工业开发中的形式化方法：成就、问题和未来

Jean-Raymond Abrial
Swiss Federal Institute of Technology, Zurich
（国际软件工程大会特邀报告，2006 年元月，上海）

摘要：本文简单介绍用 B 形式化方法做的两个实际项目，它们说明了复杂系统里的重要部分可以如何用这样一种“**构造即正确**”的方式开发出来。随后分析了关于将这种方法用于工业开发的一些正面和反面意见，并分析了其中的困难。

1，引言

本文的意图是报告在工业中应用形式化方法的经验。虽然下面主要给出了我知道的几个应用 B 方法 [1] 的案例，但从这些案例得到一些结果也适合其他形式化方法的应用。我将先简要介绍相关案例，而后分析其中的所学所悟。在做这些之前，首先简要概述一下什么是这里所说的“形式化方法”。

2，形式化方法

本文用术语“形式化方法”时取其最精确、或许最窄的意义。应该说明，今天人们赋予这一术语许多不同理解。为使本文简短，这里只给出与使用 B 形式化方法有关的**最重要概念**。

2.1 形式化开发

本文案例中使用的 B 形式化方法是一种途径，工程师可以按这种途径把**软件需求文档**转换到某种**可执行代码**。这一说法似乎与普通程序员做的事情没什么差别。使这一途径与常规方法大相径庭的地方在于应用它的工程师**再不需要去编任何程序**。事实上，他们不再使用某种经典程序设计语言，而是工作在更抽象的层次上，**那里根本就没有执行的概念**。

由于这种情况，在这里不能通过执行代码的方式做测试和调试。显然我们不可能保证工程师不再犯任何错误，因此需要通过其他途径验证他们做出的东西。在下面 2.2 和 2.4 节，我们将看到用什么东西取代可执行的测试。

这一开发途径可以划分为三个独立的相继进行的阶段：

(1) 阶段 1，从**软件需求文档**中**逐步**抽取出问题的细节。这一工作要通过逐步求精最终构造出一个**抽象模型**。**抽象模型**的抽取和逐步构造需要大量人工参与。

(2) 阶段 2，软件需求文档已经不再使用了，现在的出发点就是得到的**抽象模型**，它被（仍然是逐步地）转换到一个**具体模型**。这一阶段也需要一些人工参与，但是如我们将在 4.5 节里看到的，这种参与不像前一阶段那么广泛。

(3) 阶段 3，将**具体模型**自动翻译为**可执行代码**。这一阶段完全无须人工参与。

这种途径的整体效果是：得到的**可执行代码**是**构造即正确的**（相对于**软件需求规范**而言）。下面几小节将简要说明这些阶段的一些情况。

2.2 抽象模型

本小节将简要描述抽象模型的结构，还要说明工程师**在构造它的过程中**进行验证的方法。

抽象模型里首先包含一些**数据**，数据通过**不变式**引入。这些大大扩充了程序设计语言里用的**数据声明**的简单概念。不变式是用**一阶逻辑**或**集合论**结构（集合、关系、函数等）写出的谓词。正如其名字说明的，一个不变式也就是数据的一种性质，在系统的动态活动中总保持不变（虽然其间数据本身可能变化）。注意，不变式也可能描述了某种全局性质，这种性质是不可能在程序语言里表达的。还请注意，变量及其不变式并不是一下子引进的，而是随着**抽象模型**的构造工作的进展，通过逐步叠加的方式将它们结合进来。

模型的动态特性用简单**转换**描述。在许多情况下这种转换为**非确定性的**，一般用经典的集合论结构定义。再说一次，与数据一样，随着抽象模型的构造工作的进展，也要通过逐步叠加的方式扩充这种转换。

2.3 证明

工程师通过**数学证明**的方式，逐步验证他们正在构造的**抽象模型**。注意，在这里要证明的语句不是工程师自己定义的，而是由一个称为**证明义务生成器**的工具自动生成。为了生成证明义务，该工具将不断分析抽象模型的各个精化形式。请注意，由工具确定应该证明什么（它可能生成数以千计的证明义务）是非常重要的。如果让工程师自己做，他们很可能在生成这种待证明语句时犯错误，从而把复杂性从一个地方搬到了另一个地方。

