形 式 化 方 法:

基于 B 方法的严格软件开发

(8)

抽象机组织－(2)

裘宗燕

北京大学数学学院信息科学系

2010年春季

概要

本节进一步讨论系统规范的结构化构造问题，包括：

- 并行代换的进一步解释。由于抽象机包含等的存在，实际上产生了更复杂的并行代换情况，本节将详细讨论这方面情况
- 抽象机的证明义务。抽象机包含提出了新的证明义务。这里将详细介绍抽象机的证明义务，包括包含带来的证明义务
- 如何在 B 规范中处理软件系统与物理环境的联系，包括时间问题。下面将讨论一些想法和可能的处理方法
- 其他抽象机连接和组合机制，包括 **SEES** 连接和 **USES** 连接
- **SEES** 和 **USES** 抽象机组合机制产生的证明义务方面的情况
- 使用 **SEES** 和 **USES** 的系统规范实例
- 特别是 B Book 上的货单系统实例，是一个有一定规模的系统，还可以进一步扩充

并行代换（同时代换）

当一个抽象机包含了一个或多个子抽象机时，它的一个操作就可能并行地同时调用多个子操作，同时更新多个机器的状态。而被包含机器还可能包含子抽象机，其操作又可能调用子抽象机的操作，形成复杂的并行代换

（这里有一点需要注意：不能同时调用同一个抽象机的两个操作，因为这种调用可能要求对同一变量同时做多次代换，是错误的）

假设有调用 $\mathsf{op}_1 \parallel \mathsf{op}_2$，两个操作分别有前条件和体，那么：

$$\textbf{PRE } P_1 \textbf{ THEN } S_1 \textbf{ END} \parallel \textbf{PRE } P_2 \textbf{ THEN } S_2 \textbf{ END}$$
$$= \textbf{ PRE } P_1 \wedge P_2 \textbf{ THEN } S_1 \parallel S_2 \textbf{ END}$$

出现在结果中的 $S_1 \parallel S_2$ 还可能进一步归约

如果操作的体非常复杂，这样的并行归约还可以进一步归约。为了生成证明义务，需要最终消除代换里的 || 组合符

下面看一些常用的归约规则

并行代换（同时代换）

$$S \parallel \mathsf{skip} = S$$

$$S_1 \parallel S_2 = S_2 \parallel S_1$$

$$(x_1, \ldots, x_n := E_1, \ldots, E_n) \parallel (y_1, \ldots, y_m := F_1, \ldots, F_m)$$
$$= x_1, \ldots, x_n, y_1, \ldots, y_m := E_1, \ldots, E_n, F_1, \ldots, F_m$$

$$(\textbf{IF } P \textbf{ THEN } S_1 \textbf{ ELSE } S_2 \textbf{ END}) \parallel T$$
$$= \textbf{ IF } P \textbf{ THEN } S_1 \parallel T \textbf{ ELSE } S_2 \parallel T \textbf{ END}$$

$$(\textbf{CHOICE } S_1 \textbf{ OR } S_2 \textbf{ END}) \parallel T$$
$$= \textbf{ CHOICE } S_1 \parallel T \textbf{ OR } S_2 \parallel T \textbf{ END})$$

$$(\textbf{PRE } P_1 \textbf{ THEN } S \textbf{ END}) \parallel T$$
$$= \textbf{ PRE } P_1 \textbf{ THEN } S \parallel T \textbf{ END}$$

$$(\textbf{ANY } x \textbf{ WHERE } E \textbf{ THEN } S \textbf{ END}) \parallel T$$
$$= \textbf{ ANY } x \textbf{ WHERE } E \textbf{ THEN } S \parallel T \textbf{ END}$$

要求 T 不是前条件代换
如果是可以先用上一条规则

并行代换（同时代换）

可以用这些等价定律把并行组合符向 **CHOICE**，**IF** 的内部移动，直至并行组合的是两个简单代换，这时就可以用前两条规则删除 ||。看下面例子：

IF $x > 0$ **THEN** $y := y + 1$ **ELSE** $y := y - 1$ **END**
|| **ANY** u **WHERE** $u \in 1..10$ **THEN** $z := z + u$ **END**

$=$ **IF** $x > 0$
THEN $y := y + 1$ || **ANY** u **WHERE** $u \in 1..10$ **THEN** $z := z + u$ **END**
ELSE $y := y - 1$ || **ANY** u **WHERE** $u \in 1..10$ **THEN** $z := z + u$ **END**
END

$=$ **IF** $x > 0$
THEN ANY u **WHERE** $u \in 1..10$ **THEN** $z := z + u$ || $y := y + 1$ **END**
ELSE ANY u **WHERE** $u \in 1..10$ **THEN** $z := z + u$ || $y := y - 1$ **END**
END

$=$ **IF** $x > 0$
THEN ANY u **WHERE** $u \in 1..10$ **THEN** $z, y := z + u, y + 1$ **END**
ELSE ANY u **WHERE** $u \in 1..10$ **THEN** $z, y := z + u, y - 1$ **END**
END

抽象机的证明义务

现在讨论抽象机包含带来的证明义务

设 M 包含（和/或扩充）若干其他抽象机，由 M 生成的证明义务自然应该是被包含抽象机的证明义务和 M 本身的证明义务的某种组合

为方便讨论，下面讨论中总以抽象机 M 包含抽象机 M_1 的情况作为讨论对象，包含多个抽象机的情况是下面讨论的自然推论

前面说过，如果 M_1 有参数而 M 包含了 M_1 的实例，给 M_1 的实参就必须满足 M_1 里 **CONSTRAINTS** 提出的要求，这是 M 的一项证明义务

