形 式 化 方 法:

基于 B 方法的严格软件开发

(13)

系统结构和实现

裘宗燕

北京大学数学学院信息科学系

2010年春季

概要

本部分讨论与 B 软件项目开发有关的一些问题，并通过一些简单的例子，展示开发中的一些情况。相关内容包括

- 模块之间的各种连接关系
- B 里的模块化开发过程，开发中 B 模块的创建和连接关系
- 通过实例介绍实现的技术，包括
  - 简单的精化实现
  - 通过 **IMPORTS** 封装具有脆弱性的模块，实现强健的模块
- 讨论实现步骤的数据精化问题
  - 一般情况
  - 通过实例介绍其中的考虑和一些问题

B 规范的组织

现在讨论 B 规范的组织和信息共享问题。这部分内容可以参考《B Book》第 12.2 节 “共享” 和 12.4 节 “多重精化和实现”，以及 Atelier B 语言手册的第 8 节 Architecture（体系结构）

基于 B 的一个完整开发在 Atelier B 里就是一个 project（项目），其中用一组 components 建模一个系统，最终的目标是产生一个可执行程序

前面的讨论都是关于如何保证所开发系统的功能安全性（safety），也就是说，保证系统运行中不出现我们不希望出现的状态

现在要研究的是基于 B 方法的项目的概念来构造软件的方法

在基于B 方法的开发中，我们用一个 B 模块模拟软件里的一个子系统，模块由一些 B 组件构成。B 方法里的组件有三种：抽象机，精化和实现

一个 B 模块总有一个抽象机作为它的规范（抽象规范）；它可以有一个实现（机器），还可能有作为其规范和实现之间的桥梁的若干个精化（机器）。进一步说，它还可能有一段代码

B 模块分类

Atelier B 规范提出根据性质把 B 模块分为 3 类：1) 从某个抽象机器出发经过一系列精化开发出来的模块；2) 基本模块；3) 抽象模块。下表是一些情况

性质\模块	开发的模块	基本模块	抽象模块
有无抽象机器	有	有	有
有无实现	有	无	无
有无代码	有	有	无

其中开发的模块的代码通过翻译自动生成，而基本模块的代码由手工写出

下面分门别类讨论几种模块的情况

基本模块:

情况简单，又称基本机器。它们只有一个抽象机。基本模块只能被其他模块导入，作为系统结构图最下面的叶子结点

基本模块必须有关联的代码，这些代码不是由抽象机翻译得到的，而是直接实现的。基本机器通常用作已有代码或某些底层服务的接口，相应功能存在于 B 语言之外。系统提供的输入输出接口函数是典型的例子

B 模块分类

抽象模块:

抽象模块的基本用途是供其他模块（抽象机或精化）**INCLUDES** 以享用其中描述的信息。这种只是作为抽象开发的一种媒介

开发的模块:

这是本课程讨论的主题。首先，有一个抽象机描述了这一模块的规范。这时 B 语言被作为规范语言使用，外部模块只能通过抽象机定义的接口使用这个模块。从外部使用的观点看，B 模块就等于它的抽象机

一部抽象机的精化也是一个部件，它维持了抽象机的接口和行为，但可能重新构造了操作和内部状态，代以更具体的变量。在精化时，抽象机的集合和具体数据都自动保留；其他数据可以精化，即，可以保留、替换或抛弃；也可以引入新的数据。一个精化还可以被下一个精化所精化

一个实现是一个 B 部件，是从一部抽象机出发逐步精化的最后结果，用 B 语言的子集 B0 写出。实现里的数据都能直接对应到实际计算机语言的数据表示，操作体必须用可以直接翻译到高级语言程序的语句（或指令）形式

B 软件项目

一个完整的 B 项目由一集 B 模块实例构成，这些模块实例通过某些连接相互关联成为一体，有关连接要遵循一些规则（下面讨论）

一个模块实例就是某部抽象机的一个拷贝

- 通过实例化，一部抽象机可以在同一项目里多次使用
- 一部抽象机的多个实例具有各自独立的数据空间，这由该抽象机里定义的可修改数据（抽象变量和具体变量）确定
- 抽象机里的常量由抽象机确定，在其所有实例之间共享
- 从一个抽象机实例出发调用操作时，操作中使用的是该实例的变量

需要区分抽象实例和具体实例

- 抽象实例是在规范阶段通过 **INCLUDES** 在抽象数据空间里创建的
- 具体实例是在实现阶段创建的，实际构成了项目开发出的程序的具体数据空间（下面讨论 **IMPORTS** 连接时还会讨论）

抽象机实例

每个抽象机实例都有自己特殊的名字

- 如果没有重命名，实例的名字就是抽象机的名字
- 如果重命名，实例的名字就是重命名前缀加“.”再加抽象机名
- 如果一部抽象机在项目里实例化多次，就必须重命名它们

如果没有重命名，实例的名字就是抽象机的名字（这种情况很常见）。但抽象机实例和抽象机是两种东西，不能搞混了

重命名一个抽象机实例，也就重命名里实例里的所有变量（无论抽象变量或具体变量）和操作，使用时必须采用加重命名前缀的写法

但重命名对集合和常量没有影响，一部抽象机里定义的集合和常量由该抽象机的所有实例共享（无论它们是否重命名）

抽象机之间的连接

各个层次的抽象机之间有多种不同的连接，现在讨论一些相关情况

INCLUDES 连接:

INCLUDES 连接一个 B 组件 MN（抽象机或精化）和一个机器实例 M_{inst}，使 MN 内部包含实例 M_{inst} 的所有成分。**INCLUDES** 是抽象层的连接，用于实现抽象机规范的模块化描述

USES 连接:

当一个 B 组件 MN（抽象机或精化）**INCLUDES** 了若干机器实例时，可能一些被包含实例需要参考另一机器里的信息。这时应该让这些实例 **USES** 该被参考的机器。**USES** 连接只能出现在几个被平行地包含的机器之间