要做的证明主要关注两方面情况：

(1) **抽象模型**里的转换是否保持了不变式的成立。这方面的证明称为**不变式保持性证明**。

(2) **抽象模型**的每个更准确的版本是否破坏了前一版本中已经证明的性质，这称为**精化正确性证明**。在开发抽象模型的过程中，第二类关注所导致的证明义务通常比不变式保持性的证明义务简单一些。究其原因，在这一步做的事情也就是**叠加**（参看前一节），每一步简单加入一些新变量，也简单地加入一些转换，不破坏前面步骤中引进的东西。

这些证明中的大多数都能由一个称为**证明器**的工具**自动**完成。其中比例很小的一部分（将在 3.3 节说明）证明需要通过交互方式完成，工程师需要给证明器提供一些提示。

2.4 具体模型

具体模型的结构与**抽象模型**完全一样，也由数据和转换构成，也是通过逐步求精的方式逐渐构造出来的。

然而，虽然在具体模型的最后版本里包含的数据仍是**真正的集合论对象**，但其中也有些数据与计算机化的对象（如标量、指针、数组、文件等）一一对应。和**抽象模型**的情况一样，**具体模型**的最后版本还是用转换定义系统的动态行为。但此时的转换已经都是确定性的，而且与经典程序语言里可以找到的东西类似，有顺序复合、条件语句、循环、过程调用等。

对于**具体模型**，证明义务生成器生成的待证明语句与验证抽象模型时生成的东西类似。但另一方面的情况有些不同，现在更复杂的是关于精化的语句，因为本质上看现在正在做两件事：

(1) 从抽象模型里的抽象集合论结构（集合、关系、函数等）到计算机化的集合论对象的**数据精化**。

(2) 将基于集合论的非确定性转换，转换到经典的确定性的程序设计语句。

2.5 可执行代码

除了通过数学证明的方式验证之外，还要用另一个工具检查**具体模型**的最后版本，看它是否只包含能机械地翻译到可执行代码的数据和转换。一旦通过了这个检查，我们就可以用连续的两个步骤完成相关翻译工作了（当然也可以一步完成）：

(1) **具体模型**被自动地翻译为所需要的形式，采用某种经典的程序设计语言。在下面将要讨论的案例中用的语言是 ADA。

(2) 用一个普通编译器，将由具体模型生成的程序语言版翻译成可执行代码。

注意：这些自动化步骤是本方法的**软肋**，因为无法保证这两个翻译器（特别是第二个）完成的是正确翻译。我们将在下面 4.6 节看到可能如何（部分地）处理这一问题。还请注意，本方法的另一软肋是**软件需求文档**，我们将在 4.1 和 4.10 节回到这一重要问题。

3，案例研究

本节给出的两个案例跨越了 8 年时间，第一个系统从 1998 年 10 月开始一直工作，而第二个将于 2006 年 9 月投入使用。任何人都有机会去体验这两个系统，因为第一个案例就是巴黎城市铁路 14 号线的全自动无人驾驶地铁服务系统，而第二个将提供 Roissy 机场各候机楼之间的全自动无人驾驶穿梭车服务。

请注意，在这两个系统里，并不是所有程序都是采用形式化方法开发的。预先进行的系统研究确定了哪些部分应该采用形式化的开发和证明，它们对应于系统中安全攸关的部分，大约占整个程序的三分之一。

线路全长	**8.5 公里**
停站数	**8**
列车间距时间	**115 秒**
速度	**40 公里/小时**
列车数	**17**
每天旅客数	**350000**

表 1：巴黎地铁 14 号线的参数

线路全长	**3.3 公里**
停站数	**5**
列车间距时间	**105 秒**
速度	**26 公里/小时**
列车数	**14**
每小时旅客数	**2000**