下面讨论 M 的各种证明义务，这是第一次正式地完整说明抽象机 M 的证明义务，把由于包含了抽象机而得到的信息和产生的需要放在一起考虑

抽象机的证明义务

1. 证明 M 参数要求能得到满足。设 M 的参数是 p， 其 **CONSTRAINTS** 是 C，产生的证明义务是

$$\exists p \,.\, C$$

也就是说，存在满足 C 的参数实例。这个证明不牵涉被包含抽象机， M 的 **CONSTRAINTS** 里不会引用被包含抽象机的信息

2. 证明 M 的集合和常量存在实例，存在满足其 **PROPERTIES** 的集合和常量

被包含抽象机的集合和常量也看作是 M 的集合和常量一起处理，要求存在集合和常量实例，满足 M 及其所有包含抽象机的 **PROPERTIES**，在证明这一要求可满足时，抽象机的 **CONSTRAINTS** 也作为前提

对 M 包含 M_1 的情况，设它们的集合、常量、**CONSTRAINTS** 和 **PROPERTIES** 分别为 st, c, C, P 和 st_1, c_1, C_1, P_1，证明义务是

$$C \wedge C_1 \Rightarrow \exists st, c, st_1, c_1 \,.\, (P \wedge P_1)$$

注意，在证明 M_1 时已经证明了 $C_1 \Rightarrow \exists st_1, c_1 \,.\, P_1$，这一结论可能有助于解决上面这个证明义务

抽象机的证明义务

3. 证明在 M 和 M_1 的 **CONSTRAINTS** 和 **PROPERTIES** 成立的条件下，机器的合法状态集合不空。也就是说，存在满足机器的不变式的状态

设 M 和 M_1 的不变式分别为 I 和 I_1，这一证明义务是

$$C \wedge P \wedge C_1 \wedge P_1 \Rightarrow \exists v, v_1 \,.\, (I \wedge I_1)$$

同样，对于 M_1 已经证明过 $C_1 \wedge P_1 \Rightarrow \exists v_1 \,.\, I_1$

4. 证明 M 的初始化代换满足不变式其不变式 I

由于 M 的 **INITIALISATION** 子句里可以引用 M_1 初始化后的状态和其他信息（参数、常量和集合），而且将先做 M_1 的初始化，再做 M 的初始化，初始化表现为两个顺序进行的代换

设 M 和 M_1 的初始化表达式分别为 Si 和 Si_1，M 在这里的证明义务是

$$C \wedge P \wedge C_1 \wedge P_1 \Rightarrow [Si_1 \,;\, Si]\, I$$

这里只需证明代换后满足 M 的不变式。M_1 初始化满足其不变式的证明已做过，M 的初始化不改变 M_1 的状态，因此不需要再考虑 I_1

抽象机的证明义务

5. 证明 M 的各个操作都维持其不变式 I

   如前所述，所有被 M 提升的操作，以及 M 扩充的抽象机的所有操作都需要做这个证明。虽然这些操作只改变被包含（或扩充）抽象机的状态，如 M_1 的状态，但 M 的不变式里可能引用 M_1 的状态，这种操作的作用可能被 M 的不变式看到，因此需要证明其不变式维持性

   在做这组证明时，所有被包含机器的不变式都可以用在证明中

   假设 M 的一个操作的代换是 **PRE** R **THEN** S **END**，相应的证明义务是

   $$C \wedge P \wedge I \wedge C_1 \wedge P_1 \wedge I_1 \wedge R \Rightarrow [S]\, I$$

   不需要证明 M 的操作对 M_1 的不变式 I_1 的维持性，因为这些操作不直接修改 M_1 的状态

抽象机证明义务的情况看起来很繁杂，但它们的构造和归约都是和机械的工作，不难用程序实现。任何 B 工具都包含自动生成所有证明义务的功能

B 工具在这一层面的工作，就是找出抽象机里与证明义务有关的各种成分，基于它们构造出需要证明的逻辑公式（证明义务），而后设法证明之

软件和物理世界

有些软件运行在一个抽象环境里。例如各种管理系统，被管理的对象都直接反映为各种数据，通过人或设备输入系统。软件系统运行中做的工作就是操作这些数据，做各种统计和计算

但是也有许多软件系统，它们在运行中需要不断地与外部的物理世界打交道。例如各种控制系统，它们需要

- 通过某些接口设备接受外部感应器来的信息
- 通过某些接口设备送出控制信号，驱动外部物理设备

在描述软件规范的时候，如果不描述软件外部的世界，写出的规范就不完整，这种情况下怎么能作为一个完整的系统来验证和检查？

在这方面，抽象机和抽象机组合机制可以帮我们的忙

我们可以考虑将与系统有关的物理世界的一部分封装为一部或几部抽象机，用这些抽象机填充系统中的空白，写出貌似 “完整” 的抽象机规范

下面举些例子说明这方面的一些可能做法

物理世界的封装

假设要描述的系统需要不断从两个传感器获得信息，其中一个的合法值范围是 0..100，另一个的值范围是 0..500。可以把这两个传感器封装为抽象机：

MACHINE $RWInput$ /* 这是一个无内部状态的抽象机 */
OPERATIONS
$res \longleftarrow senser1 = res :\in \mathbf{0..100};$
$res \longleftarrow senser2 = res :\in \mathbf{0..500}$
END

