程序设计规则检查:一种保障软件质量的基本方法*)

李书浩　齐治昌
(国防科技大学计算机学院　长沙410073)

Programming Rule Checking: An Underlying Practice for Software Quality Assurance

LI Shu-Hao　QI Zhi-Chang
(School of Computer Science, National University of Defense Technology, Changsha 410073)

Abstract　Programming rule checking is an important approach of quality assurance in the coding phase of the software lifecycle. In this paper we introduce the notion, the objective, and the general procedures of programming rule checking. Following that, we examine how rule checking improves software quality from the perspectives of validity, understandability, portability, and safety. The tool support of rule checking is also presented. After making investigations on the mechanism of rule checking, we propose some approaches on the generalization, on the standardization, as well as on the practical application of rule checking.

Keywords　Programming standards, Styles, Rule checking

1　引言

程序设计风格是程序员在遵循程序设计语言语法规则的前提下编码的喜好和习惯。高级程序设计语言为程序员提供了越来越多的自由,使他们可以不拘泥于固定的格式来书写程序,并且允许他们以尽量简洁的方式书写。例如,有些编译器会自动给未初始化的变量赋缺省值。

但是,程序员仍然应当遵循一些程序设计的规则,忽视这些规则常常会引发软件错误,其中有些错误很难排查。我们来看下面的C语言语句:

```
if(flag &&((x=y)==0))
```

显然,这句话语法正确,语义明显。但是按照MISRA C程序设计标准,它却违背了我们的规则。MISRA C标准中的规则[MISRA33]告诉我们:应该避免使赋值号(=)出现在布尔表达式的&&或||之右。如果出现在它们之右,那么这种赋值就成了条件赋值。我们知道:

· 在if (flag && ((x=y)==0))中,若flag为FALSE,则不必继续后面的赋值操作;

· 在if (flag‖((x = y)==0))中,若flag为TRUE,则不必继续后面的赋值操作。

在上面的C语言语句中,如果flag为TRUE,那么该语句能按作者的意图执行;如果flag为FALSE,那么就导致了一个典型的因编码不慎而引起的软件错误。

程序设计标准是程序设计规则的集合。如果我们能制定出一套比较系统的规则,那么我们就得到了一套指导我们如何编码的标准。正是基于此考虑,英国汽车工业软件可靠性协会(MISRA)于1994年提出了针对C语言的标准——MISRA C程序设计标准[4]。MISRA C标准从程序的静态分析、静态数据流分析、复杂性分析等各方面对编码提出了具体要求。MISRA C标准已经被越来越多的公司所采纳,基于这种标准的代码规则检查也被越来越多的工具所支持,如LDRA Testbed, Logiscope RuleChecker, Polyspace Verifier等等。实践证明,采用诸如MISRA C程序设计规则和Ada的SPARK安全子集之类的标准能够使得各公司在可靠编码(solid coding)的基础上来进行项目开发[1]。

本文介绍程序设计规则检查的性质、任务和过程;介绍几种有代表性的程序设计标准、标准的(多视角)分类;用具体例子作详细阐述关于规则检查如何从改善正确性、可理解性、可移植性、安全性等方面保障软件质量;介绍规则检查的工具支持;对规则检查作了进一步的思考,包括规则检查应用的推广、规则库的组织与维护,以及规则检查的实用化。最后是小结。

2　程序设计规则检查:任务和过程

2.1　程序设计规则检查的任务

程序设计规则检查不是为了发现程序中的语法错误,因为这是编译器要做的;它也不是为了发现逻辑错误,因为那是程序员应该做的。它仅仅是为了发现因为程序员的粗心而造成的潜在错误和容易导致错误的部分,而这些又恰好是合乎程序语言的语法要求而编译器却不能发现的。规则[MISRA53]指出:程序中应该避免出现不起任何作用的语句。语句"x ;"就没有作用。显然,这个语句是合法的,编译器检查不出它的错误,但它很可能是程序员因疏忽而忘了编写相应代码的结果,规则检查碰到这种情况就会发出警告。