INCLUDES 和 **USES** 都是抽象层组件之间的连接，在实现机器里（在实现层次）不能用 **INCLUDES** 和 **USES**

抽象机之间的连接

IMPORTS 连接：

IMPORTS 建立实现 MN 和一个抽象机器 M 的实例之间的连接，用于创建 M 的一个具体实例（M 的实现的实例），并完全地占用其服务

MN 称为 M 的父模块，它完全控制 M 实例里的数据使用（通过调用 M 的操作）。这样 **IMPORTS** 连接用于层次性地构造基于 B 项目开发的软件：一个模块的实现通过 **IMPORTS** 导入其他提供低层服务的模块

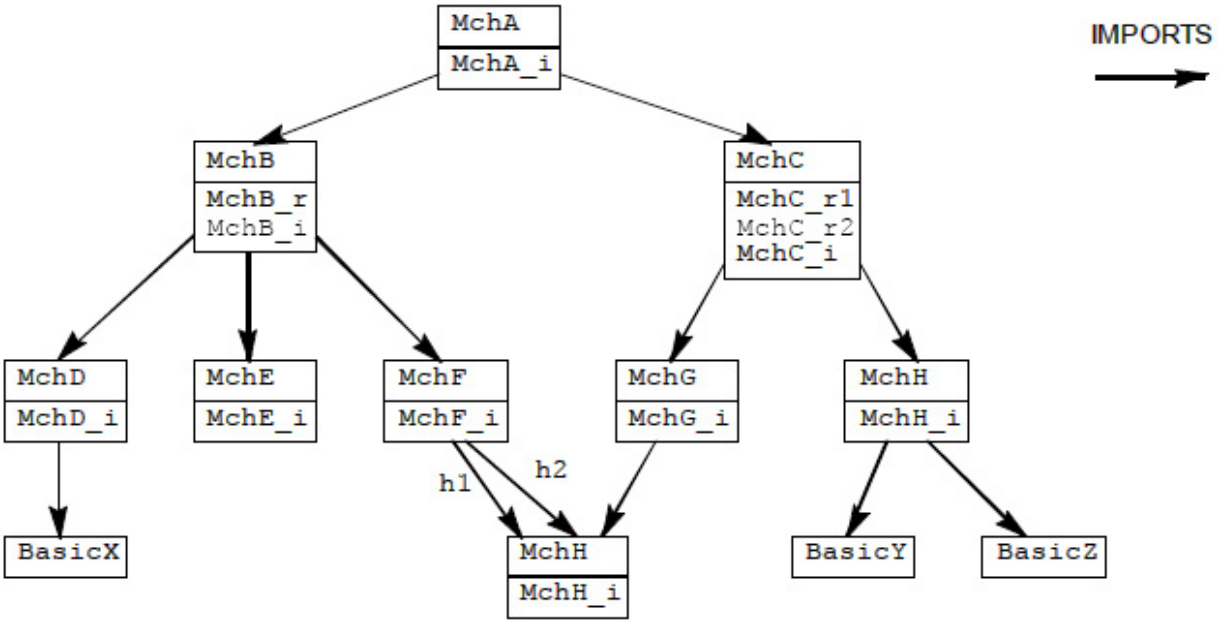
B 项目的 **IMPORTS** 图描述 B 部件的实现之间的 **IMPORTS** 关系，其中利用在 **IMPORTS** 连接上加重命名标记的方式，说明被导入的机器实例

在一个 **IMPORTS** 图中有一个特殊的根结点作为整个项目的主模块（机器），它的操作是项目的入口点。从这个模块开始，其他模块按 **IMPORTS** 关系分为不同层次，把整个项目分解为越来越小的部分

项目的 **IMPORTS** 图完全描述了一个 B 项目的可执行部分（所有最终模块）的结构。其中每个模块有一个代码模块，或者是抽象机开始的开发的最终结果，或者是基本机器的代码

抽象机之间的连接：IMPORTS 图

下面是一个系统的 **IMPORTS** 图，其中箭头描述 **IMPORTS** 连接。连接上可能标出重命名标记，如果有到一个模块的多个 **IMPORTS** 连接，就表示创建了该模块的多个实例。矩形中描述了各模块开发的精化过程



抽象机之间的连接

一个 B 模块实例只能通过 **IMPORTS** 导入到项目里一次。如果在一个项目里需要某个模块的多个实例，就必须在重复 **IMPORTS** 时予以重命名。同理，多次重命名需要使用不同的重命名前缀