表 2：Roissy 机场穿梭车的参数

3.1 案例 1：巴黎地铁 14 号线

表 1 给出了这条地铁线路的基本情况 [11]。该系统中通过形式化开发的部分在一篇很好的论文 [5] 中描述得很清楚，高度推荐有兴趣的读者去读一读。下面的粗略描述就取自这篇文章。

由于这一繁忙的地铁线路是完全自动化的，安全攸关的部分关注列车的行驶和停止，列车和站台门的开关。整个程序划分为三类不同的子系统：轨边设备（沿轨道安装了若干这样的设备），车载设备（每列列车一套设备）和线路设备（一套设备）。这些子系统是高度互连的。在每个子系统里，用形式化方法开发的安全部分都具有如下特征：它们都是顺序的而且是周期性的程序（350ms），每个程序由一个不可中断的作业构成。

3.2 案例研究 2：Roissy 机场穿梭车

表 2 给出了这一穿梭车线路的主要参数，数据来自论文 [11]。这一系统也有一篇描述清晰的论文 [4]，同样推荐感兴趣的读者阅读。

Roissy 机场穿梭车系统的设计源自芝加哥的 O'Hare 机场的轻型穿梭车系统，两个系统之间

的不同在于前者有放置在轨道旁的计算机化的重要部分，称为**轨边控制单元**。在轨道旁配置了一些这样的单元，通过一个以太网络将它们连在一起。

轨边控制单元驱动列车，给列车发送必须执行的预定义的速度程序。轨边有一些传感器检查实际运行情况，它们连在**轨边控制单元**上，使这些单元可以根据实际情况给列车发信号。

项目	第 1 个案例	第 2 个案例
ADA 代码行	86000	158000
证明个数	27800	43610
交互式证明的百分比	8.1	3.3
交互式证明所用人月	7.1	4.6

表 3：案例研究的比较

3.3 两个案例研究的比较

表 3 给出了一些数据，使我们可以比较这两个案例。最重要的数据是第一项和最后一项。第一项是两个程序的代码行数，而最后一项是执行相应的交互式证明所花费的时间。

ADA 代码行数代表了用形式化方法开发的软件系统部分的规模。工程师**不修改**这些代码行。

交互式证明的时间按每天完成 15 个交互式证明，每月 21 天计算。从表中可以看到，从第一个到第二个项目，进步是很明显的：大致上自动生成了约两倍的代码行，证明时间减少一半。厂商说第二个案例的重要时间节省在具体模型的构造。我们将在 4.5 节解释这些差异来自何处。注意，两个案例都**没做单元测试**，我们将在 4.3 节看到仍然执行了什么测试。

两个案例间有一个重要不同点：第一个系统的软件需求分析是专门针对它自身做的；而第二个系统的需求由一份现有的需求文档导出（芝加哥 O'Hare 机场穿梭车）。在第二个项目中，这一文档经过修改和扩充，以适应 Roissy 机场穿梭车的新需求和功能要求。这一做法产生了一些问题，到后来开发抽象模型的时候才发现。

3.4 其他类似的案例

类似的列车控制系统包括正在按照同样方式开发的纽约城地铁，巴塞罗那地铁，布拉格地铁，巴黎地铁的 1 号线。

4. 所学所悟

在这一节，我想分析将这一方法应用于工业开发的几个要点。

4.1 需求文档的重要性

作为这两个案例（以及其他项目）的出发点是软件需求文档。显然，这种文档的质量和完全性是极端重要的。

这种文档是用半形式化的混合方式写出的，但作为整体，它主要用自然语言撰写。一般而言，这种文档可能有两类相对弱点：

(1) 其质量通常较差，或者是太短，或者是相反，太罗嗦。遇到这两种情况，要提取**精确的需求**都很难。应该清楚，使用软件形式化方法并不能更正软件需求文档里的问题。换句话说，如果需求文档里存在一个错误或一项疏漏，该错误或疏漏也很可能出现在抽象模型里。注意，无论如何，对抽象模型做各种证明可以揭示出软件需求文档里一些不一致性。