2.2　程序设计规则检查的过程

程序设计规则检查主要是对程序源代码作静态分析。被检查的对象可以是单个的源文件,也可以是头文件,甚至可以是整个系统的代码。这种检查是对程序源代码扫描,进行词法和语法分析,或者静态数据流分析,但不会涉及到动态成份。要想进行完全的规则检查,程序源代码应该没有语法错误。

程序设计规则检查可以穿插在编译器的词法分析和语法分析之前、之中和之后进行。穿插进行的好处是,开发规则检查工具无需从词法分析、语法分析做起,也节省了规则检查工具重新做词法分析和语法分析的时间。

李书浩　博士研究生,主要研究方向为软件测试。齐治昌　教授,博士生导师,主要研究方向为软件工程等。

在编译器的预处理过程中，也就是代码扫描之后、词法分析之前，规则检查器可以检查代码格式、注释风格、字符集的合法性等。在词法分析之后、语法分析之前，可以检查标识符命名的合理性。在条件编译和宏替换时，可以检查它们是否满足相应的规则。在程序的语法分析过程中，则可以对控制结构、数据类型、运算等进行规则检查，还可以对程序作静态数据流分析，复杂性度量等等。规则检查的流程如图1所示。

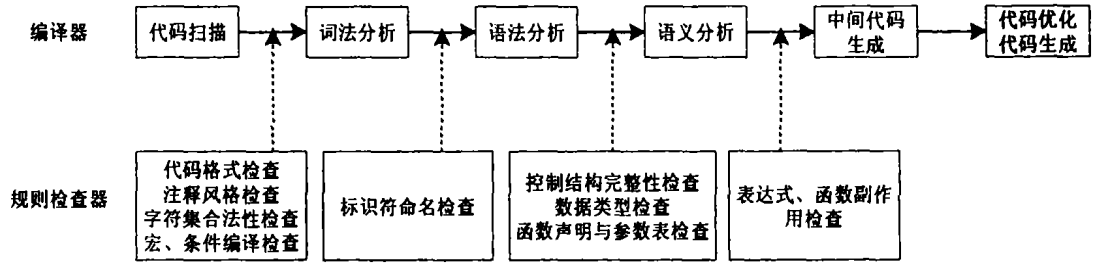

图1 程序设计规则检查流程

3 程序设计标准：现状和构成

3.1 程序设计标准现状

迄今为止，已经有很多单位和个人提出了各种程序设计规则，其中比较系统的、形成了程序设计标准的有以下几种：

(1)MISRA C/C++/Ada 标准 MISRA 标准是英国汽车行业软件可靠性协会(MISRA)1994年提出的程序设计标准[4]。MISRA 标准在汽车业界被广泛采纳。当然，它也被用于安全攸关或者业务攸关的系统。LDRA Testbed、Logiscope RuleChecker、Polyspace Verifier 等工具都支持此标准。

(2)DERA C/C++标准 DERA 标准是英国国防部的国防评估与研究局(DERA)在 LDRA 的帮助下开发的。它需要对源代码作比较复杂的数据流和信息流分析。工具 LDRA Testbed 支持这种标准。

(3) Telelogic 的 Logiscope RuleChecker 标准 Logiscope RuleChecker 既支持 MISRA C/C++标准，也有一套自己的标准，用户还可以按照其规则文件的格式来自行定义新的规则。

Logiscope C rule set 自己的标准包括如下几套规则：代码表示规则集(Code Presentation rule set)、复杂性规则集(Complexity rule set)、控制流规则集(Control Flow rule set)、名字规则集(Naming rule set)、移植性规则集(Portability rule set)、资源规则集(Resource rule set)。

(4)其它标准和规范 由于 C 和 C++语言在业界的广泛应用，也由于近年来对编码规范性要求的提高，许多公司和个人都提出了针对某些程序语言的编码原则和规范。比较著名的是 Motorola 公司 Steve Hawkes 和 Mare Peters 的 Guidelines for Effective C++，Scott Meyers[2] 的 Effective C++和 More Effective C++，SUN 公司的 C++ User's Guide (Forte Developer 6 Upgrade 2)等。国内也有这方面的工作，如 Motorola 中国公司陈世忠[5]的《C++编码规范》，它包括了300多条原则。