另外，一个 B 项目里应该有一个且仅有一个开发模块作为根模块，项目里不出现到这个模块的 **IMPORTS** 连接将其导入其他模块

SEES 连接：

SEES 连接是系统的 **IMPORTS** 图里的横向连接，说明一个部件参考另一 B 机器实例，也就是说，以读的方式（不能写，也不能调用操作）访问该实例

所有要被 **SEES** 的模块都需要通过 **IMPORTS** 导入到项目里

如果某个 B 组件 **SEES** 某个模块实例，那么它的精化也要 **SEES** 该实例

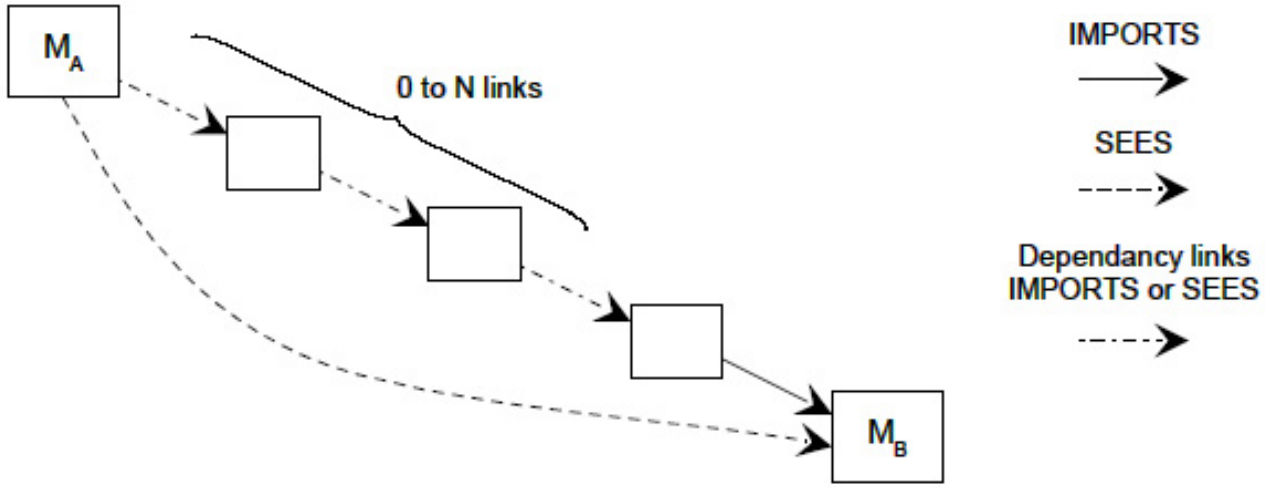
说一个 B 模块依赖于另一 B 模块，如果前一模块的实现 **SEES** 或者 **IMPORTS** 后一模块的一个实例。一个 B 项目的依赖图是在其导入图上添加所有的 **SEES** 连接。**SEES** 连接也标明被 **SEES** 机器的实例的重命名前缀

抽象机之间的连接和依赖

在 **SEES** 链上可能做多次重命名

如果一个 B 组件 **SEES** 某个模块实例 M_i，那么它就不能去 **SEES** 属于 M_i 的导入子图里的机器实例

下图是一个不合法的结构：



基于 IMPORTS 和 USES 的系统组织

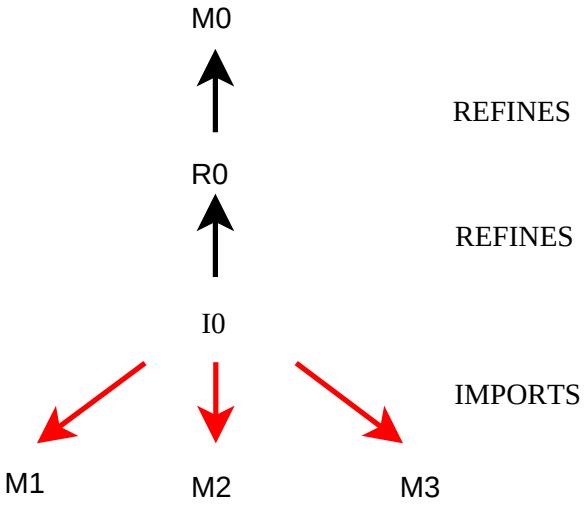
现在更详细地讨论一些有关 B 规范组织的问题。首先我们进一步考虑基于 **IMPORTS** 连接的抽象机组织和精化的关系，而后介绍如何通过 **USES** 连接在实现阶段中得到信息的共享

假定项目开发中首先做出了软件规范的主抽象机 M0，对这一抽象机做了两次精化，做出了实现 I0

实现规范 I0 通过 **IMPORTS** 子句导入了抽象机 M1, M2 和 M3，在这三个机器的基础上实现自己的功能

注意：M1, M2 和 M3 是三个抽象机（不是精化也不是实现），它们在抽象层面上描述了三个 B 模块

I0 通过 **IMPORTS** 导入抽象机 M1, M2 和 M3 规范，但实际用的是它们的最终实现模块的实例，它只能通过 M1, M2 和 M3 描述的接口去用那些实例



继续开发

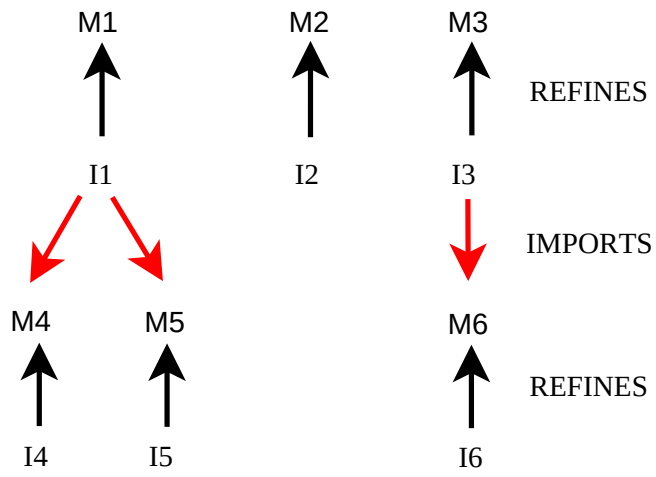
基于抽象规范模块 M1, M2 和 M3 还需要进一步开发，假设通过一步精化就做出了它们各自的实例 I1, I2 和 I3

其中 M1 的实现 I1 导入了抽象机规范 M4 和 M5

实现 I2 没有导入任何抽象机

实现 I3 导入抽象机 M6

M4, M5 和 M6 都是经过一步精化做出了实现（为简单起见）



注意：M1, M2, M3, M4, M5, M6 的开发都是独立完成的，例如 M6 的精化并不关注 M3 和 I3；而 I3 的开发只依赖于 M6，与 I6 无关