在系统规范中某个适当的抽象机里 **INCLUDES** 上面抽象机，就可以描述和验证系统的其他部分了，就像把外部世界包含到规范里似的

写这种抽象机时也要当心。例如采用上面的抽象机，就是假定

- 物理世界中的实际传感器的返回值确实在描述的范围内
- 整个规范系统的验证也是在这一假设下进行
- 如果系统实际中出现了检测值超出范围的情况，造成的后果不可预料

欧洲航天局的阿里亚娜五型火箭爆炸，可以认为就是这种情况造成的（加速度的物理检测值超出预定的范围）

物理世界的封装

另一种考虑是写出更为一般的规范，迫使使用规范的地方去验证结果的合法性，出现不合法输入的情况时必须专门处理

MACHINE $RWInput$
OPERATIONS
$res \longleftarrow senser1 = res :\in \mathbb{Z};$
$res \longleftarrow senser2 = res :\in \mathbb{Z}$
END

这样写模块，实际上迫使调用这两个操作的所有地方都进行检查

把输入中可能出现的 “错误” 隔离在尽可能靠近信息入口处，是保证软件安全的基本设计原则

VAR x **IN** $x \longleftarrow senser1$;
IF $\mathbf{0} \leq x \wedge x \leq \mathbf{100}$
THEN ... /* 正常情况下的处理 */
ELSE ... /* 系统输入出错情况下的处理 */
END
END

物理世界的封装

前面描述规范时，着重考虑了操作参数的条件和检查，在那里实际上认为系统的外来数据可能从操作的参数进入系统，需要防范

上面这样封装物理世界的模块是外来信息的另一种入口，同样要特别注意

另一种可能做法是把物理世界来的信息作为对抽象机的操作。这种看法大家已经很熟悉了，这里不再仔细讨论

设备操控也可以用类似方式模拟。例如需要加速/减速，刹车，停止，可以定义下面抽象机，其中所有操作都什么不做：

MACHINE $Actuator$
OPERATIONS
$speedUp(xx)$ = **PRE** xx ... **BEGIN** skip **END;**
$speedDown(xx)$ = **PRE** xx ... **BEGIN** skip **END;**
$brake(xx, yy)$ = **PRE** xx ... $\wedge$ yy ... **BEGIN** skip **END;**
$stopEngine$ = **BEGIN** skip **END;**
END

规范中需要做实际控制的地方就调用这些操作（包含这个模块）

模拟时间

软件里根本没有时间，任何时间概念都来自物理世界或者硬件设备。但我们写软件规范时（像写软件时一样），有时会希望使用时间概念

要模拟时间，先要考虑好是软件去找时间，还是时间来找软件

前一种情况，在程序里的体现是调用系统提供的时间库函数。后一种情况也有，就是时钟中断，由硬件时钟定时触发程序

要模拟外来的自动的时钟信号，实际上可以定义一个操作，并认为这一操作将会不断自动地被环境调用，在操作了描述系统的相应行为

模拟调用时钟函数，可以定义一个表示时间功能的抽象机

定义这一抽象机时，有一些特殊问题需要考虑

- 时间具有单向性，后一次调用函数得到的时间应该在前一次之后
- 时间具有 “随机性”，或说我们调用时间函数的间隔具有随机性
- 还需考虑具体情况的需要

模拟时间

一个可能的模拟时间的模块（例如模拟过去的秒数）：

MACHINE $RWInput$
VARIABLES $past$
INVARIANT $past : \mathbb{N}$
INITIALISATION $past := 0$
OPERATIONS
$res \longleftarrow time =$
BEGIN $res := past \ ||$
$past :\in \{y \mid y : \mathbb{N} \wedge past < y \wedge y < past + 10\}$
END
END

用这一模块模拟时间，肯定会带来一个无法证明的证明义务（自然数越界），但这个问题十分清楚

这只是一个简单的处理，如果实际需要，我们可以按照这种思路写一个更符合需要的模拟时间的抽象机，在写系统规范时使用

其他抽象机组合机制

INCLUDES（和 **EXTENDS** 等）用于基于较小的抽象机构造更大的抽象机，被 **INCLUDES** 的抽象机成为新抽象机的组成部分，完全被新抽象机控制

这只是构造大型系统的一种方式：基于组件装配更大的组件。只有这种构造机制是不够的，B 方法还提供了其他构造机制

B 方法提供了另外两种构造机制是 **SEES**（查看）和 **USES**（使用），它们都是用于建立不同的独立抽象机之间的联系，使一些抽象机可以去“访问”另一些抽象机的信息，使用被联系的抽象机里的各种信息

两者的相同点和不同点：

- **SEES** 和 **USES** 都用在一个抽象机的规范里，说明这一抽象机需要参考另外的一个或多个抽象机里的信息
- **SEES** 描述的关系更为松弛，定义的抽象机与被 **SEES** 的抽象机的状态之间没有关系，只是在其一些部分引用了被 **SEES** 的机器的信息
- 通过 **USES** 建立的关系比通过 **USES** 建立的关系更强一些，允许定义的抽象机的 **INVARIANT** 里引用被 **USES** 机器的状态，实际上是建立了两个抽象机的状态之间的联系

SEES－查看

可以让一部抽象机（如 M）以 “只读” 方式查看另一抽象机（如 M_1），建立一个 “查看” 关系。为此需要在 M 的规范里写一个 **SEES** 子句

SEES M_1

如果 M_1 是带参数的机器，**SEES** 子句不描述 M_1 的参数，M 也不能访问 M_1 的参数（这些参数可能由某个 **INCLUDES** M_1 的机器提供）

