维普资讯 http://www.cqvip.com

从构造性证明中提取程序*)

石铠源　庞建民　赵荣彩　戴　超
(信息工程大学信息工程学院计算机科学与技术系　郑州 450002)

摘　要　通常情况下,我们很难确定一段给定的程序是否符合它的规范,程序提取是一种从构造性证明中提取函数式程序的机制,其构造特性很好地保证了生成程序的正确性。这就为我们提供了一种开发正确性软件的方法。本文基于对 Coq 中程序提取机制的研究,阐述了它的理论基础、实现机制及应用。

关键词　程序提取,Coq,构造性证明,Curry-Howard 同构,归纳构造演算

Program Extraction from Constructive Proof

SHI Kai-Yuan　PANG Jian-Min　ZHAO Rong-Cai　DAI Chao
(Institute of Information Engineering, Information Engineering University, Zhengzhou 450002)

Abstract　It is very difficult to decide in general whether a given program meets its specification. Program extraction is a mechanism that one can automatically extract a functional program from a constructive proof, which by its very construction is correct as well. This offers us a way to produce correct software. This paper presents the theoretical background, realizability and application of the program extraction mechanism based on Coq proof assistant.

Keywords　Program extraction, Coq, Constructive proof, Curry-howard isomorphism, Calculus of inductive construction

1　简介

从构造性证明中提取程序,这一机制已经有大约 20 年的历史了,在 Coq[3] 系统中最早的实现是 1989 由 C. Paulin 设计完成的。Coq 是由法国国家计算机科学与自动控制研究所(INRIA)研制开发的一种基于高阶逻辑的证明助手软件。

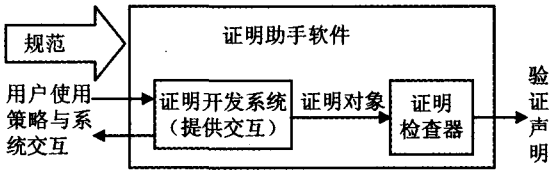

图 1　证明助手软件

通过它,我们可以定义函数或谓词;声明数学定理和形式化规范;交互式地构造证明并检查证明是否正确。目前 Coq 系统被应用到定理证明、协议验证和程序验证等诸多领域。

为什么 Coq 能够提供程序提取的机制呢? 这是由 Coq 所基于的理论框架所决定的——Coq 的元语言是归纳构造演算(Calculus of Inductive Constructions, Cic)[3],它是 λ 演算[4] 的扩展。它支持构造性证明,构造性的证明与计算机程序概念有密切关系,例如:为构造地证明命题 $(\forall x\in A)(\exists y\in B)P(x,y)$,必须要给出函数 f,使 f 应用于 A 中元素 a 时,产生 B 中元素 b,使 $P(a,b)$ 成立。如果 $P(a,b)$ 描述了一个规格说明(specification),则证明该命题的函数 f 就是满足该规格说明的一个程序[10]。根据 Curry-Howard 同构[7],构造性证明可以同构于函数式语言。

此外,其它的证明助手软件像 Minlog, Isabelle/HOL[1] 以及定理证明器 PX 和 Nuprl[9] 也提供程序提取的机制,通过提取程序进行程序验证在程序正确性保证方面发挥着越来越大的作用。

在这里,我们讨论的是 Coq 7.0 版本之后使用的程序提取机制,它在表示能力,执行效率,代码优化等多方面都较之前的版本有了很多的改进。目前,这一机制支持的目标函数语言包括 Ocaml、Haskell[8] 和 Scheme[9]。

2　理论背景

1985 年,T. Coquand 用扩展的 Automath 语言把 Martin-Löf 直觉主义类型论[2] 和 Girard 的多态 Fw 演算结合起来,首次提出构造演算(Calculus of Construction, CoC)。这是一个很强的逻辑系统,能够进行有力的公理化,但无法直接表示归纳定义。归纳概念必须通过函数间接地定义,因此系统效率较低并且繁琐。1989 年,T. Coquand 和 C. Paulin 用原始归纳定义扩展该形式系统,形成了归纳构造演算[3]。

Cic 的逻辑是构造性的逻辑,所谓构造性是指:与经典逻辑只关心公式的真值不同,构造逻辑关注的是实际的证明对象本身,能具体地给出某一对象或者能给出某一对象的计算方法。即当我们把能证实"存在一个 x 满足性质 A"的证明称为构造性的,是指能从这个证明中具体地给出满足性质 A 的一个 x;或者能从此证明中得到一个机械的方法,使其经有限步骤后即能确定满足性质 A 的这个 x 来[11]。