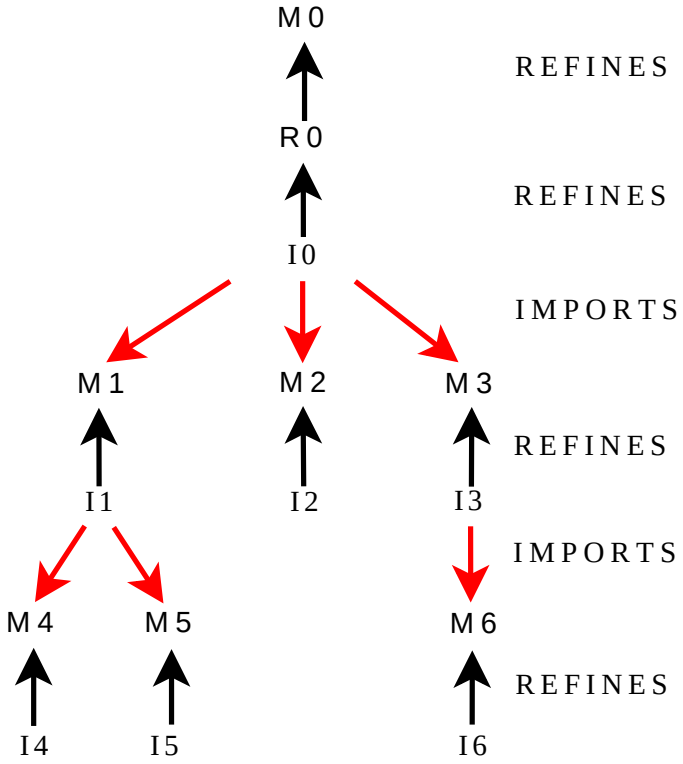
这个例子实际上说明了应该如何在 B 方法里开发复杂软件系统，如何将逐层分解的软件模块组织为一个 B 开发过程

完整开发过程

把这些开发步骤放在一起，就得到本软件整个开发过程的图示

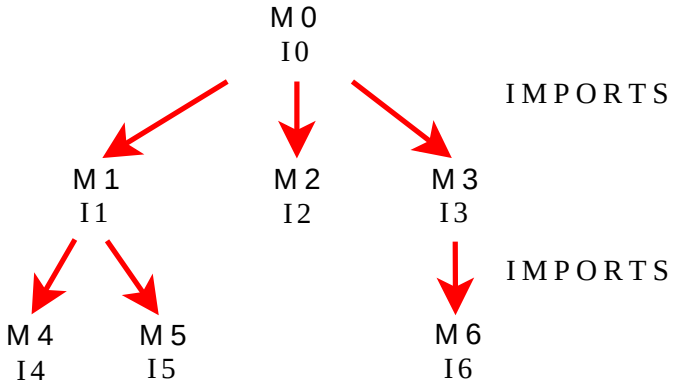
在这一开发过程中，存在 6 个独立开发的软件模块

每个模块有自己的规范和实现，有自己的开发过程（精化过程）



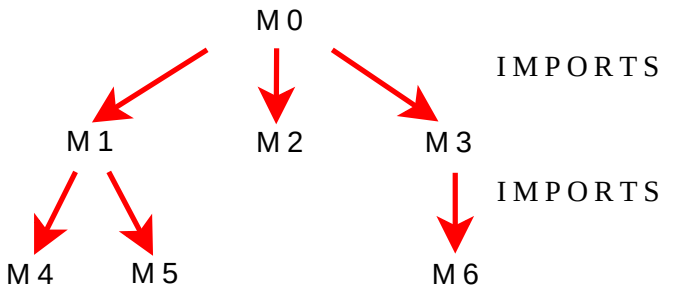
软件模块的树形结构

去掉开发结构图里的精化步骤（去掉表示 **REFINES** 的箭头），得到的是由一个个模块的规范机器/实现对作为结点的树形结构



去掉结构图里的所有实现模块，得到的是从规范的观点看到的软件系统的组织结构

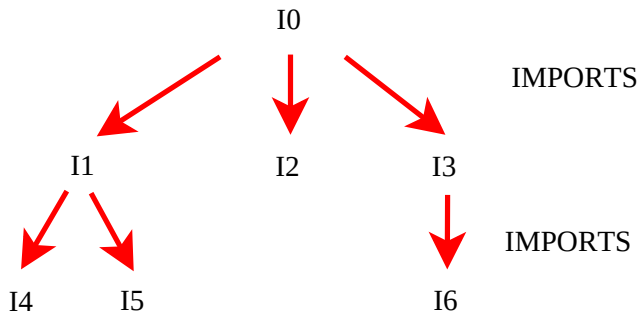
注意，**IMPORTS** 并不是抽象规范层的机制，但它带来的模块间连接关系可以反射到规范层面



软件模块的树形结构

去掉开发结构图里的所有规范模块，得到的是从实现观点看到的软件组织结构

注意，**IMPORTS** 是实现层面的机制，但它是连接到抽象规范部件（而不是连接到实现部件）。但连接关系可以反应到实现的层面



上述软件组织结构有非常好的性质：

- 软件里的模块分为一些层次，高层模块借助于低层模块的功能实现
- 每个模块的开发具有独立性，相互依赖达到最小
- 只要一个模块的规范（抽象机）不修改，其精化和最终实现都可以根据需要自由修改，这样的变动并不影响使用它的模块

然而，这种树形结构也有局限性，有时不能满足软件开发中的需要。关键问题在于树中只有不同层次之间的关系，不允许其他联系

B 模块之间的 USES 连接

原则上说，在树形结构里处于不同分支的结点之间是可以交换信息的，但这种交换只能通过它们的公共祖先结点，以及到这种祖先结点的路径上的结点上逐级传递。这种方式可能太麻烦，也不够有效

建立不同分支里结点之间的直接联系，相当于建立一些“短路”连接关系。在软件开发的场景中，要考虑的关键问题是这种短路关系会不会破坏系统的完整性，会不会破坏已证明的性质