SEES 关系下 M 可以以只读方式访问 M_1 里的信息，包括访问其集合、常量和状态。在 M 的 **PROPERTIES**、 **INITIALISATION** 和不变式里可以引用 M_1 的集合和常量；在 M 操作的前条件和体里都可以引用 M_1 的集合和常量，还可以以只读方式引用 M_1 的变量

当 M **SEES** M_1 时，M 并不控制 M_1（与 **INCLUDES** 不同），这样在 M 两次访问 M_1 状态之间可能出现对 M_1 操作的调用，导致 M_1 的状态变化

由于这种情况，M 的不变式不能引用 M_1 的状态。因为对 M_1 操作的调用不受 M_2 的控制，因此 M_1 的状态变化不受 M 的控制。如果 M 的不变式引用了 M_1 的变量，只局部检查 M 本身就无法保证 M 不变式维持性

SEES－查看

被查看机器与查看它的机器是相互独立的机器

- 假设另一抽象机 M' **SEES** M。虽然现在 M **SEES** M_1，但 M' 并不自动 **SEES** M_1，也就是说 **SEES** 关系并不传递
- 如果 M' 要查看 M_1，那就必须在 M' 里明确地写 **SEES** M_1
- 如果 M_1 包含了其他机器（例如包含了 M_2），M 访问 M_2 的集合、常量和变量的规则与它访问 M_1 相应成分的规则完全一样

使用 **SEES** 的一种典型情况是系统里需要某个（或某些）待定集合或枚举集合，在整个系统里的许多地方都需要它（们）

一般而言，大型规范通常是由基于包含关系的、分层定义许多抽象机构成

- 每个机器有自己的局部集合和常量，以及自己的状态
- 不同抽象机的集合、常量和状态互不相交

但可能许多机器都需要用某个（某些）集合，这时就可以用 **SEES** 构造

SEES－实例

假定要描述一个零售店系统，它可能包含许多抽象机，它们互不包含：

- 抽象机 *Price* 跟踪各种价格的情况
- 抽象机 *Shop* 描述商店的活动
- 抽象机 *Customer* 表示顾客的购买行为

这几个机器都要用到一个公共集合 *GOODS*。问题是该集合在哪里描述？

- 它不适合放在各个机器里面，因为系统里不应该有该集合的多份拷贝
- 它不能放在被上述几部机器包含的机器里，被几个机器包含需要分别实例化（并可能需要重命名），同样会产生重复的拷贝
- 如果放在一部机器里或被一部机器包含，其他机器就不能看到它

如果只能用包含方式构造，解决问题唯一的方法是写一个很大的抽象机描述整个系统，这样就不能得到模块化和结构化组织的优势了

SEES 关系可以用于解决这种问题

实例：零售系统 Retail

本实例只是为了表现 **SEES** 的使用，并不特别关心系统设计的合理和有用性

先引进一个表示商品的简单机器：

MACHINE *Goods*
SETS *GOODS*
END

其中待定集合 *GOODS* 表示被考虑的商品。完全可以用另一个变量集合，表示正在销售的商品

这里的 *GOODS* 是个静态确定的集合，其他机器可以通过 **SEES** 这个机器查看这一信息，它们不需要控制这种信息，更不需要改变它

也就是说，这样定义的一部机器只是简单地把这个系统里的一部分公用信息分离出来，系统的其他部分可以通过 **SEES** 连接关系来使用这部分信息

例如，商店 *Shop* 和客户 *Customer* 可能需要共享商品目录信息，它们就可以 **SEES** 这个 *Goods* 机器

实例：零售系统 Retail

零售系统还需要报价，考虑定义下面机器：

MACHINE $Price$
SEES $Goods$
VARIABLES $price$
INVARIANT $price \in GOODS \rightarrow \mathbb{N}_1$
INITIALISATION $price :\in GOODS \rightarrow \mathbb{N}_1$
OPERATIONS
$setprice(item, pr) =$
PRE $item \in GOODS \wedge pr \in \mathbb{N}_1$
THEN $price(item) := pr$
END;
$pr \longleftarrow queryprice(item) =$
PRE $item \in GOODS$
THEN $pr := price(item)$
END
END

抽象机 $Price$ 说明它 **SEES** 抽象机 $Goods$。因此在其规范里可以引用集合 $GOODS$

这个机器里几个部分都引用了集合 $GOODS$

这个机器很简单。如前所述，这里并不特别关注机器的设计，主要关心机器之间的 **SEES** 关系

实例：零售系统 Retail

考虑抽象机 $Shop$，它 **SEES** 两部抽象机 $Goods$ 和 $Price$，使用其中信息

MACHINE $Shop$
SEES $Goods, Price$
VARIABLES $takings$
INVARIANT $takings \in \mathbb{N}$
INITIALISATION $takings := 0$
OPERATIONS
$sale(item) =$
PRE $item \in GOODS$
THEN $takings := takings + price(item)$
END;
$tt \longleftarrow total = tt := takings$
END

抽象机 $Shop$ 说明它 **SEES** 抽象机 $Goods$ 和 $Price$

这个机器里引用了集合 $GOODS$，其操作里还引用 $Price$ 里的变量 $price$

注意：后面这一引用是只读引用，这是允许的

实例：零售系统 Retail

最后一个抽象机是 *Customer*，它也引用了 *Goods* 和 *Price*

MACHINE
  Customer
SEES *Goods*, *Price*
CONSTANTS *limit*
PROPERTIES $limit \in GOODS \rightarrow \mathbb{N}_1$
VARIABLES *purchases*
INVARIANT $purchases \in \mathbb{P}(GOODS)$
INITIALISATION $purchases := \{\}$
OPERATIONS
  $pr \longleftarrow buy(item) =$
  PRE $item \in GOODS \wedge price(item) \leq limit(item)$
  THEN $purchases := purchases \cup \{item\}$ ||
    $pr := price(item)$
  END
END