下面我们来简单介绍一下 Cic。

2.1　归纳构造演算(Cic)

2.1.1　Cic 的语法结构

Cic 的语法如图 2 所示。

*)本课题得到欧盟项目 TYPES(项目编号:types project 29001)资助。**石铠源**　硕士研究生,主要研究方向:计算机辅助推理与安全性保证;**庞建民**　博士,教授,主要研究方向:计算机辅助推理与软件理论;**赵荣彩**　教授,博士生导师,主要研究方向:分布式计算与软件理论;**戴　超**　硕士研究生,主要研究方向:计算机辅助推理与漏洞发现。

维普资讯 http://www.cqvip.com

表达式：
t ::= s
| x | c | C | I
| $\forall$ x : t,t | λ x : t, t | let x := t in t | (t t)
| Case (t ,t,...,t)
| fix x_i $\{x_1/k_1 : t := t \ldots x_n/k_n : t := t\}$

图 2　Cic 的语法

它可以表示变量，常量，归纳结构和归纳类型；λ 抽象，let 定义，函数应用，分支条件式，以及递归式等。其中 s 是一个特殊的常量，它是良型（well_formed）的类型，具有累积（cumulative）特性，即如果一个项是 Set 类型，它同时也是 Type_i 类型的。具体表示如图表 3 所示，Type_0 就是 Set 和 Prop 的类型，Type_1 则是 Type_0 的类型，依次类推。

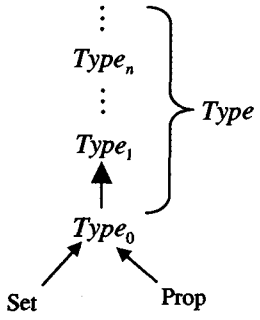

图 3　Cic 中的类型

Cic 的类型判断的表示形式是 $\Gamma \vdash t : T$，它表示 T 是 t 在上下文 Γ 中的有效类型。

2.1.2　Cic 的类型规则（typing rules）

每一个语法项，都有它特定的表示形式。图 4 给出了 Cic 中每一个语法项的类型规则，规范的格式保证了我们可以根据不同类型来编写不同的程序提取函数，实现与每一个语法项相对应的提取。例如：对于递归定义 $Fix\ f\{f/1 : A := t\}$，它可以被提取为 *let rec* $f\ x = (\varepsilon(t)x)$，具体的提取函数请参考文[6]。

2.1.3　Cic 的规约规则（reduction rule）

图 5 列出了 Cic 的规约规则

beta : $((\lambda x : X, t)u) \rightarrow_\beta t\{x \leftarrow u\}$;
belta : $c \rightarrow_\delta t$,
　如果当前上下文 Γ 中包含 $(c := t : T)$;
zeta : $let\ x := t\ in\ u \rightarrow_\zeta u\{x/t\}$;
对分支条件式，有：
iota : $\mathrm{Case}(C_i \vec{p}\,\vec{u}, P, f_1, \cdots, f_n) \rightarrow_\iota f_i \vec{u}$
　其中 C_i 是包含 $|\vec{p}|$ 个参数的归纳类型的第 i 个构造子；
对递归式，有：
iota : $if\ F : f_1/k_1 : A_1 := t_1 \cdots f_n/k_n : A_n := t_n\ then$
$(\mathrm{Fix}\ f_i\{F\}u_1 \cdots u_{k_i}) \rightarrow_\iota (t_i\{f_j \leftarrow fix\ f_j\{F\}\}_{\forall j} u_1 \cdots u_{k_i})$

图 5　Cic 的规约规则

简单起见可以用 $\rightarrow_r$ 来表示所有的规约。通过规约，可以对证明对象进行化简，得到最为基本的结构，便于我们进行提取。

$$WF(\phi)\quad \frac{\Gamma \vdash T : s \quad x \notin \Gamma}{WF(\Gamma;(x:T))} \qquad \frac{\Gamma \vdash t : T \quad c \notin \Gamma}{WF(\Gamma;(c := t : T))} \quad (\mathrm{WF})$$

