 的不变式。证明了这一点之后，在 M_2 之外调用 *operation1* 就是安全的了
- 这种证明需求带来了新的证明义务
- 提升后的操作既可以在 M_2 外面使用，也可以在 M_2 的操作里使用

在 **PROMOTES** 后面可以列出一系列操作名，逗号分隔，表示将被包含机器里的这些操作提升为当前机器的操作

EXTENDS

如 M_2 提升了被包含机器 M_1 的所有操作，M_2 就是 M_1 的一个扩充：

- M_2 包含 M_1 的状态变量，还可能有所扩充（可能扩充状态空间的维）
- M_2 提供了 M_1 的所有操作，还可能有所增加（可能扩充接口）

扩充是包含的一种特殊情况，通过 **INCLUDES** 子句和 **PROMOTES** 描述比较麻烦。B 方法为此提供了一种专门描述方式。在 M_2 里写

EXTENDS M_1

说明 M_2 是 M_1 的扩充，定义的 M_2 默认提供了 M_1 的所有操作

一个抽象机可以是若干已有抽象机的扩充，它包含了所有被它 **EXTENDS** 的抽象机的功能，提供了它们的所有操作。被扩充的抽象机名字列在 **EXTENDS** 之后，多个名字之间用逗号分隔

一个抽象机可以扩充了几个抽象机，同时又包含了几个抽象机。下面讨论里提到的被包含抽象机，也包括了被扩充抽象机

多重包含和重命名

有时可能需要包含同一抽象机的几个实例，直接包含将出现名字冲突，这时就要重命名（重命名还可用于排除不同抽象机里的名字之间的相互冲突）

重命名的方式是在被包含抽象机名前加前缀名和一个圆点。这样重命名也重新命名了被包含抽象机里的所有变量和操作（集合和常量不重命名）

例：假定已有表示车轮的抽象机 *wheel*，它有内部变量 $\boldsymbol{x}$ 和 $\boldsymbol{y}$，两个操作 *pressure* 和 *temperature* 检查轮胎的压力和温度

现要定义 *car* 抽象机，它要包含四个车轮。可以写包含子句

INCLUDES $LF.whell, RF.wheel, LR.wheel, RR.whell, \ldots$

在抽象机 *car* 里包含了 *wheel* 抽象机的四个实例，此时

- $LF.x$ 和 $LF.y$ 表示 $LF.whell$ 里的那两个变量
- $LF.pressure$ 和 $LF.temperature$ 检查车轮 $LF.whell$ 的压力和温度

抽象机的重命名具有传递性。假设 $\boldsymbol{M_2}$ 包含 $\boldsymbol{M_1}$ 时将其重命名为 $\boldsymbol{a.M_1}$，而 $\boldsymbol{M_3}$ 在包含 $\boldsymbol{M_2}$ 时将其重命名为 $\boldsymbol{b.M_2}$。如果要在 $\boldsymbol{M_3}$ 里（的初始化或操作子句里）引用 $\boldsymbol{M_1}$ 里的变量 $\boldsymbol{x}$，就要写 $\boldsymbol{b.a.x}$

调用抽象机的操作

要调用被包含抽象机的操作，只需给出被调用操作的名字，并根据该操作的情况给出实际输入参数和接受输出的变量

根据被调用操作的定义，操作调用可能有下面几种形式：

$\boldsymbol{op\_name}$	调用无输入无输出参数的操作
$\boldsymbol{op\_name(Exp\_List)}$	调用有输入无输出参数的操作
$\boldsymbol{Var\_List \longleftarrow op\_Name}$	调用无输入有输出参数的操作
$\boldsymbol{Var\_List \longleftarrow op\_Name(Exp\_List)}$	调用有输入有输出参数的操作

操作调用是一种代换（前面讲代换时说还剩下一种情况，就是这个）。假设已有操作的定义是（注意，操作体是一个代换 $\boldsymbol{S}$）

$$op \;=\; \boldsymbol{S}$$

如果希望调用这一操作后谓词 $\boldsymbol{Q}$ 成立，实际上就是要求做了相应的代换 $\boldsymbol{S}$ 后谓词 $\boldsymbol{Q}$ 成立，即

$$[op]\boldsymbol{Q} \;\widehat{=}\; [\boldsymbol{S}]\boldsymbol{Q}$$

这个写法不精确，应该加上输入和输出参数的代换。下面说明

调用抽象机的操作

假设有操作

$report \longleftarrow update(value) =$

PRE $value \in \mathbb{N}$

THEN

$var, report := value, good$

END

做下面调用

$res \longleftarrow update(var1 + \mathbf{13})$

相当于做下面代换

PRE $var1 + \mathbf{13} \in \mathbb{N}$

THEN

$var, res := var1 + \mathbf{13}, good$

END

这个代换是对原来的操作体做下面代换的结果

$report, value := res, var1 + \mathbf{13}$

对代换做代换

前面讨论里用到对代换做代换

$$[S_1]S$$

我们希望得到一个新的代换，作为实际操作的规范

由于希望得到的结果仍然是代换，这就要求 S 经过代换后其 := 的左边仍是一个或几个变量（也就是说只是变量换名，否则代换结果就不是代换了），而代换后 := 的右边可以是任意合法的表达形式