3.2 程序设计标准的构成

程序设计标准是一套比较系统的程序设计规则的集合。

以 LDRA Testbed 所采纳的 MISRA C 程序设计标准[1]为例，标准中的规则共200多条。从标准本身的性质和目的来看，可以分为六大类：

· 主静态分析标准：对源代码作词法和语法分析。源代码既可以是单个的文件(包括头文件)，也可以是整个系统。

· 复杂性分析标准：分析程序代码，以过程为单位报告其复杂性结构。如果有必要的话，还可以计算出过程、文件、甚至整个系统的复杂性度量值。

· 静态数据流分析标准：提供关于源文件或者系统的四类信息：过程调用信息；数据流 anomalies(施加于程序变量之上的可能错误的动作序列，如对变量进行两次连续定值，中间却没有使用该变量)；过程参数分析；全局变量分析。

· 交叉引用标准：在整个源文件或系统范围内对所有的数据元素(data items)作完全的交叉引用，从而知道每一个数据的使用类型(是用作全局变量、还是局部变量、或者是用作参数)，给出被分析程序的完全调用树的文本表示。

· 信息流分析标准：信息流分析也称为变量依赖分析，它研究程序变量之间的相互依赖关系(如控制依赖、数据依赖)。

· 质量报告标准：检查函数是否有副作用，程序控制的层次结构是否太深。

从程序设计语言角度看，LDRA Testbed 所采纳的 MISRA C 标准中的规则涉及了如下方面：①基本数据类型、运算符与表达式；② 标识符、常量/变量；③ 数组；④自定义数据类型 —— 枚举、结构、联合；⑤程序控制结构；⑥函数与过程；⑦指针；⑧预处理命令；⑨代码中的注释；⑩括号问题(为了使结构更清晰，防止歧义)；⑪对字符集/文件/过程/参数表/表达式的各种限制；⑫其它。

4 程序设计规则检查：保障软件质量的途径

我们从正确性、安全性、可理解性、可移植性等方面考察规则检查是如何保障软件质量的。主要以 MISRA C、DERA C，以及 LDRA Testbed 所特有的一些规则为例子以说明。

4.1 正确性

易出错的运算符：

[MISRA1]避免使用易出错的运算符。易出错的运算符包括=-、=+、+=、-=、++、--等，它们在书写上容易引起混淆。如，"$i=-1$"容易误写为"$i=-1$"

[DERA136]避免使用前缀++或--。这些运算符都比较简洁，使用这些运算符要倍加小心。C 语言的制订者当初就是想用这些运算符来减轻程序员的工作量，使代码更紧凑，同时为程序员提供表达上的灵活性。但是，像=-之类的运算符号很容易引起混淆。从提高软件质量的角度看，将 I =-K 写为 I=I-K 也是值得的。

易出错的运算：

[MISRA50]避免对浮点数作"相等"比较。由于浮点数是用二进制表示的，而二进制不能精确地表示所有的浮点数(存在舍入问题)，因此看似相等或不等的两个浮点数在计算机内部的表示可能不同(比如最后一位不等)，此时用"相等"或"不

等"这种精确比较方法就可能得出与期望相反的结果。

对浮点数作相等比较很容易出错。规则的背后所体现的还是数据的机器码表示。程序语言层次分辨不出的差别，当进行到编译层次和机器码层次时，差别就出来了。

4.2 可理解性

可读性：

[MISRA59]循环体(如 for 循环体)即便只有一句，也应该使用大括号。if-then-else 中的两个分支即便都只有一句，也应该使用大括号。如，If C then S1 else S2应该写为 if C then {S1} else {S2}。

[MISRA34]逻辑表达式中各个子表达式应该使用括号，虽然逻辑运算符有优先级。

以上几种情况不加括号也并不算错，但适当地添加括号后，能够使程序的结构更清晰，从而提高可理解性。但对编程人员来说，这在某种程度上施加了限制，使得他们的代码不那么紧凑。

结构完整性：

[DERA141]then 子句不能为空。

[MISRA62]switch 语句中没有 default 分支。如果不是程序员遗漏而是因为 default 语句为空，那么最好还是加入 default 以表示程序员考虑到了缺省情况。