$$\frac{WF(\Gamma)}{\Gamma \vdash \mathrm{Set} : \mathrm{Type}_i} \qquad \frac{WF(\Gamma)}{\Gamma \vdash \mathrm{Prop} : \mathrm{Type}_i} \qquad \frac{WF(\Gamma) \quad i < j}{\Gamma \vdash \mathrm{Type}_i : \mathrm{Type}_j} \quad (\mathrm{Ax})$$

$$\frac{WF(\Gamma) \quad (x:T) \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{WF(\Gamma) \quad (c := t : T) \in \Gamma}{\Gamma \vdash c : T} \quad (\mathrm{Var})(\mathrm{Cst})$$

$$\frac{\Gamma \vdash T : s_1 \quad \Gamma;(x:T) \vdash U : s_2 \quad P(s_1, s_2, s_3)}{\Gamma \vdash \forall x : T, U : s_3} \quad (\mathrm{Prod})$$

$$\frac{\Gamma \vdash \forall x : U, T : s \quad \Gamma;(x:U) \vdash t : T}{\Gamma \vdash \lambda x : U, t : \forall x : U : T} \qquad \frac{\Gamma \vdash t : \forall x, U, T \quad \Gamma \vdash u : U}{\Gamma \vdash (t\,u) : T\{x \leftarrow u\}} \quad (\mathrm{Lam})(\mathrm{App})$$

$$\frac{\Gamma \vdash t : T \quad \Gamma;(x := t : T) \vdash u : U}{\Gamma \vdash \mathrm{let}\ x := t\ \mathrm{in}\ u : U\{x \leftarrow u\}} \quad (\mathrm{Let})$$

$$\frac{\Gamma \vdash U : s \quad \Gamma \vdash t : T \quad T \leq_{\beta\delta\zeta} U}{\Gamma \vdash t : U} \quad (\mathrm{Conv})$$

$$\frac{WF(\Gamma) \quad \mathrm{Ind}_n(\Gamma_I := \Gamma_C) \in \Gamma \quad (I : A) \in \Gamma_I}{\Gamma \vdash I : A} \quad (\mathrm{I\text{-}Type})$$

$$\frac{WF(\Gamma) \quad \mathrm{Ind}_n(\Gamma_I := \Gamma_C) \in \Gamma \quad (C : T) \in \Gamma_C}{\Gamma \vdash C : T} \quad (\mathrm{I\text{-}Cons})$$

for all $(I : A) \in \Gamma_I, \Gamma \vdash A : s$
for all $(C : T) \in \Gamma_C$, $\Gamma;\Gamma_I \vdash T : s_C$

$$\frac{\mathrm{I}_n(\Gamma_\mathrm{I}, \Gamma_\mathrm{C})}{WF(\Gamma;\mathrm{Ind}_n(\Gamma_I := \Gamma_C))} \quad (\mathrm{I\text{-}WF})$$

$$\mathrm{Ind}_n(\Gamma_I = \Gamma_C) \in \Gamma \qquad (I : \overrightarrow{\forall p : T}, A) \in \Gamma_\mathrm{I} \qquad \sigma = \left\{\vec{p} \leftarrow \vec{q}\right\}$$

$$\left|\vec{p}\right| = n \quad \Gamma \vdash e : I\ \vec{q}\ \vec{u} \quad \Gamma \vdash P : B \quad C(I\ \vec{q} : A_\sigma; B)$$

for all constructor $C_i : \overrightarrow{\forall p : T}, \overrightarrow{\forall x : X}, I\ \vec{q}\ \vec{y}$,

$$\frac{\Gamma \vdash f_i : \overrightarrow{\forall x : X_\sigma}, P\ \vec{y}_\sigma(C_i, \vec{q}, x)}{\Gamma \vdash \mathrm{case}(e, P, f_1 \ldots f_m) : P\ \vec{u}\ e} \quad (\mathrm{Case})$$

$$\frac{\forall i\ \Gamma \vdash A_i : s_i \quad \forall i, \Gamma;(\overrightarrow{f : A}) \vdash t_i : A_i \quad F(\vec{f}, \vec{A}, \vec{k}, \vec{t})}{\Gamma \vdash \mathrm{fix}\ f_i\{f_1/k_1 : A_1 := t_1 \ldots f_n/k_n : A_n := t_n\} : A_j} \quad (\mathrm{Fix})$$