可以证明，如果建立模块之间的短路连接，但只允许一个模块通过这种短路查询另一模块中的封装数据，这种直接联系就不会带来危险

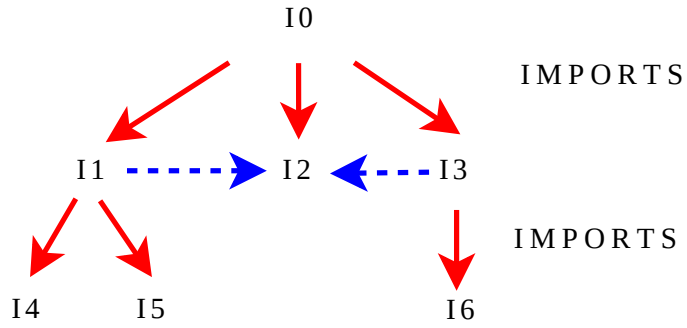
多个模块查询一个模块，共享其功能的一种特殊情况是被查询模块里根本没有变量。这方面的一个典型实例是实现通用数学函数的库模块

实际中可能更多的是更一般的情况：被共享访问的模块里有数据状态。在这种情况下，只要限制可能的访问方式，也不会带来问题

在 B 方法里，可以用 **SEES** 机制建立模块之间的短路连接，使若干个模块可以同时查询参考一个模块里的信息

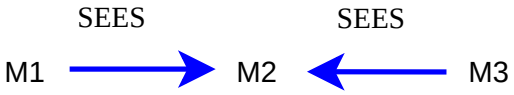
B 模块之间的 USES 连接

右图是前面开发的系统结构的一种变形情况，其中实现模块 I1 和 I3 都要查询 I2 里的信息，它们都建立到 I2 的 **SEES** 连接



问题在于需要把这种信息参考关系反映到这些实现的上层开发阶段，也就是说，需要在开发前期确定将来的实现中应该有哪些信息联系

反映到这个具体实例里，就是让 M1 和 M3 也参考 M2，建立它们之间的连接



通过 **SEES** 建立了连接，对于查看其他机器的机器的行为有明确的约束：在 M1 和 M3 的操作里可以引用 M2 的变量，但只允许已只读方式引用

这一规定符合实现层面的情况，在实现层的 **SEES** 关联也只允许查询

B 模块之间的 USES 连接

SEES 关系还有另外的约束：在 M1 和 M3 的不变式里不能引用 M2 的变量，原因是我们希望被 **SEES** 的模块独立于查看它的模块

例如，在开发 M2 时，显然不应该考虑它将来可能被哪些机器参考。如果在 M1 和 M3 的不变式里出现 M2 的变量，实际上就可能对这些变量增加新的约束，从而要求对 M2 重新做一些证明

应注意，上面讨论说明，模块之间引进 **SEES** 连接的基本想法是做一种优化，使原来可以做的事情能以一种更方便的方式描述

对目前这个例子，原先 M1 和 M3 都可能在操作里参考 M2 里的变量的情况。实现这种功能的方式是在 I0 里调用操作时，可以把 M2 的相关信息传进去。这种做法不会导致几个机器之间的状态依赖（没有不变式造成的连接）

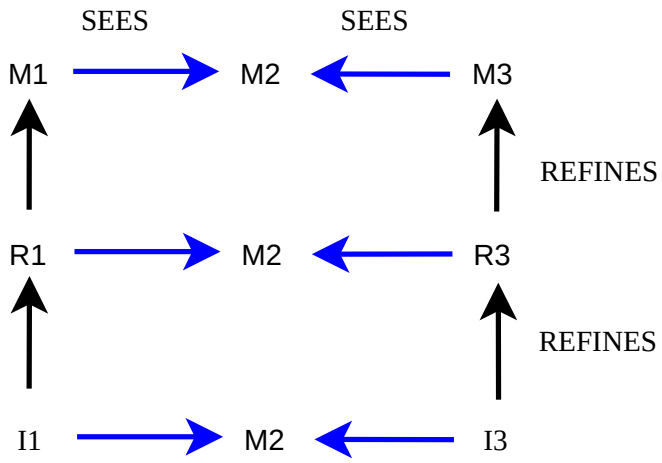
再强调一下：因为只是作为优化，因此也不应该引进新的功能和要求，也不应该引进对 M2 的新要求。因此这里不允许在 M1 和 M3 的不变式里出现 M2 的变量

B 模块之间的 USES 连接

在 M1 和 M3 里查看 M2 后，还需要对它们继续开发

在从 M1 和 M3 继续开发产生出来的（精化或实现）机器里，还要继续查看 M2

但还有一点问题值得注意：在实现模块 I1 和 I3 里最终查看的是 M2 的实现模块 I2。抽象机 M2 里的变量在 I2 里未必存在



显然 I1 和 I3 不可能去访问 M2 的实现模块 I2 里不存在的变量

实际上，对 M2 里的变量的访问只能通过 M2 提供的可用查询操作实现。这种技术的意义十分清楚，也是实际程序设计中经常使用的

最终情况是：实现模块 I1 和 I3 需要参考实现模块 I2 的状态，而这种访问是通过 M2 定义的 "变量查询" 操作实现的

SEES 连接

根据上面讨论，可总结出在 B 规范里使用 **SEES** 的情况：

- **SEES** 子句可以写在抽象机、精化或者实现里
- 被 **SEES** 的对象只能是抽象机，一个抽象机可以被多个 B 部件 **SEES**
- 每个被 **SEES** 的抽象机都要在整个开发里的某个位置被导入（**IMPORTS**，而且只能被 **IMPORTS** 一次）
- 在 **SEES** 一个抽象机时不需要考虑参数问题，因为该机器的参数会在它被（做唯一的一次） **IMPORTS** 时实例化

一个被 **SEES** 的抽象机对于查看它的不同 B 部件（抽象机、精化、实现）的可见性见《B Book》第 12.2.3 节和附录 D。应注意其中一些情况：

- 被 **SEES** 机器的集合和常量可以在查看它的部件的各部分使用
- 被 **SEES** 机器的变量只能在操作里读，如果本部件是实现，被 **SEES** 机器的抽象常量和抽象变量只能用在循环不变式里（与实际操作的实现无关）
- 在精化和实现的操作里，可以调用抽象机的查询操作