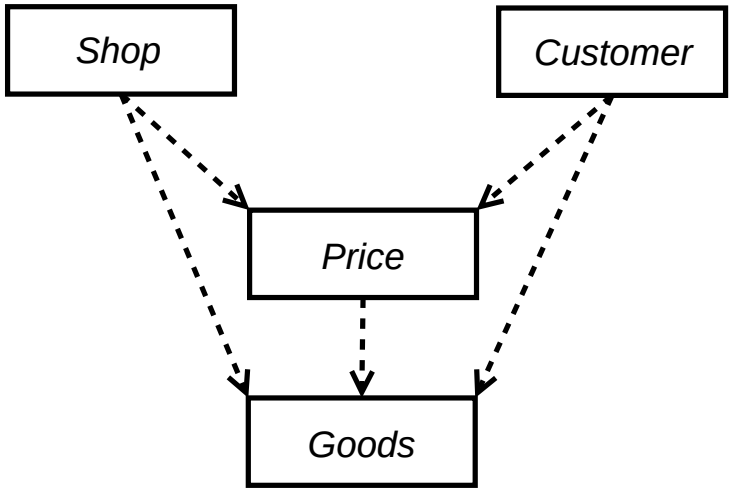
Customer 也 **SEES** 机器 *Goods* 和 *Price*

机器里引用了 *Goods* 的集合 *GOODS*，操作里引用 *Price* 的变量 *price*

这里的常量 *limit* 可以看作顾客对各种商品的预期可接受价格

实例：零售系统 Retail

这四个抽象机之间的关系如下图所示：



其中的虚线箭头表示 **SEES** 关系

USES－使用

如果要在抽象机 M 里使用抽象机 M_1，需要在 M 里写

USES M_1

在此之后在 M 里用 M_1 的方式与 M **SEES** M_1 差不多

仅有的不同之处在于现在 M 的不变式里可以引用 M_1 的状态（引用 M_1 的变量），这样就可以建立 M 的状态与 M_1 的状态之间的联系

注意，现在 M 不控制 M_1，M_1 完全独立于 M，这实际上表示 M 的状态依赖于系统里的其他部分（M_1），这种依赖性在软件系统里也很常见

如果发现软件中某些部分依赖于系统里的其他部分，而两者又相互独立，就可以用 **USES** 描述两部分之间的关系

M 对 M_1 的状态依赖会带来新的问题

下面分析这方面问题

USES

如常，M 的操作需要维持 M 的不变式，但现在出现了新情况：

- M 的不变式里可能引用了 M_1 的状态
- M_1 的状态可能由于 M_1 的操作被调用而改变，而这种调用不受 M 的控制（因为 M_1 是另一个独立模块）
- M_1 的操作改变了 M_1 的状态，这种改变会不会打破 M 的不变式？

这种情况有可能出现。但 M_1 是已经开发完成的抽象机，不应把它维持（将来的任何）使用方 M 的不变式拿回来作为 M_1 的证明义务

另一方面，M 也不应把 M_1 的操作维持自己的不变式作为证明义务

但这一证明必须在某个层面上考虑。B 方法里解决这一问题的想法是：

- M 和 M_1 是一个系统的两个模块，它们最终会被某抽象机 M' 包含，这样，M' 就将控制 M 和 M_1 的执行
- 在考虑 M' 的证明义务时，需要包括保证 M' 里对 M_1 的操作的使用都能维持 M 的不变式（并不要求普遍意义的维持性）

USES

USES 关系用于建立不同抽象机的状态之间的某些实际系统需要的联系

这种必要的联系可以沿着抽象机的包含关系链传递，直到某个可以保证这种相互关联的逻辑关系成立的抽象机，在那里解决不变式问题

如果一个抽象机 **USES** 其他抽象机，那么

- 它可能无法维持自己的不变式（因为不变式还依赖于其他抽象机的状态）
- 这实际上说明，使用 **USES** 抽象机的不变式维持性出现了一些待定条件，这些条件需要系统开发的后面阶段来保证
- 这种情况和 **INCLUDES** 或 **SEES** 都不同（**INCLUDES** 关系中被包含机器的操作被包含机器完全控制着，而 **SEES** 关系中被查看机器的操作不会影响查看它的机器的不变式）

我们应该根据 **USES** 的这种特点考虑它的使用

SEES, USES—证明义务

在机器 M 里写了 **SEES** 或者 **USES** 子句，并不给 M 带来新的证明义务，但会影响原本要做的一些证明义务

假设抽象机 M **SEES** 或者 **USES** 抽象机 M_1，其证明义务有如下情况：

1. 由关于 M 的参数 p 的 **CONSTRAINTS** C 带来的证明义务，要求证明存在满足 **CONSTRAINTS** 的参数。与前面 **INCLUDES** 的证明义务一样

   $$\exists p \,.\, C$$

2. 需要证明存在满足 M 的 **PROPERTIES** 条件的集合和常量实例。在做这一证明时，被 M 查看或使用的抽象机的都可以作为证明的前提

   假定 M 的集合、常量和 **PROPERTIES** 分别是 st, c, P，M_1 的 **PROPERTIES** 是 P_1，有关 M 的 **PROPERTIES** 的证明义务是

   $$C \wedge C_1 \wedge P_1 \;\Rightarrow\; \exists st, c \,.\, P$$

SEES, USES－证明义务

3. 证明当所有 **CONSTRAINTS** 和 **PROPERTIES** 都成立的情况下，确实存在满足不变式的状态。设 M 的变量和不变式分别是 v 和 I，而 M_1 的变量和不变式是 v_1 和 I_1，如果 M **SEES** M_1，那么 M 的不变式不引用 M_1 的状态，证明义务是

$$C \wedge P \wedge C_1 \wedge P_1 \Rightarrow \exists v \,.\, I$$

如果是 **USES** M_1，不变式 I 与 M_1 的状态有关，因此要求证明

$$C \wedge P \wedge C_1 \wedge P_1 \Rightarrow \exists v, v_1 \,.\, (I_1 \wedge I)$$