对于软件的维护人员来说，他们显然希望程序的结构完整。完整的结构表明程序员已经考虑过了各种情况，没有遗漏。然而这要求程序员编码时显式地列出各种情况下的处理办法，即便这本来是不需要的，因此这增加了程序员的负担。

类型转换与类型检查：

[DERA144]条件表达式(ternary expression)中的类型不匹配。对条件表达式的两个候选项进行类型检查，看它们的类型是否匹配。如果不匹配，则要求用户进行显式的类型转换。显式类型转换的特点是清晰、直观，开发人员和维护人员不容易忽略。

[MISRA44]使用了冗余的类型转换。冗余的类型转换只会降低程序的可读性，令用户费解。

一般的规则检查工具只能做静态类型检查工作，即在编译时检查变量的类型，而不是在程序运行时才检查。非"强类型"的语言在定义类型时只需对变量名进行说明，而其类型可暂不说明。如，语言中的 union 类型。取消语言的"强类型"限制会使用户更方便地书写程序，在类型匹配上不必花更多的精力。但是这却引起了可读性、可移植性、安全性等一系列问题，它引起了类型匹配检查的麻烦，很多情况下不得不为此将类型检查推迟到程序运行时刻，这样也影响了程序运行的效率。

程序规模与程序复杂性：

[DERA145]LCSAJ 密度超过某值。这说明程序控制层次的结构太深，控制复杂性太高。

[DERA146]表达式中的子条件超过若干个。这说明布尔表达式中的子条件太多。

以上规则并不是编译器对程序所施加的限制，而是程序设计标准的制定人员从许多实际项目中总结出来的经验。规则的思想是模块化、功能分解。

注释风格：

[MISRA9]应该避免嵌套注释。"/*"、"*/"型的注释有一个匹配的问题。嵌套注释容易破坏这种匹配关系。即便嵌套的注释能够正确地匹配，不同的编译器对嵌套注释的处理也可能不同，有些编译器根本不支持嵌套注释。

[Testbed110S]应该避免使用单行注释"//…"。并不是所有的编译器都支持单行注释。应该遵循 ANSI C 的标准。

4.3 可移植性

可移植性指支持软件系统或工具在各类计算机上实现，以及从一种计算机移植到另一种计算机而不需作本质变动的能力。

[MISRA13]避免用基本类型来对变量进行声明。由于不同的编译器的基本类型的字节数可能会相差很大，因此要想使代码便于移植，基本类型只能在 typedef 中使用，不能在其它地方使用，如声明变量。

[MISRA46]防止一个表达式中出现多个函数。含有多个函数的表达式，其结果与执行时的计算顺序有关。

[MISRA46]防止表达式有副作用。如，x = A[j] + j++；x 的结果值和右边的计算顺序有关。这里表达式 j++ 就可能有副作用。

[原则23.15][5]不要假设函数参数的计算顺序。函数参数的计算顺序没有统一标准。从左到右、从右到左、甚至从中间到两边或者随机选择都有可能，它们在理论上都是合法的。如，在函数 function(*pArray++,*pArray++)中，可能是左边的参数先自增，也可能是右边的参数先自增。

可移植性主要涉及硬件和软件平台，包括机器字长、机器字编码顺序(big ending or little ending)、程序语言字符集、表达式计算顺序等方面。可移植性要求从一定程度上限制用户编码的灵活性，程序也可能有时显得繁琐，还需要用户了解相应的硬件、软件平台知识。

为提高可移植性，还有一点可以考虑，那就是数据抽象。在使用数据类型时，最好根据机器字长自行定义数据类型，避免直接使用寄存器、位运算等程序语言的低级特征。

4.4 安全性

安全性是确保所表述软件系统正常工作、不至于因为某一部分的操作不当而对其它部分或环境产生破坏的能力。

防止副作用：

[MISRA33]避免函数的全局变量带来副作用。函数通过全局变量和其它方式(函数参数、函数返回值)返回值，从而有副作用。

[MISRA33、46]避免函数的参数带来副作用。函数通过函数参数返回值，从而有副作用。

函数副作用会带来的很多问题，它从一定程度上破坏了软件的安全性。函数的作用可以通过几种途径表现出来：函数返回值、函数参数的返回值(非传值方式)、全局变量的变化、对其它对象的修改(如通过指针访问并修改其它变量或数据结构)。以上四种方式中最后一种尤其危险，第二种和第三种也不推荐采用，只有第一种方式比较好。