(2) 抽象模型开发者对需求文档的理解里可能出现错误，使抽象模型没有反映软件需求文档作者的意图。这一问题可以部分地通过组织一个**复审小组**（独立于开发小组）的方式处理，该小组的工作就是检查抽象模型，以确认它正确反映了软件需求文档的内容。这一队伍在两个案例中都起了重大的作用。

无论如何，这两点都如此重要，我们还要在下面（4.10 节）建议一些方式来改善目前的情况。

4.2 抽象模型的难点

使用形式化方法，最困难的部分肯定是在构造抽象模型时遇到的。一般而言，工程师（特别是软件工程师）并没有经过有关**建模**训练方面的良好教育，如前一节指出的，他们在**软件需求文档的撰写**方面也没有良好的训练。

常见情况是，软件工程师在第一次遇到这种方法时掌握不好抽象模型的**逐步构造过程**。人们倾向于或者一步也无法前进，或者只能做很少几步。这样就大大加重了证明的负担（我们将在 4.3 节回到证明负担的问题），并给根据软件需求文档检验抽象模型的工作带来很大困难。事实上，软件工程师对精化概念（逐步求精，与其技术方面无关）的**理解也很差**。请注意，其他更成熟的工程领域里的情况并不是这样，在那里建模和精化都是很自然的。

不能很好地做精化有一个原因，那就是很不容易决定构造步骤的组织方式：要把软件需求文档里的各组成部分结合到抽象模型里，应该采用的最佳顺序是什么？是从功能性需求开始，而后考虑安全性需求，然后再考虑失误处理的需求？还是采用其他顺序？目前对这一问题还没有明确的答案。经验还扮演着重要的角色。

虽然有这些困难，由于形式化的抽象模型将变成未来设计中的唯一模型，这一工作的重大优势之一就是迫使工程师去深入理解软件需求文档。在前面讨论的两个案例中，在此期间都出现了开发者和软件需求文档的作者之间大量的各种形式的交流（email，会议，审查）。通过这种交流，大量的澄清和扩充被加入需求文档里。

作为这种关系中很重要的一个方面，软件需求文档变成了有关需求的可追溯性的起点。这种可追溯性首先可以在模型里看到，而后进入未来的设计，最后进入可执行代码。

4.3 测试和证明的比较

在这两个案例里，单元测试和集成测试（对于安全攸关的系统，这些都是很繁重的工作，因而代价高昂）都被完全抛弃了。人们认为**证明**是比测试**好得多的验证过程**。请注意，去掉单元测试是巴黎地铁的管理部门 RATP 给开发商的建议。

当然，这并不意味着所有测试都不做了。现在可以把测试工作更好地集中到比软件模块及其集成更重要的方面，即集中到软件需求文档本身。前面我们已经认为需求文档是一个薄弱点。这里的想法是建立一个检验小组（在 4.1 节提到过），定义一些能检查软件需求文档里的错误和缺失的**全局性的功能测试**（以及灾难场景）。

由于证明已经部分地代替了测试，对两者做一些对比，看看每一种方法能给开发者提供些什么东西，也是很有意义的事情。

4.3.1 测试

良好准备的程序测试包括四个阶段，列举如下：

(1) 定义手头要测试的程序的某个**精确属性**。

(2) 详尽描述测试的**预期结果**，需要在测试**之前**做好。注意，有时很难得到这种详尽描述，因为它必须有某种根据，而且显然这一根据应该独立于被测试的程序。然而在一些时候，作为测试依据的详尽描述的根据就出自测试者的头脑，是测试者通过查看程序究竟做了些什么而得到的！

(3) 测试本身是由被测试的程序运行的。

(4) **比较**测试结果与测试前预期的详尽描述的结果。测试结果与预期结果相同并不意味着这个程序是正确的，而仅仅意味着它通过了测试。如果比较的结果是否定的，我们可以认为是这个程序不正确，也可以认为是所预期的结果不正确。