4. 证明 M 的初始化代换 **INITIALISATION** 建立的状态满足不变式

由于 M 的初始化代换可以查看 M_1 的状态（无论是 **SEES** 还是 **USES**关系），因此要求先做 M_1 的初始化

假定 M 和 M_1 的初始化代换分别是 Si 和 Si_1，这里的证明义务是

$$C \wedge P \wedge C_1 \wedge P_1 \Rightarrow [Si_1 \,;\, Si]\, I$$

SEES, USES－证明义务

5. 最后是证明 M 的所有操作都能维持 M 的不变式

被 **SEES** 或 **USES** 的抽象机能为这些证明提供更多信息，包括有关它们的参数、集合或常量的限制和性质的信息（由 C_1, P_1），还有有关它们的状态的信息（由不变式 I 描述）

假定要证明的 M 操作的代换是 **PRE** R **THEN** S **END**，相应证明义务是

$$C \wedge P \wedge I \wedge C_1 \wedge P_1 \wedge I_1 \wedge R \Rightarrow [S]\, I$$

如果是 M **SEES** M_1，由于 I 里不涉及 M_1 的变量，因此 I_1 对 I 的证明不会有任何贡献，可以将 I_1 从上式的前件中删除

如果是 M **USES** M_1，不变式 I 可能引用 M_1 的状态，描述自己的状态与 M_1 的状态之间的关系

即使在这种情况下，M 的证明义务也只证明自己的操作维持自己的不变式 I。M_1 的操作与 M 的不变式 I 的关系推迟到上层的（以后写的）同时包含 M 和 M_1 的那个机器里处理

实例：货单系统（B Book，8.1节）

用一个较大的系统说明如何做递增式开发，用到前面讨论的抽象机组合功能

现在要开发一个系统规范，该系统能用于在商业环境里成立客户货单，其中管理一些实体：客户、产品、货单和条目（货单条目）。简介如下：

- 一个客户在系统里有一条记录，其中信息包括客户类别：正常normal、可疑dubious或者朋友friend。每个客户有最大容许值allowance，是发给该客户的货单的价值上限。不同类别的客户有不同的折扣率
- 一种货品有一个记录，其中记录其价格price，状态status，有货（avaliable）或者缺货（soldout），及其代用品（substitute，如果有）。代用品是另一货物且当时有货
- 货单有明确的客户，包括某个特定的折扣率percentage，还有最大容许值allowed。这两个属性在创建时给定
- 某个货单条目针对一种特定货品article，包括其数量quantity和单价unit_cost。后面这项来自货品的price

功能：创建/修改客户，创建/修改产品，创建/销毁货单，向货单加条目

货单系统

系统需要满足一些规则，下面是规则的非形式描述：

1. 售缺的货品不能加入任一货单
2. 如果售缺货品存在替代品，系统自动用替代品代替原货品
3. 同一货单里不存在对应同一货品的两个或两个以上条目
4. 不给可以客户开货单
5. 一张货单的货品总值不大于货单的容许值
6. 朋友可以得到 20% 的折扣，其他客户无折扣

还要建立完善的错误报告：

- 货品售缺又无代用品时（法则1/2）
- 想加入的货品或代用品已在货单里（法则2/3）
- 想加入新货品或代用品将使总值超过容许值（折扣后）（法则2/5/6）
- 商品编号，客户编号等用完（因为都是有穷集合）

货单系统：机器 Client

下面的想法是通过一组逐渐构造出来的抽象机，作为系统规范的组件，最后将所有抽象机组合起来做出系统的规范（对书上的机器有些修改）

首先用一部抽象机封装客户，其中定义几个集合：可能客户的集合 CLIENT，客户名字集合 CLIENT_NAME 和客户类别集合 CATEGORY：

MACHINE *Client*

SETS

$CLIENT$;
$CLIENT\_NAME$;
$CATEGORY = \{friend, dubious, normal\}$

CONSTANTS

$discount, init\_allow$

PROPERTIES

$discount \in CATEGORY \rightarrow (\mathbf{0..100}) \wedge$
$discount = \{friend \mapsto \mathbf{80}, dubious \mapsto \mathbf{100}, normal \mapsto \mathbf{100}\} \wedge$
$init\_allow \in \mathbb{N} \wedge$ /* 没有这条，可能认为它是 $\mathbb{Z}$ */
$init\_allow = \mathbf{500}$

货单系统：机器 Client

机器 Client 的状态

VARIABLES

$client, category, allowance, cname$

INVARIANT

$client \subseteq CLIENT \wedge$
$category \in client \rightarrow CATEGORY \wedge$
$allowance \in client \rightarrow \mathbb{N} \wedge$
$cname \in client \rightarrow CLIENT\_NAME$