图 4　Cic 的类型规则

基于 Cic 的各种规则，Coq 系统内部把构造性证明表示为 λ 项，也就是说构造性证明具有计算能力，为什么呢？非构造性证明是没有计算能力的，例如纯存在性证明只能让人知道某个方程的根是存在的，但如何求解以至能不能求出这个根均是未知的。构造性证明就是针对纯存在性证明的这个缺陷提出的，要证明一个方程的根是存在的，就必须给出求解它的有效方法。因此，构造性证明的过程就是一个计算的过程。我们知道 λ 演算是函数式语言的理论基础，是函数式程序设计语言的纯理论部分的范式，是由函数抽象和函数应用组成的系统。而这两个特点也是程序设计语言所具有的共同之处：抽象与过程定义机制对应，应用与过程调用对应[10]。Curry-Howard 同构是对构造性证明和函数式程序之间这种关系的理论解释。

2.2　Curry-Howard 同构

1958 年，Curry 发现了简单带类型的 λ 演算可以同构于命题直觉主义逻辑，1969 年，Howard 又发现带依赖类型的 λ 演算可以同构于一阶直觉主义逻辑。之后，随着不断的丰富，

维普资讯 http://www.cqvip.com

这种同构的表示范围越来越广，可以表达包括 Martin-Löf 理论、Cic 等各种直觉主义逻辑系统。

如图 6 所示，Curry-Howard 同构在构造逻辑和函数式语言的计算之间建立了重要的对应关系：

构造逻辑	函数式语言
$\dfrac{\Gamma,A\vdash B}{\Gamma\vdash A\Rightarrow B}$	$\dfrac{\Gamma,x:\Gamma\vdash t:B}{\Gamma\vdash \lambda x:A.t:A\to B}$
$\dfrac{\Gamma\vdash A\Rightarrow B\ \Gamma\vdash A}{\Gamma\vdash B}$	$\dfrac{\Gamma\vdash t:A\to B\quad \Gamma\vdash a:A}{\Gamma\vdash (t\ a):B}$
$\dfrac{\Gamma\vdash A\ \Gamma\vdash B}{\Gamma\vdash A\wedge B}$	$\dfrac{\Gamma\vdash a:A\quad \Gamma\vdash b:B}{\Gamma\vdash a,b:A\times B}$
$\dfrac{\Gamma\vdash A\wedge B}{\Gamma\vdash A}$	$\dfrac{\Gamma\vdash t:A\times B}{\Gamma\vdash fst\ t:A}$

图 6　构造逻辑与函数式语言逻辑的同构

在 Coq 中我们可以将这种同构关系具体地表述为命题作为类型和证明作为 λ 项。下面以命题(A->B->C)->(A->B)->A->C 为例，阐述一下这种同构关系。

在手工证明情况下的证明步骤如下：

```
A->B->C,A->B,A|-A
                由假设 A                    1
A->B->C,A->B,A|-B
                由假设 A->B 和 1            2
A->B->C,A->B,A|-C
                由假设 A->B->C 和 1,2       3
|-(A->B->C)->(A->B)->A->C                   4
```

与手工证明不同，Coq 通过向用户向系统应用策略进行交互式证明，自顶向下不断地对目标进行求精，达到可以自动推理的地步。在这里，因为命题逻辑是可判定的，Coq 提供的 auto 策略可以进行自动证明：

```
>Parameter A B C:Prop.
>Theorem example:(A->B->C)->(A->B)->A->C.
>Proof.
>auto.
>Qed.
```

将证明结果打印出来，我们看到：

```
>example=
fun(H : A->B->C)(H0 : A->B)(H1 : A)=>
H H1(H0 H1)
   : (A->B->C)->(A->B)->A->C
```

不难看出，Coq 内部实际上将证明目标看作一个 λ 项，证明的过程就是对 λ 项的类型检查过程，我们看一下这个过程：

```
H : A->B->C,H0 : A->B,H1 : A=>H1 : A        1
H : A->B->C,H0 : A->B,H1 : A=>H0 H1 : B     2
H : A->B->C,H0 : A->B,H1 : A=>H H1(H0 H1)
: C                                         3
=>lambda……H H1(H0 H1) : (A->B->C)->(A->
B)->A->C                                    4
```

对比前面的手工证明，将项去掉，并将类型看成命题，容易看出，它与手工证明过程是一致的。

3　程序提取实现

因为 Coq 证明的内部表示是 λ 项，我们可以说 Coq 的证明项就是函数式程序。但这时的函数式程序是不规范的，我们需要对它进行一定的转换工作，让它变为符合一定语法规范的程序。这主要涉及两方面的工作，一是消除逻辑部分，二是转换为实际的程序。