4.3.2 证明

对一个模型做**证明**，情况与刚才讨论的测试完全不同。事实上，上面测试步骤 (1) （选择一个要测试的性质）现在不存在了。原因很简单：未来程序的抽象模型不是别的什么东西，它实际上就是合格的程序必须满足的一组属性。换句话说，属性**就是模型的一个组成部分**，而不是后来为了做程序测试而顺便选出的东西。这就是程序设计和抽象建模之间最重要的不同点。在程序设计中，你设法弄清计算机应该做什么。而建模问题与指挥计算机无关，其矛头直指未来程序的静态和动态属性，而且允许工程师对这些属性进行推理检查。

一个证明的结果可能遇到下面几种情况：

(1) 证明器成功完成证明（自动地或交互地）。

(2) 证明器成功证明了假定应该证明的东西的否定。

(3) 证明器失败了，但工程师很自信地认为要证明的东西应该是对的。

(4) 证明器失败了，而且工程师觉得要它去证明被证明命题的否定也会失败。

情况 (1) 对应于我们的目标，对一个完整模型的证明阶段最终需要所有的证明。情况 (2) 很有趣，因为它指出了模型中需要修改的东西。对于情况 (3)，大部分情况下不应该把证明器的失败归咎于证明器本身：这一情况说明模型太复杂，应该去重新组织。所以说这种指示也是极其有价值的。在证明器遇到困难的时候，很可能时因为模型的结构太差。这一结果使我们很吃惊，这种信息现在变成了系统质量的一种检验器。作为这种认识的结果，如果证明器不能自动完成 90%以上的证明，那么工程师就需要去重新组织自己的模型。情况 (4) 意味着模型本身不够丰富，因此需要去扩充它。

所有这些情况都是很有趣的，各种结果都很好地集成到开发过程本身。总结一下：建模和证明本身并不是目标，而是对我们希望构造的系统提出各种问题的绝好基础。

4.4 方法文档

在两个案例中，工程师都用了一本称为“B 开发手册”的文档。这一文档起的作用与 Gamma 等人的“设计模式”一书 [9] 类似，但用在完全不同的上下文里。该文档就像是一本“方法仓库”，帮助工程师去做自己的数学模型。人们发现这样的文档是非常重要的，它使工程师能以一种非常系统化的方式工作。我们将在下一节里看到，这一“B 开发手册”可能被部分地转换为一种特殊工具。

除了其他东西外，“B 开发手册”里还给出了一些模拟各种不同类型的自动机时（软件需求文档里常有这种东西）应遵循的规则。它还说明了如何建立用 B 开发部分与没有采用形式

化方法开发的部分之间的接口。最后，它提出了许多用于做数据精化的技术，说明了怎样写出证明器更容易自动证明的形式化模型。

4.5 构造具体模型

这里给出的两个案例之间最主要的差别在具体模型的构造方面。对于第二个案例，具体模型几乎完全是用一个名为“EdithB”的精化工具完成的。该工具能做一些半自动的数据精化，在几篇论文中有对它的细致介绍 [7,8]。

EdithB 工具解析抽象模型的子模型，检查其中哪里定义和使用了抽象集合论的数据。它有一个预定义的数据精化模式的数据库，设法系统化地用这些模式做精化。如果 EdithB 失败了，工程师还可以提供自己的精化模式，并把它加入 EdithB 的数据库中。

为了保证 EdithB 不执行错误的精化（EdithB 本身可能包含程序错误），它做出的结果全都要经过证明，就像工程师做的一样。这样就避免了对 EdithB 本身做非常复杂的证明。

使用 EdithB 的效果是很惊人的，大约 2/3 的 B 模型可以自动生成。把这样做出的模型与完全手工做的模型做了比较，结果是最后的代码虽然效率稍低一些，但还是很令人满意的。