INITIALISATION

$client, category, allowance, cname := \varnothing, \varnothing, \varnothing, \varnothing$

我们可能定义一些与客户有关的操作，包括加入客户，修改其容许值，修改其类别等等

具体定义哪些操作，可以根据需要设计

货单系统：机器 Client

加入客户的操作，任选一个客户（客户号）。可以认为 CLIENT 是允许的客户号集合，client 是已经使用的客户号集合

OPERATIONS

$cc \longleftarrow create\_client(cnm) =$
PRE
$cnm \in CLIENT\_NAME \wedge$
$cc \in CLIENT \wedge$
$client \neq CLIENT$ /* 尚且有剩余的客户号 */
THEN
ANY $c1$ **WHERE** $c1 \in CLIENT - client$ **THEN**
$client := client \cup \{c1\} \;||$
$category(c1) := normal \;||$
$allowance(c1) := init\_allow \;||$
$cname(c1) := cnm \;||$
$cc := c1$
END
END;

货单系统：机器 Client

另外几个操作都很简单：

$modify\_category(cc, cat) =$
PRE $cc \in CLIENT \wedge cc \in client \wedge cat \in CATEGORY$
THEN $category(cc) := cat$
END;

$modify\_allowance(cc, al) =$
PRE $cc \in CLIENT \wedge cc \in client \wedge al \in \mathbb{N}$
THEN $allowance(cc) := al$
END;

$cnm \longleftarrow read\_client(cc) =$
PRE $cc \in CLIENT \wedge cc \in client$
THEN $cnm := cname(cc)$
END
END

货单系统：机器 Product

产品机器 Product 封装有关货品的各种属性

MACHINE $Product$
SETS $PRODUCT; PRODUCT\_NAME; STATUS = \{available, sold\_out\}$
VARIABLES
$product, pname, price, status, substitute$
INVARIANT
$product \subseteq PRODUCT \wedge$
$pname \in product \rightarrow PRODUCT\_NAME \wedge$
$price \in product \rightarrow \mathbb{N} \wedge$
$status \in product \rightarrow STATUS \wedge$
$substitute \in product \nrightarrow product \wedge$
$substitute \in product \nrightarrow status^{-1}[\{available\}]$
INITIALISATION
$product, pname, price, status, substitute := \varnothing, \varnothing, \varnothing, \varnothing, \varnothing$

注意：只有 available 的产品可以作为替代品，而且并不是每件产品都有替代品。上面不变式里描述了这些情况

货单系统：机器 Product

OPERATIONS
$prd \longleftarrow create\_product(pn, pri) =$
PRE $pn \in PRODUCT\_NAME \wedge pri \in \mathbb{N} \wedge product \neq PRODUCT$
THEN
ANY $p1$ **WHERE** $p1 \in PRODUCT - product$ **THEN**
$prd := p1 \parallel product := product \cup \{p1\} \parallel$
$pname(p1) := pn \parallel price(p1) := pri \parallel$
$status(p1) := available$
END
END;

$modify\_price(prd, pri) =$
PRE $prd \in PRODUCT \wedge prd \in product \wedge pri \in \mathbb{N}$
THEN
$price(prd) := pri$
END;

同样可能出现产品号用完的情况，应该有错误报告

货单系统：机器 Product

$make\_unavailable(prd) =$
PRE $prd \in PRODUCT \wedge prd \in product$
THEN $status(prd) := sold\_out \parallel substitute := substitute \rhd \{prd\}$ **END;**

$make\_available(prd) =$
PRE $prd \in PRODUCT \wedge prd \in product$
THEN $status(prd) := available$ **END;**

$assign\_substitute(prd, prd1) =$
PRE $prd \in PRODUCT \wedge prd1 \in PRODUCT \wedge prd \in product \wedge$
$prd1 \in product \wedge status(prd1) = available$
THEN $substitute(prd) := prd1$ **END;**

$pn \longleftarrow read\_product(prd) =$
PRE $prd \in PRODUCT \wedge prd \in product$
THEN $pn := pname(prd)$ **END**
END

注意 *make_unavailable* 里对 *substitute* 的修改

货单系统：机器 Invoice

货单机器使用（**USES**）机器 Client 和 Product，它要访问这两个机器里的一些变量等。局部集合是 INVOICE（货单）和 LINE（货单行）

这一机器了定义了许多变量

MACHINE
Invoice
USES
Client, *Product*
SETS
INVOICE; *LINE*
VARIABLES
invoice, *customer*, *percentage*, *allowed*, *total*,
line, *origin*, *article*, *quantity*, *unit_cost*
INITIALISATION
$invoice, customer, percentage, allowed, total := \varnothing, \varnothing, \varnothing, \varnothing, \varnothing \parallel$
$line, origin, article, quantity, unit\_cost := \varnothing, \varnothing, \varnothing, \varnothing, \varnothing$

集合 INVOICE 代表所有可能货单，LINE 是可能的货单行

货单系统：机器 Invoice

OPERATIONS

$inv \longleftarrow create\_invoice\_header(clt) =$

PRE $clt \in CLIENT \wedge clt \in client \wedge$
$category(clt) \neq dubious \wedge invoice \neq INVOICE$

THEN

ANY $invnew$ **WHERE** $invnew \in INVOICE - invoice$

THEN

$invoice := invoice \cup \{invnew\} \;||$
$customer(invnew) := clt \;||$
$percentage(invnew) := discount(category(clt)) \;||$
$allowed(invnew) := allowance(clt) \;||$
$total(invnew) := \mathbf{0} \;||$
$inv := invnew$