SEES 连接

SEES 连接没有传递性

- 如果抽象机或精化 M 查看机器 N，那么 M 的精化 M' 并不自动查看 N，而是要求在 M' 里明确写出 **SEES** 要求查看 N
- **SEES** 连接可以在任何开发步骤建立，写抽象机规范时可以建立 **SEES** 连接，在精化或实现时完全可以去建立新的 **SEES** 连接

在 **SEES** 一个抽象机时也可以做重命名，但

- 这种重命名只是为了在本部件里使用和描述方便
- 这种重命名没有传递性，M 重命名 **SEES** 的机器 N，那么 M 的精化 M' 不必在 **SEES** 时重命名 N，在 M' 里 N 也不自动被重命名
- 另外，如果另一机器 M1 也 **SEES** 抽象机 N，它可以自由决定是否重命名 N。如果重命名，也可以不用与 M 同样的重命名

总而言之，用 **SEES** 查看一个抽象机的信息，最终要求该抽象机的实现被通过 **IMPORTS** 包含在同一个项目里，真正在运行中参考那里的信息

多重精化和实现

这里介绍的“多重精化”只是在开发软件中可能有用的一种技术，其中的想法就是通过一个精化去同时精化几个抽象机

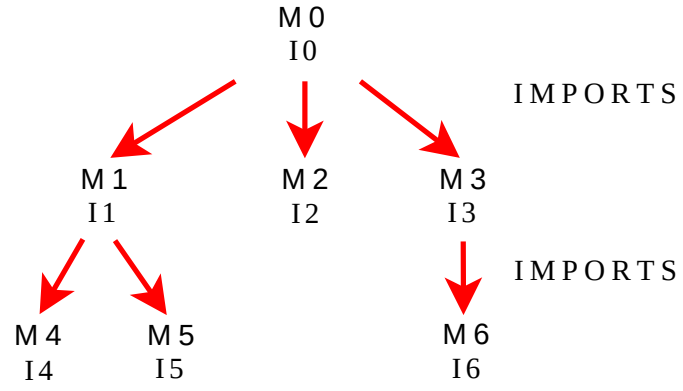
多重精化只是一种可以用在开发中的技术，可能用于优化系统的结构

假设开发中出现了右图所示的结构（前面的例子）

这里的几个机器都已做出了实现

但可能有时可能遇到一些情况，其中某些抽象机（或精化）的实现很容易做成一个机器

在这种情况下，就可以考虑用一个实现机器作为多个部件的精化。这就是这里提出的多重精化

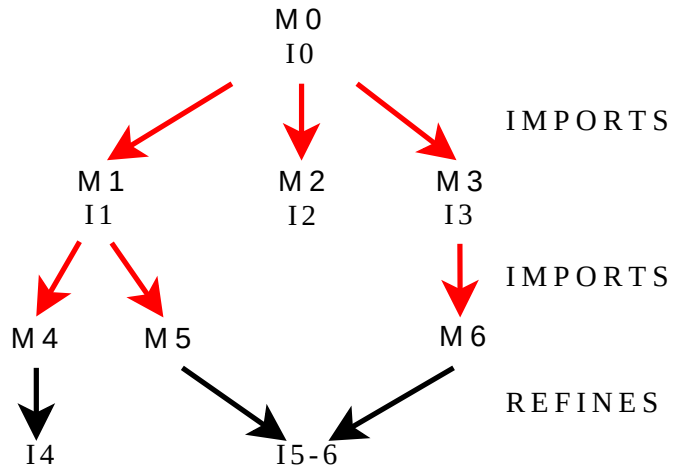


假设现在很容易写一个实现来精化抽象机 M5 和 M6，这种情况下，只用一个机器就简化了系统的结构

多重精化和实现

右图中用一个实现机器 I5-6 作为两个抽象机的实现

多重精化只是一种描述优化，可以减少规范的代码，并不影响系统执行时的结构



Atelier B 不支持多重精化。Atelier B 手册的 Introduction 里说

"The B Language Reference Manual describes the B language supported by Atelier B release 3.6. This language is based on the language presented in The B-Book, however some progresses such as trees, recursivity or multiple refinements are not currently supported by B language."

精化和实现的实例

看一个简单实例。第 8 次课讲过一个零售系统实例，其中包括几个抽象机：

MACHINE $Goods$
SETS $GOODS$
END

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;**
$pr \longleftarrow pricequery(item) =$
PRE $item \in GOODS$ **THEN** $pr := price(item)$ **END**
END

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

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 $purchases := purchases \cup \{item\} \parallel pr := price(item)$ **END**
END

Customer 的精化和实现

抽象机 $Customer$ 维护了一个已选货品的集合 $purchases$，其精化和实现也都需要维持这样的集合。某种元素的集合是软件中常用的数据结构，程序库可已有一个通用的表示集合的抽象机 $\boldsymbol{Set}$:

MACHINE $\boldsymbol{Set}(ELEM)$
VARIABLES set
INVARIANT $set \subseteq ELEM$
INITIALISATION $set := \varnothing$
OPERATIONS
$\boldsymbol{add}(elm) =$
PRE $elm \in ELEM$ **THEN** $set := set \cup \{elm\}$ **END;**
$res \longleftarrow \boldsymbol{member}(elm) =$
PRE $elm \in ELEM$
THEN $res :=$ bool$(elm \in set)$ **END;**
... ...
END

我们考虑基于这一抽象机实现 $Customer$

Customer 的精化和实现

通过 **IMPORTS** 抽象机 $\boldsymbol{Set}$，利用它的功能保存所选货品，可以做出下面的实现。当然，这一实现仍然需要参看抽象机 *Goods* 和 *Price*