上例中被代换的代换是：

PRE $value \in \mathbb{N}$

THEN

$var, report := value, good$

END

要对它做代换，并保证代换结果仍是合法的代换，对于 $var, report$ 只能做换名，而对其中出现的其他名字，即 $value, good$，则没有任何限制

前面的代换 "$report, value := res, var1 + \mathbf{13}$" 满足这些要求

实例：银行保险箱

考虑银行的一组保险箱，给出一个规范，模拟保险箱的开闭状态变化

下面考虑分层开发方法，首先模拟一组柜子，模拟它们的门的情况，每个门可以是打开的或者关闭的：

MACHINE $Cases$
SETS $DOOR; POSITION = \{open, closed\}$
VARIABLES $position$
INVARIANT
$position \in DOOR \rightarrow POSITION$
INITIALISATION
$position := DOOR \times \{closed\}$
OPERATIONS
$opening(dd) =$
PRE $dd \in DOOR$ **THEN** $position(dd) := open$ **END;**
$closedoor(dd) =$
PRE $dd \in DOOR$ **THEN** $position(dd) := closed$ **END**
END

实例：银行保险箱

现在基于前面 $Cases$ 定义新抽象机，实现一组带锁的柜子 $LockCases$，它以 $Cases$ 作为部件，增加了柜子是否锁定的状态：

MACHINE $LockCases$
INCLUDES $Cases$
PROMOTES $closedoor$
SETS
$STATUS = \{locked, unlocked\}$
VARIABLES
$status$
INVARIANT
$status \in DOOR \rightarrow STATUS \wedge$
$position^{-1}[\{open\}] \subseteq status^{-1}[\{unlocked\}]$
INITIALISATION
$status := DOOR \times \{locked\}$

把 $Cases$ 的 $closedoor$ 提升为 $LockCases$ 的接口操作

注意不变式：“打开的门一定是没有锁定的门”

实例：银行保险箱

$LockCases$ 的操作如下：

OPERATIONS
  $opendoor(dd) =$
    PRE $dd \in DOOR \wedge status(dd) = unlocked$
    THEN $opening(dd)$
    END;
  $unlockdoor(dd) =$
    PRE $dd \in DOOR$
    THEN $status(dd) := unlocked$
    END;
  $lockdoor(dd) =$
    PRE $dd \in DOOR \wedge position(dd) = closed$
    THEN $status(dd) := locked$
    END
END

由于提升了 $Cases$ 的操作 $closedoor$，这一抽象机共有 4 个接口操作

实例：银行保险箱

考虑一组钥匙，其中有些钥匙在用（当时被插入了锁的钥匙孔），其他钥匙没在用。抽象机定义很简单：

MACHINE $Keys$
SETS KEY
VARIABLES $keys$ /* 在用钥匙集 */
INVARIANT $keys \subseteq KEY$
INITIALISATION $keys := \{\}$
OPERATIONS
  $insertkey(kk) =$
    PRE $kk \in KEY$ **THEN** $keys := keys \cup \{kk\}$ **END;**
  $removekey(kk) =$
    PRE $kk \in KEY$ **THEN** $keys := keys - \{kk\}$ **END**
END

在上面这些抽象机的基础上，可以实现银行保险柜抽象机 $Safes$，其中定义钥匙与门（锁）的关系，和一些相关操作

部件抽象机的一些操作被提升为保险柜抽象机的操作

实例：银行保险箱

MACHINE $Safes$
INCLUDES $LockCases, Keys$
PROMOTES $opendoor, closedoor$
CONSTANTS $unlocks$
PROPERTIES $unlocks \in KEY \rightarrowtail DOOR$ /* 每个柜各有一把钥匙 */
INVARIANT $status^{-1}[\{unlocked\}] \subseteq unlocks[keys]$ /* 有钥匙才能开锁 */
OPERATIONS
$insert(kk, dd) =$ /* 插入钥匙 */
PRE $kk \in KEY \wedge dd \in DOOR \wedge unlocks(kk) = dd \wedge$
$kk \notin keys$
THEN $insertkey(kk)$
END;
$extract(kk, dd) =$ /* 拔出钥匙 */
PRE $kk \in KEY \wedge dd \in DOOR \wedge unlocks(kk) = dd \wedge$
$kk \in keys \wedge status(dd) = locked$
THEN $removekey(kk)$
END;

实例：银行保险箱

$unlock(dd) =$
PRE $dd \in DOOR \wedge unlocks^{-1}(dd) \in keys$
THEN $unlockdoor(dd)$
END;
$lock(dd) =$
PRE $dd \in DOOR \wedge unlocks^{-1}(dd) \in keys \wedge$
$position(dd) = closed$
THEN $lockdoor(dd)$
END;
$quicklock(dd) =$
PRE $dd \in DOOR \wedge unlocks^{-1}(dd) \in keys \wedge$
$position(dd) = closed$
THEN $lockdoor(dd) \parallel removekey(unlocks^{-1}(dd))$
END
END

总结

- 大型软件的规范也需要良好的组织
- B 提供了一些用于组织规范的机制，支持基于较小较简单的抽象机，定义更大更复杂的抽象机，支持软件规范的分层组织和定义
- **INCLUDES** 将被包含抽象机看作所定义的抽象机的组成部分
  - 被包含抽象机的状态看作是外围抽象机的状态的一部分
  - 被包含抽象机里的信息可以按一定规则在外围抽象机里使用
  - 包含带来一些证明义务，要证明操作调用满足被调用操作的前条件等
- 可以通过 **PROMOTES** 子句把被包含自动机的操作提升起来，但这样做也会带来新的证明义务
- 可以用 **EXTENDS** 方式基于一个或一组抽象机扩充，定义一个新的包含所有被扩充抽象机的功能的抽象机。被 **EXTENDS** 的抽象机的功能都是新定义抽象机的功能，这样做也会产生一批证明义务