END

END;

客户要求购买货品时，先为客户创建一个货单头，其中没有货单项。只允许给非可疑客户创建货单头。这里还要求存在闲置的货单（编号）

货单系统：机器 Invoice

删除货单的操作，牵涉到删除该货单的所有货单行（注意，lns 是集合）

$remove\_invoice(inv) =$

PRE $inv \in INVOICE \wedge inv \in invoice$

THEN

LET lns **BE** $lns = origin^{-1}[\{inv\}]$ **IN**

$invoice := invoice - \{inv\} \;||$
$line := line - lns \;||$
$customer := \{inv\} \lhd customer \;||$
$percentage := \{inv\} \lhd percentage \;||$
$allowed := \{inv\} \lhd allowed \;||$
$total := \{inv\} \lhd total \;||$
$origin := lns \lhd origin \;||$
$article := lns \lhd article \;||$
$quantity := lns \lhd quantity \;||$
$unit\_cost := lns \lhd unit\_cost$

END

END;

货单系统：机器 Invoice

添加货单行的操作

$ln \longleftarrow new\_line(inv, prd) =$
PRE
$inv \in INVOICE \wedge prd \in PRODUCT \wedge$
$inv \in invoice \wedge prd \in product \wedge$
$status(prd) = available \ \wedge (inv, prd) \neq \mathsf{ran}(origin \otimes article)$
THEN
ANY $newln$ **WHERE** $newln \in LINE - line$
THEN
$line := line \cup \{newln\}$ ||
$origin(newln) := inv$ /* 本行属于哪个货单 */ ||
$article(newln) := prd$ /* 本行的货品 */ ||
$quantity(newln) := \mathbf{0}$ /* 货品数量暂设为 0 */ ||
$unit\_cost(newln) := (price(prd) \times percentage(inv)/\mathbf{100})$ ||
$ln := newln$
END
END;

货单系统：机器 Invoice

修改货单行，修改购买数量的操作

$modify\_line(ln, qt) =$ /* qt is the new quantity for the ln */
PRE
$ln \in LINE \wedge ln \in line \wedge qt \in \mathbb{N} \wedge$
$status(article(ln)) = available \wedge$
$total(origin(ln)) - quantity(ln) \times unit\_cost(ln) + qt \times unit\_cost(ln)$
$\leq allowed(origin(ln)) \wedge$
$total(origin(ln)) - quantity(ln) \times unit\_cost(ln) \geq \mathbf{0}$
THEN
$quantity(ln) := qt$ ||
$total(origin(ln)) := total(origin(ln)) - quantity(ln) \times unit\_cost(ln) +$
$qt \times unit\_cost(ln)$
END;

条件和代换里都要计算修改后的总价值

货单系统：机器 Invoice

查找行和删除一个货单里的所有货单行的函数

$ln \longleftarrow the\_line(inv, prd) =$
PRE $inv \in INVOICE \wedge inv \in invoice \wedge prd \in PRODUCT \wedge$
$prd \in product \wedge status(prd) = available \wedge ln \in LINE \wedge$
$(inv, prd) \in \mathsf{ran}(origin \otimes article)$
THEN $ln := (origin \otimes article)^{-1}(inv, prd)$ **END;**

$remove\_all\_lines(inv) =$
PRE $inv \in INVOICE \wedge inv \in invoice$
THEN
LET lns **BE** $lns = origin^{-1}[\{inv\}]$ **IN**
$line := line - lns \;||\; origin := lns \lhd origin \;||$
$article := lns \lhd article \;||\; quantity := lns \lhd quantity \;||$
$unit\_cost := lns \lhd unit\_cost$
END
END
END

货单系统：机器 Invoice_System

把前面机器都包含到一个机器里，EXTENDS 提升所有成员机器的操作

MACHINE
$Invoice\_System$
EXTENDS
$Client, Product, Invoice$
OPERATIONS
$res \longleftarrow some\_clients\_exists = res := \mathsf{bool}(client \neq \varnothing);$
$res \longleftarrow clients\_not\_saturated = res := \mathsf{bool}(client \neq CLIENT);$
$res \longleftarrow client\_not\_dubious(clt) =$
PRE $clt : CLIENT$ **THEN** $res := \mathsf{bool}(category(clt) \neq dubious)$ **END;**
$res \longleftarrow some\_product\_exists = res := \mathsf{bool}(product \neq \varnothing);$
$res \longleftarrow products\_not\_saturated = res := \mathsf{bool}(product \neq PRODUCT);$
$res \longleftarrow product\_avaliable(prd) =$
PRE $prd : PRODUCT$ **THEN** $res := \mathsf{bool}(status(prd) = available)$ **END**
END

总结

这两次课主要讨论了两个问题：

1. 大型抽象机的模块构造
   - 三种机制，**INCLUDES**, **SEES**, **USES**
   - **INCLUDES** 实现实质性包含，被包含抽象机作为正在定义的抽象机的一部分，可以通过 **PROMOTES** 或者 **EXTENDS** 将被包含抽象机的接口提升为所定义抽象机的接口操作。提升将带来新的证明义务
   - **SEES** 和 **USES** 实现抽象机之间的信息参考，**USES** 的特点是可以建立两方抽象机之间的状态关联，但带来新的证明义务（将来组合时）
2. 抽象机的证明义务，**CONSTRAINTS**, **PROPERTIES**, **INVARIANT** 都产生证明义务，证明了所有的证明义务，就保证了抽象机的内在一致性

本节还讨论了一些技术，并用实例说明了上述机制的使用