IMPLEMENTATION *Customer_Imp*
REFINES *Customer*
SEES *Goods*, *Price*
IMPORTS $\boldsymbol{Set}(GOODS)$
VALUES $limit = \boldsymbol{\lambda}\, g \cdot (g \in GOODS \mid \mathbf{1000})$
INVARIANT $set = purchases$
OPERATIONS
  $pr \longleftarrow \boldsymbol{buy}(item) =$
  BEGIN
    $pr \longleftarrow pricequery(item)$;
    IF $pr \leq limit(item)$ **THEN** $add(item)$ **END**
  END
END

这里用 $\boldsymbol{\lambda}$ 表达式是说明情况，实际上需要提供各种货品的限价，给出一个实际的函数作为 *limit* 的值，这个值应该能够用数组表示

先进先出队列：规范

考虑下面实例，抽象机 Fifo 是一个先进先出队列的规范：

```
MACHINE Fifo(ELEM, capacity)
CONSTRAINTS capacity : NAT1
VARIABLES contents
INVARIANT contents : seq(ELEM) & size(contents) <= capacity
INITIALISATION contents := []
OPERATIONS
    input(elm) =
      PRE elm : ELEM & size(contents) < capacity
      THEN contents := contents <- elm END;

    elm <-- output =
      PRE contents /= []
      THEN elm, contents := first(contents), tail(contents)
      END
END
```

抽象机管理一个 ELEM 元素的队列，队列尚不满时可以加入元素

先进先出队列：实现

队列的一种可能实现方式（代价可能太大，只是作为例子）是用一个 Archive（归档）抽象机作为实现的基础，所有 input 的数据项都进入永久保存的归档序列。该抽象机的规范：

```
MACHINE
    Archive(ELEM)
VARIABLES read, entries
INVARIANT read : NAT & entries : seq(ELEM)
INITIALISATION read, entries := 0, []
OPERATIONS
    enter(elm) =
      PRE elm : ELEM THEN entries := entries <- elm END
    elm <-- lookup =
      PRE read : NAT1 &  read <= size(entries)
      THEN elm := entries(read) END;
    incread = BEGIN read := read + 1 END;
    resetread = BEGIN read := 0 END;
END
```


先进先出队列：实现

状态变量 read 表示关注的位置，也就是下次查询操作执行的位置。另两个操作完成重置 read 指示器和指示器移位

基于 Archive 很容易实现 Fifo:

```
IMPLEMENTATION Fifo_Imp(ELEM, capacity)
REFINES Fifo
IMPORTS Archive(ELEM)
INVARIANT entries \|/ read = contents
OPERATIONS
   input ( elm ) = enter(elm) ;
   elm <-- output =
     BEGIN
       incread;
       elm <-- lookup
     END
END
```

入队数据进入归档序列 entries，从 read 开始的归档序列与队列元素相同

健壮的先进先出队列

前面的队列机器具有脆弱性，如果在不合适的时候调用其输入（队列满时）或输出（队列空时）操作，抽象机 Fifo 的实现都会崩溃

下面考虑一个健壮的先进先出队列，其中的操作都检查当前状态，并报告操作的成功与否。先写出这一抽象机的规范，然后考虑它的实现

队列提供一个查询其元素个数的操作 numberQ，它总能成功返回结果

操作 add 和 remove 总能执行到结束，绝不会崩溃。而且它们将会报告完成操作的情况：用 ok 报告成功完成加入或删除元素的操作；用 failed 说明操作无法成功完成的情况（要求加入元素时队列已无空间，或要求删除元素时队列里无元素）

remove 操作成功完成时返回删除的元素。为保证它不能成功完成时也能返回一个元素，抽象机增加了一个默认值参数 elem0，作为这时的返回值

用户方对 remove 的正确使用方式中，应该先检查确认操作已成功完成，而后才能去考虑使用其返回值


```
MACHINE RobustFifo(ELEM, capacity, elem0) /* elem0 表示默认值 */
CONSTRAINTS capacity : NAT1 & capacity < 1000 & elem0 : ELEM
SETS REPORT = {ok, failed}
VARIABLES queue
INVARIANT queue : seq(ELEM) & (size(queue) <= capacity)
INITIALISATION queue := []
OPERATIONS
    rpt <-- add(elm) =
      PRE elm : ELEM
      THEN IF size(queue) < capacity
        THEN rpt := ok || queue := queue <- elm
        ELSE rpt := failed END
      END;
    rpt, elm <-- remove =
      IF size(queue) = 0
      THEN rpt := failed || elm := elem0
      ELSE rpt := ok || elm := first(queue) || queue := tail(queue)
      END
END
```

健壮的先进先出队列

考虑用如下方法实现这种健壮队列：

- 导入一个前面定义的 Fifo 机器，维护系统状态；用一个计数器机器 Counter 记录队列的大小
- 虽然 Fifo 及其实现都具有脆弱性，在其操作的前条件不满足的情况下会崩溃。包装在 RobustFifo 的实现的内部使用，可以保证这些前条件

实现中使用的 Counter 机器如下：

```
MACHINE Counter
VARIABLES counter
INVARIANT counter : NAT
INITIALISATION counter :: NAT
OPERATIONS
    num <-- number = num := counter;
    setzero = counter := 0;
    inc = counter := counter + 1;
    dec = PRE counter > 0 THEN counter := counter - 1 END
END
```


健壮的先进先出队列：实现

下面给出这样做出的实现。首先是系统前面的静态部分：

```
IMPLEMENTATION RobustFifo_Imp(ELEM, capacity, elem0)
REFINES RobustFifo
IMPORTS count.Counter, Fifo(ELEM, capacity)
INVARIANT count.counter = size(contents) &
    count.counter = size(queue) & contents = queue
INITIALISATION count.setzero
```