4.5 性能问题

通过遵从一些简单规则，可以用很小的代价换来性能的显著提高。

[MISRA28]不要申明 register 类型的变量。使用 register 变量可能会妨碍编译器对代码进行全面优化。编译器安排 register 的使用几乎总比手工安排 register 的使用做得好一些。

[原则24.3][5]如无必要，不要使用非 int 的整型类型。对任何平台，int 类型都和该平台的字长(16位、32位、或64位)匹配，非 int 的整数类型可能导致无法使用某些优化手段。

性能问题涉及到机器字长、操作系统调度、编译器优化等方面。如何编码能够更好地发挥机器的性能，用户需要了解一些相关的知识。

4.6 小结

通过以上分析可见，程序设计规则检查体现了一种矛盾，这就是程序设计语言提供给编程人员的灵活性和对硬件、软件平台缺乏足够了解的情况下随意利用这种灵活性开发出的软件的质量存在问题的矛盾。作为折中，程序员既要充分利用这种灵活性，也要遵循一些行之有效的程序设计规则和软件工程原则。程序员最好还能了解一些硬件和系统软件方面的知识。

5 程序设计规则检查：工具支持

各种著名的程序设计标准几乎都有相应的工具支持。对于像 MISRA C 这样有影响的标准，有很多的软件静态分析与代码检查工具都对其提供支持。

5.1 LDRA Testbed

LDRA Testbed 是英国 Liverpool 数据研究协会(Liverpool Data Research Associates)开发的程序设计规则检查工具。除了规则检查之外，它还能够计算软件质量、复杂性、可维护性方面的一些度量。LDRA Testbed 不仅支持程序静态分析，还支持动态分析，如程序的语句覆盖、分支覆盖、过程调用情况等。

LDRA Testbed 可对 C、C++、Ada 等语言进行程序设计规则检查，它支持 MISRA C/C++/Ada 标准，也支持 DERA C/C++标准。

LDRA Testbed 支持规则的扩充。用户如果想添加新规则，那么他必须先向 MISRA 递交提案。

5.2 Logiscope RuleChecker

Logiscope 是瑞典 Telelogic 公司的工具套集，包括 TestChecker、RuleChecker、ImpactChecker、Audit 等。其中的 RuleChecker 有 C、C++、Ada、Java 等语言的规则检查工具。它既支持 MISRA 标准，也有自己的规则集，还允许用户自行定义新规则。用户也可以根据需要修改 RuleChecker 规则集中的规则。

Logiscope RuleChecker C 的规则集合包括如下6个子集，共110多条规则：① Code Presentation rule set，代码表示规则集；② Complexity rule set，复杂性规则集；③ Control Flow rule set，控制留规则集；④ Naming rule set，名字规则集；⑤ Portability rule set，移植性规则集；⑥ Resource rule set，资源规则集。

Logiscope RuleChecker 的规则集具有开放的接口。每个规则对应一个规则文件(.rl 文件)，规则文件有文件头和文件体(规则检查的处理过程)。用户可以根据需要制定自己的规则。

5.3 其它规则检查工具

Polyspace Verifier 是法国 Polyspace 公司的软件测试和规则检查工具。它可以对 C 程序和 Ada 程序进行检查，它支持 MISRA C/Ada 程序设计标准。

QA C 是英国 Programming Research 公司的 C 语言静态分析工具，它有利于早期开展测试和评审活动。它能够进行公司或者行业范围的程序设计规则检查，发现潜在的编码问题。QA C 还可以产生40多种为业界所接受的程序度量，包括环路复杂性、静态路径数等。

6 程序设计规则检查：进一步思考

6.1 规范语言层次的规则检查

程序设计规则检查体现的是一种思想，这种思想应该可以应用到规范语言层次，包括形式化规范、半形式化规范和自然语言规范。

在形式化规范和半形式化规范(如 UML 规范)方面，可以定义类似的规则。在 UML statechart 中，对于分支伪状态处的复合迁移，我们应该详细检查分支条件是否有遗漏，缺省情况下是否需要对应某个迁移。如果不包含缺省条件，而且已有的各条件不是对所有情况的一个划分，那么有些复合迁移就会无法完成。