3.1　逻辑部分的消除

一个构造性证明包含信息部分和逻辑部分，其中只有信息部分有计算意义，对我们提取程序是有用的，逻辑部分只用来保证某些属性，没有计算意义。把这些内容提取出来只会增加程序的冗余而降低程序执行效率。根据 Cic 的类型规则，Coq 系统的基本类型中，Prop 表示逻辑命题，例如 True 和 False；Set 表示数学类集(mathematical collections)，例如自然数类型 nat，布尔类型 bool 等。Set 类型的对象构成信息部分，而 Prop 类型的对象构成逻辑部分，所以，系统可以通过用户的声明和定义容易地区分出逻辑部分和信息部分。同时，Coq 的类型系统可以保证逻辑部分不会影响信息部分中的计算内容。所以在提取的时候就可以消去所有属于 Prop 类型的对象。去掉了逻辑部分，提取出来的程序自然就避免了逻辑错误。

3.2　转换为实际的函数式程序

关于生成什么样的代码，我们有两种选择：某一特定语言的源代码或是直接的二进制码，最终选择了前者，因为它简单，可读性好，易于编译优化等。那么，我们应该选择哪种特定的语言作为目标语言呢？Cic 特殊的结构要求我们需要带有归纳类型 λ 演算，所以选择了 OBJECTIVE CAML、HASKELL 等类 ML 语言。

底层实现机制的相似保证了大部分的 Coq 表示都可以直接对应到类 ML 语言的表示，但 Coq 的类型系统与类 ML 语言的类型系统是有差别的，这就会带来一些异常，例如，从 Coq 中提取出来的项，类 ML 语言没有相关的项与之对应，或是 Coq 中提取出来的项的类型无法被类 ML 语言的类型系统识别。解决这一问题，对于 Ocaml 是通过加入 Obj. magic 函数，对那些无法判别类型的项用'a 表示，'a 可以表示任意项，具体可以参考下面的例子。对于 Haskell，使用函数 unsafeCoerce 实现和 Obj. magic 同样的功能。而 Scheme 是无类型的，所以我们可以暂时不考虑。

最后，为了让生成的代码可读性更好，执行效率更高，需要进行代码优化，例如去掉一些不必要的归纳类型；内嵌某些函数，跳过一些没有意义参数求值等。

3.3　程序提取实例

首先，证明命题："在一组有穷多个自然数当中必然存在一个最大的数"，之后从证明当中抽取出一个函数，它可以用来计算最大值。

首先用归纳定义给出规范 maxn：

该归纳定义包括三个构造子(constructor)：maxnO 表示如果集合只有一个元素，最大值就是这个元素；maxnSg 表示如果 m 是前$(n+1)$个自然数组成的集合中的最大值，且 m 比第$(n+2)$个数大，则 m 是前$(n+2)$个自然数组成的集合中的最大值；maxnSl 表示如果 m 是前$(n+1)$个自然数组成的集合中的最大值，且 m 比第$(n+2)$个数小，则第$(n+2)$个数是前 $n+1$ 个自然数组成的集合中的最大值：

维普资讯 http://www.cqvip.com

```
>Require Arith.
>Require Compare_dec.
>Inductive maxn (f:nat->nat):nat->nat->Prop:=
      maxnO : (maxn f O (f O))
    | maxnSg : forall n m:nat,(maxn f n m)->
        (ge m (f (S n)))->
        (maxn f (S n)m)
    | maxnSl : forall n m:nat,(maxn f n m)->
        (le m (f (S n)))->
        (maxn f (S n)(f (S n))).
```

接下来我们来证明对于所有的 f 和 n 存在一个自然数 m，它是集合$\{f(0),\cdots,f(n)\}$的最大值，证明是构造性的，篇幅有限，没有完全列出：

```
>Lemma max_val : forall (f:nat->nat)(n:nat),
      exists m:nat,maxn f n m.
>Proof.
...
>Qed.
```

这时，系统会返回我们交互信息“max_val is defined”。现在我们用 Coq 相关命令来提取程序：

```
>Require Extraction.
>Extraction Language Ocaml.
>Extraction“maxv” max_val.
```