其中初始化调用了 Counter 机器的操作，用 PROMOTES 把 RobustFifo 机器的操作提升为本机器的接口操作

操作部分的实现见下页，其中调用了两个 **IMPORTS** 的机器的操作

这个具体例子的价值不大，但它展示了一种技术：通过 **IMPORTS** 连接，基于一个脆弱的抽象机实现一个强健的软件模块。这种技术很有用，用于保证可能出问题的操作总在其前条件成立的情况下被调用

```
OPERATIONS
    rpt <-- add ( elm ) =
      VAR num IN
        num <-- count.number;
        IF num < capacity
        THEN rpt := ok; input(elm); count.inc
        ELSE rpt := failed
        END
      END ;

    rpt , elm <-- remove =
      VAR num IN
        num <-- count.number;
        IF num > 0
        THEN rpt := ok; elm <-- output; count.dec
        ELSE rpt := failed; elm := elem0 END
      END
END
```


实现步骤中的数据精化

显然，上面实现并不令人满意，因为抽象机 `Fifo` 的实现依赖于一个代价极高的抽象机 `Archive`。下面考虑其他实现可能性

在实现的层面上同样可以考虑 “数据精化”，当然这时要考虑 “实现” 对于数据表示的限制。基本原理与做精化时一样：

- 设计新机器的实现层面的状态表示
- 通过 “连接不变式” 建立实现机器的状态与被它精化的机器状态的连接

不同点

- 实现机器的相关状态是在被它 **IMPORTS** 的机器里的变量中描述的，而不是直接在实现机器里的 **VARIABLES** 里描述的
- 实现机器中改变状态的操作都需要通过被 **IMPORTS** 机器的操作完成
- 实现机器的连接不变式描述被它精化的抽象机器的状态与被它 **IMPORTS** 的那些机器的状态之间的联系

实现中的数据精化

现在以 Fifo 机器为例讨论实现的数据精化问题

用一个永久性数据结构实现队列元素存储，显然并不必要，因为队列不会重复用已删除的元素，现在考虑改进

下面的考虑是用一个大小为 capacity 的有穷数组（大家都知道的技术，但需考虑如何在抽象层面上描述和证明），具体方式

- 加入队列的元素将存入数组
- 如果元素被删除，就忽略它
- 要解决的主要问题是：数组长度有限，当加入元素时达到数组末尾，下面元素怎样放置和找出

这里的想法是（常规技术）：

- 有效元素总从数组中某个位置开始存放
- 有效元素的个数已知，不能多于 capacity
- 如果存（取）达到数组末端，就转回数组开始继续存（取）

用数组实现有限队列的状态

假设存放元素的变量是 $array$，存放元素的开始位置由变量 pos 描述，当前数组元素个数由变量 $size$ 描述

不难写出这些变量与原机器里的变量 $contents$ 之间的关系，实际上也就是精化中的连接不变式：

$$contents = ((array \downarrow (pos - 1)) \frown (array \uparrow (pos - 1))) \uparrow size$$

其中

- $array \downarrow (pos - 1)$ 是 $array$ 丢掉前 $pos - 1$ 个元素
- $array \downarrow (pos - 1)$ 是取得 $array$ 的前 $pos - 1$ 个元素
- 两段连接后取出前 $size$ 个元素，就是队列元素

下面要做的 Fifo 的实现就是基于这样的考虑，其中 **IMPORTS** 了几个机器，最终做出 Fifo 的一个高效的实现

注意，用这一实现取代前面的低效实现，同样可以包装为强健的模块

几个基础抽象机

这个新实现依赖于几个抽象机

首先是一个数组，其中保存队列元素：

```
MACHINE Varray( capacity, VALUE )
CONSTRAINTS capacity : NAT1
VARIABLES array
INVARIANT array : 1..capacity --> VALUE
INITIALISATION array :: 1..capacity --> VALUE
OPERATIONS
    set(ind, val) =
      PRE ind : 1..capacity & val : VALUE
      THEN array(ind) := val END;
    val <-- get(ind) =
      PRE ind : 1..capacity
      THEN val := array(ind) END
END
```


几个基础抽象机

需要两个计数器抽象机，一个记录队列元素个数

```
MACHINE SizeCounter (maximum)
CONSTRAINTS maximum : NAT1
VARIABLES sze
INVARIANT sze : NAT & sze <= maximum
INITIALISATION sze:= 0
OPERATIONS
    sizeinc = PRE sze < maximum THEN sze := sze + 1 END;
    sizedec = PRE sze > 0 THEN sze := sze - 1 END;
    ss <-- sizeget = ss := sze
END
```

元素个数统计就是加一减一和取元素个数

几个基础抽象机

另一个记录队列元素在数组里的开始位置

```
MACHINE PosCounter (maximum)
CONSTRAINTS maximum : NAT1
VARIABLES pos
INVARIANT pos : NAT1 & pos <= maximum
INITIALISATION pos:= 1
OPERATIONS
    posinc = pos := (pos mod maximum) + 1;
    pp <-- posget = pp := pos
END
```

这里的关键就是位置移到最后需要转回数组最前面

Fifo 的新实现

新实现抽象机的静态部分：

```
IMPLEMENTATION Fifo_Imp2(ELEM, capacity)
REFINES Fifo
IMPORTS Varray(capacity, ELEM), SizeCounter(capacity),
        PosCounter(capacity)
INVARIANT
    sze = size(contents) &
    ((array \|/ (pos - 1)) ^ (array /|\ (pos - 1))) /|\ sze
        = contents
```

这里导入的三个基础抽象机，需要使用它们的实现

不变式连接起被精化的 `Fifo` 和被 **IMPORTS** 的两个抽象机的状态，后一复杂关系前面已经讨论

Fifo 的新实现

操作通过调用被 **IMPORTS** 的机器的操作实现

```
OPERATIONS
    input(elm) =
      VAR sz, ps, pp IN
        sz <-- sizeget; ps <-- posget;
        pp := (sz + ps - 1) mod (capacity - 1);
        set(pp, elm);
        sizeinc
      END;
    elm <-- output =
      VAR pp IN
        pp <-- posget;
        elm <-- get(pp);
        posinc;
        sizedec
      END
END
```