在自然语言描述方面，我们也可以定义类似的规则，以便用户描述软件需求和软件设计时考虑得更全面些。如，在描述形如 if-then-else 的分支判断情况时，既要指明条件值为 TRUE 的处理方法，也要指明条件值为 FALSE 的处理方法，这样才不会导致控制结构的成份残缺。这对应于 DERA C 规则中的[DERA141]和 Testbed 所特有的规则[Testbed8S]。

又如，在描述形如 switch-case 的选择结构中，既要指明各种 case 下的处理方法(避免使某个 case 的处理部分为空)，又要指明 default 情况下的处理方法。道理同上。这对应于规则 [Testbed25S]、[MISRA62]、[MISRA64]。还要注意，选择关系如果是多值选择的话，才值得用 switch 选择结构；如果只是布尔选择的话，那么还是用 if-then-else 的分支判断结构比较适当。这对应于 MISRA C 规则中的[MISRA63]。

对于控制结构中的边界情况和可能的特殊情况，应该引起足够的重视。这一点对于容错设计尤为重要。有时这种特殊情况的处理会相当耗费精力。

6.2 面向多语言的规则库的构造与扩充

规则应该以库的形式存在，而不是直接做到规则检查工具中。规则库可以以文件或者数据库的形式存放。规则与规则检查工具分离的好处是便于集中、统一地管理规则。考虑到以后可能修改规则和添加新的规则，规则库应该提供标准的接口。在制定规则的格式时，可以参考 Logiscope RuleChecker 工具。该工具的每一条规则对应一个规则文件(.rl 文件)，规则文件有文件头(包括规则名、规则注释等)以及规则检查的代码部分。

规则库应该以多视角的形式，有条理地组织起来。多视角便于用户检索。常用的视角可以包括程序分析视角、程序语言视角、代码属性视角等。程序分析视角包括控制结构分析、静态数据流分析、复杂性分析等。程序语言视角从程序语言各角度对规则进行归类，如变量、表达式、控制结构、数据类型、函数的调用、返回与副作用等。代码属性角度包括程序可读性、可靠性、可移植性、安全性、高效性等。规则库还应该能够以可视的方式显式，这样便于用户查看与管理。

规则库应该是面向多语言的，和具体程序语言无关的那些公共规则可以组织成公共库，各种程序语言相关的规则可以组织到各语言的特定规则库中。这样的好处是规则库可以广为使用，便于各种平台下的开发人员分享编码经验，共同养成良好的编码习惯。

6.3 编译器插件形式的规则检查工具开发

近年来编译技术已经有了很大发展。相对于90年代的 C 和 C++编译器(Turbo C/C++)而言，如今的版本功能更强大，

(下转第170页)

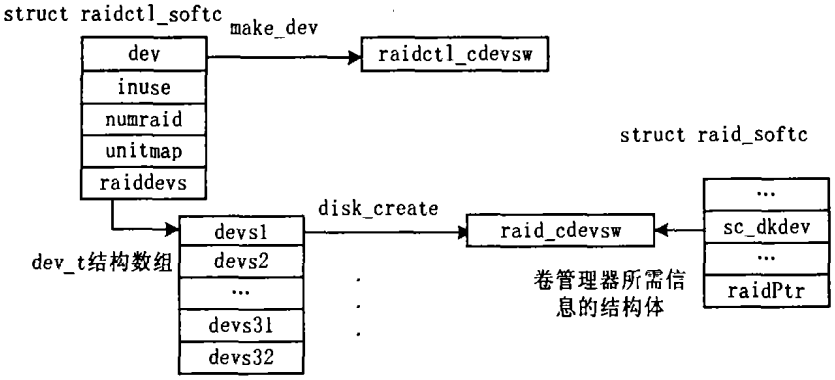

图6 卷管理器内核控制结构关系图

我们的目标是提供IDE磁盘的热插拔功能，实时支持软件RAID的出错控制和在线恢复机制。首先，采用自制转接口，使IDE在物理电气特性上支持动态插拔；在三次I/O失败后，我们认为该磁盘出错，不再可写或是被拔出；同时，给出了用户控制接口以支持磁盘在线插入功能。我们是通过两条途径操作IDE磁盘的，一是通过专门的ATA控制接口，在设备驱动一层上控制；另一个是通过FreeBSD提供的"/dev/io"设备名获得外设的I/O特权，对IDE的寄存器[8]直接访问。