得到的 maxv.ml 就是经过验证了的 Ocaml 程序，其代码如下：

```
>let max_val f n=
  let rec f2=function
    O->f O
  | S n1->match le_ge_dec (f2 n1)(f (S n1))
with
          Left_ren -> f (S n1)
        | Right_ren -> f2 n1)
  in f2 n
```

以上代码已经在 Coq 8.0pl2 和 Ocaml v3.08.4 上进行过验证。

结束语 经过十多年的发展，Coq 的程序提取机制从最开始的只是一个试验性质的工具变为一个成熟的程序验证框架。其表示能力，正确性和交互性都有了很大的提高，它在程序验证方面起着越来越重要的作用。但这并不说明这一机制已经相当完善了，首先，通过程序提取器提取出来的程序虽然进行了大量优化工作，仍然有一定的冗余结构，增加了程序的尺寸；其次，这一领域的大规模的提取工作不多，因此还缺乏进行大规模验证工作的库，需要我们进一步地补充和丰富。我们也提倡感兴趣的读者使用该系统，为系统验证库的扩充作出贡献。

致谢 感谢英国伦敦大学罗朝晖(Prof. Zhaohui Luo)教授、英国杜伦(Durham)大学保罗·卡拉汉博士(Dr. Paul Callaghan)的帮助和指导，感谢来自欧盟和杜伦大学的资助。

参 考 文 献

1 Berger U, Berghofer S. Program extraction from normalization proofs, 2005
2 Nordström B, Petersson K, Smith J. Programming in Martin-Löf's type theory. An introduction. Oxford University Press, 1990
3 The Coq Development Team. The Coq Proof Assistant Reference Manual (Version 8.0), 2005
4 Barendregt H P. Lambda calculi with types. In: Samson Abramsky. Dov. M. Gabbay. Thomas S. E. M aibaum, eds. Handbook of Logic in Computer Science. Oxford University Press, 1992, 2
5 Coquand T, Huet G. The calculus of constructions. Information and Computation, 1988
6 Letouzey P. Certified functional programming. Program extraction within Coq proof assistant: [PhD manuscript]. 2004.7
7 Heine M, Urzyczyn P. Lectures on the Curry-Howard Isomorphism. 1998.5
8 庞建民，赵荣彩. Haskell 语言的高阶特性及其应用. 计算机科学，2005(6)
9 Letouzey P. A New Extraction for Coq. Types for Proofs and Programs, Second InternationalWorkshop. Springer-Verlag, 2003
10 蒋慧，林东，孙泉，谢希仁. 构造类型论与计算机程序设计. 计算机科学，2002，29
11 郝宁湘. 构造性数学及其哲学意义. 自然辩证法通讯，1997

(上接第 265 页)

式(1)将完工成本的计算划分为两大部分：已成事实部分和对剩余任务的预测部分。假设剩余任务的完成情况和已完成任务的成本花费情况是近似的，因此用当前情况下采集的成本执行指数参与计算；式(2)也将完工成本的计算划分为两大部分：已成事实部分和对剩余任务的预测部分。但它更注意对后续阶段任务划分和成本的改进，对剩余工作成本采用重新估算的数据值，这种方法更切合实际，如果估算方法得当，估算人员有一定的工作经验，那么式(2)的 EAC 的计算结果将比式(1)精确。但是，对剩余工作成本的重新估算是需要时间和人力资源的，即估算是有成本的。应该在精确程度和花费的成本二者之间寻求一个平衡，也就是要根据项目组织自身的工作能力、项目的大小和成本以及进度的限制为先决条件，因地制宜地进行 EAC 的估算。式(3)是最简单也是最不精确的估算。它仅仅以目前项目的状况作为整个项目状况的典型代表，不考虑改进工作对项目的影响，更适合于比较小的项目。

3)跟踪偏差纠正过程，直到偏差消除。

结束语 本文主要围绕挣值分析法讲述了对软件过程管理中的项目跟踪与监控的实施背景与实施步骤，对 SPTO 的具体实施有重要的指导意义。

参 考 文 献

1 Shandilya R T(印)著. 软件项目管理[M]. 王克仁译. 科学出版社，2002
2 Conradi H, Fuggetta A. Improving software process improvement [J]. Software. IEEE, 2002
3 陈海燕，梁成才，等. CMM 二级 KPA 软件项目跟踪和监督的一个设计实施方案[J]. 计算机工程，2004
4 郑人杰，王纬，等. 基于软件能力成熟度模型(CMM)的软件过程改进—方法与实施[M]. 清华大学出版社，2003