4.6 关键代码处理器

正如 2.5 节指出的，从最后的具体模型到 ADA 的精化，以及随后从 ADA 到目标代码的翻译是这一开发过程的薄弱点。在这里说明一下如何把这部分做得更安全些。

首先，我们有两个独立开发的从 B 到 ADA 的翻译器。在对具体模型的最后精化步骤里同时使用这两个翻译器，论文 [4] 和 [8] 里解释了比较两个结果的方式。

在这里给出的案例里，除了用 B 形式化方法，还让自动生成的软件在一个扩展处理器（**关键代码处理器**，见 [6]）上执行。粗略地说，每项数据都编码为两部分：一个正常部分包含数据值；还有一个冗余部分。执行时关键代码处理器检查两个部分之间所有可能检查的不一致情况。如果出现不一致，就把整个系统置入一种安全状态（也就是说，让列车停下来）。

使用了关键代码处理器，我们就有可能检查出由于内存值的改变而导致的“自发”错误（地铁隧道里存在严重的电子干扰）。

形式化方法和关键代码处理器的使用完全是互补的，前者保证软件本身（相对于需求）的正确性，而后者保证软件执行的正确性。

4.7 开发过程的集成

在工业中应用形式化方法，最困难的障碍是如何将这种方法集成到**软件开发过程**中，因为后者是今天人们正在使用的方法。

人们在使用这样的方法时都很不情愿，主要就是因为必须在很大程度上改变软件开发过程。众所周知，开发过程本身很难开发，而**更难**的是让参与工作的工程师接受它们。这也是经理们不希望改变开发过程的主要原因。

事实上，初始阶段和中间阶段（也就是构造抽象模型和构造具体模型）远比更传统的软件开发中的类似阶段（即，技术规范和设计）重要得多。当然其代价也更高一些。而另一方面，最后的几个阶段（程序设计、集成和测试）则远不那么重要（代价也小得多）。但经理们并不喜欢这种开支模式的改变，因为他们不相信在项目开始花更多的时间和金钱，将在最后节省大量时间和金钱。

除了阶段规模方面的这些变化之外，开发过程中的主要变化是结合了证明活动。经理们会害怕工程师没有能力做交互式证明。

在本文研究的领域（轨道列车系统）里，大部分生产商都没有使用形式化方法。虽然在这个领域里存在一些对应于**安全完整性级别 4** 的需求标准，形式化方法（带有证明，如这里讨论的情况）还没有很好推广。要问为什么出现这种情况，厂商给出的标准回答是他们没理由采用形式化方法，因为潜在客户没提出这种要求。换句话说，他们不希望在这一领域特别投资，因为没有足够的客户。他们认为在这里投资不能得到更多客户的回报。

4.8 指导软件工程师

经验说明工程师很容易学习在形式化方法中使用的数学概念和记法形式。更困难的问题是获得开发形式化易于理解并容易证明的模型的能力。开始阶段工程师有一种写伪代码程序的倾向，所做的不像是在构造真正的模型。

按照我的观点，建模和写软件需求文档是两项重要训练，它们在计算机科学教学计划里的地位应该大大提升。我不认为用 UML 的课程能涵盖这方面的问题。开设这种新课程时应该特别关注实践。学生必须在这些训练中自己练习，这一点现在还没有被很好地认识。在这种训练中，一件很清楚的事情就是需要和相应工具结合起来，使学生可以用它们，否则课程就可能太抽象。我已经在苏黎世的瑞士联邦工学院开发了两门这样的课程，一门是本科生水平的课程，另一门是硕士生课程。

4.9 更少使用程序设计语言

从使用形式化方法中学到的一个重要认识，就是高级程序设计语言作为开发者基本工具的作用大大降低了。在第 3 节的例子里，工程师在开发安全攸关的部分时根本就没用 ADA 语言（请回忆一下，3.3 节解释过工程师不碰 ADA 代码）。当然，因为软件的其他部分没用形式化方法开发，根据实际需要，目前还是把具体模型的最后精化结果翻译到 ADA 语言。

事实上，有关源代码和目标代码，以及两者之间有一个编译器把前者翻译到后者的一系列概念，现在几乎都不见了。正如 2.5 节所说的，我们确实有最后的目标代码，但可能应该说不是只有一份源代码，而是对应于构造过程中的各个精化步骤，存在多个层次的源代码。（其中有些很抽象，从抽象模型到具体模型可能总共有超过 20 层！）

很明显，采用一种语言（类似于程序设计语言）定义这么多不同的模型层次可能不是一种正确方式。可能更好的方式是用一个数据库，其中能保存各层次的模型，它们之间用精化和分解关系联系起来。这一数据库将包含一批基本**模型元素**，其成分在第 2 节有简单讨论：数据、不变式、变换、精化层次、证明义务、证明等等。工程师可以浏览这个数据库，在开发抽象和具体模型的过程中，数据库也**一直在演化**。这样，证明义务生成器、证明器和精化器就可能部分取代工程师的位置，直接基于数据库完成一些更事务性的自动化工作。它们也当然会把工作的结果存入数据库。注意，这一数据库周围还可能安装许多其他工具。

4.10 形式化方法的更多使用

在 4.1 节里我们指出了软件需求文档的一些弱点。大部分情况中这一文档都很具体，其中不仅包含了未来软件的完整定义的体系结构，还有构成这一体系结构的各种模块的细节描述。当然这样做并没有错。仅有的问题就是它（被通常很有能力的人们）用非形式的方式定义。这样定义的结果使它可能包含一些**先期错误**。如 4.1 节所言，这些错误常常不能在后续阶段中通过使用形式化方法检查出来。还有如 4.3 节所言，这种错误完全可能要到**过程中很后面**

的阶段才能发现，直到执行全局测试的时候，这种情况有潜在的危险。

改善这一状况的一种可能性是采用某种形式化方法去构造软件需求文档本身。做这件事时，可能采用某种与前面提出的模型构造方法类似的方法。当然，此时我们需要从一种更原始的非形式文档出发，可以将其称为**系统需求文档**。这一文档应该包含我们未来系统（由软件和设备组成）的最重要需求，但是其中并不提及任何体系结构，甚至不必明确未来的软件部分与未来的物理设备部分之间的划分。

在这个层次上，模型的用途（如常）就是为能通过一系列精化步骤逐步构造，以获得某种和谐的符合系统需求的体系结构，有时也可能考虑相互对应的不同方面（即，安全性和功能）。解释这件事可以如何做已经超出了本文范围。现在我只想说，这里的基本想法是构造如 [3] 所提出的一些封闭模型。阐释这种方法的一本书正在准备之中，不日即可面世 [2]。

5. 总结

在本文中，我简单介绍了两个现存的并正在使用的工业案例，其中安全攸关的部分是用 B 形式化方法开发的。我还对使用这种方法的优势和困难做了一些评述，并试图解释为什么引进这种方法的事宜受到某些工业经理们的阻挠。

6. 参考文献

[1] J.-R. Abrial. *The B-Book*: Assigning Program to Meanings. CUP, 1996.

[2] J.-R. Abrial. *Event-B*. To be published, 2006.

[3] R. Back. Decentralization of process nets with centralized control. *Distributed Computing*, 1989.

[4] F. Badeau. Using B as a high level programming language in an industrial project: Roissy val. In Proceedings of ZB'05, 2005

[5] P. Behm. Metreor: A successful application of B in a large project. In Proceedings of FM'99, 1999

[6] L. Burdy. Vital coded microprocessor: Principles and applications for various transit systems. In Proceedings of IFAC-GCCT 1989, 1989.

[7] L. Burdy. Automatic refinement. In Proceedings of BUGM at FM'99, 1999

[8] D. Dolle. Vital software: Formal methods and coded processor. In Proceedings of ERTS 2006, 2006

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Massachusetts, 1995.

[10] Rodin. European Project Rodin. http://rodin.cs.ncl.ac.uk

[11] Siemens. Siemens transportation systems, 2006. http://www.siemens.fr/transportation.