添加ATA控制接口的目的是为了便于用户层和内核之间的相互通信，内核部分是利用字符设备开关表(ata_cdevsw)，在这里，我们仅需要实现ioctl的功能。通过ioctl，可以检测IDE磁盘的拔出、插入、系统ATA通道的状况，以及访问ATA状态寄存器和交换状态寄存器的基址等功能，在用户层，提供一个命令(atacontrol)，便于用户控制。

当插入一个新盘时，内核利用ata_probe()和ata_attach()可以让系统认知该盘；当拔出磁盘时，内核利用ata_reinit()重新初始化该ATA通道，则系统得知该盘已被拔出。提供ATA通道的寄存器的控制是为了方便热插拔的守护进程的轮讯，也是为了实现硬件无关性。因为不同的硬件设备提供的ATA通道的基址不是完全一致的，实时从内核中获得是最为可靠的，也是无需考虑硬件的。

结束语 本文对DAGs模型的出错控制机制和在线恢复机制提出改进，并在FreeBSD操作系统上实现了NAS系统的卷管理器。

NAS系统中的软件RAID技术还有许多有待完善或值得继续探讨的理论和应用问题。譬如：作为高端NAS服务器，应该在RAID的基础上提供文件系统快照功能以及双机热备份功能，以实现其高可靠性。这也是我们下一步的研究方向。

参考文献

1 Courtright W V II, Gibson G, Holland M, Zelenka J. A Structured Approach to Redundant Disk Array Implementation. CMU-CS-96-137 10 June 1996

2 Chen P M, Lee E K, Gibson G A, et al. RAID: High-Performance, Reliable Secondary Storage. ACM Computing Surveys, 1994, 26(2): 145～185

3 Guruswami V, Sudan M. Improved Decoding of Reed-Solomon and Algebraic-Geometric Codes. IEEE Symposium on Foundations of Computer Science, 1998

4 Holland M. On-Line Data Reconstruction In Redundant Disk Arrays. 1994

5 The FreeBSD 4.5 release source code. ftp://ftp.freebsd.org 12 July 2002

6 Writing Device Drivers. Sun Microsystems, Inc., 2000

7 Tanenbaum A S, Woodhull A S. Operating System: Design and Implementation Second Edition(影印版). 清华大学出版社, 1997. 401～503

8 Schmidt F 著, 精英科技 译. SCSI总线和IDE接口：协议、应用和编程(第二版). 中国电力出版社, 2001. 45～54

(上接第151页)

智能化程度更高。它不仅可以报告编译时的语法错误，还可以发出一些警告信息，以报告可能的用户疏漏，如变量未初始化。

规则检查工具可以以编译器插件的形式存在。它可以从编译器那里得到扫描前的源程序、词法分析结果和语法分析后的中间表示，然后进行相应的分析和规则检查，将结果以文件或者显示的形式输出。采用插件形式的好处是方便、灵活，充分利用编译器的中间运行结果。但是这需要得到编译器的接口，因此需要编译器厂商的支持。

规则检查插件最好能有供用户选择的选项。用户是否希望进行规则检查、希望进行哪些规则检查、是部分检查还是全部检查，这些都可以通过设置偏好来完成。

结束语 本文对程序设计规则检查做了介绍，列出了各种程序设计标准。并在规则检查如何保障软件质量的问题上，从正确性、可理解性、可移植性、安全性等几个角度作了剖析。然后介绍了几种著名的规则检查工具。最后对规则检查的应用领域推广、标准化、实用化等方面做了一些思考。

本文的目的主要在于引起相关人员对程序设计规则检查的重视：使程序员养成良好的编程习惯；使软件需求分析与设计人员可以以更严格、精确的方式描述、刻画需求和设计；并且倡导程序设计规则检查工具的使用。

参考文献

1 LDRA Ltd. LDRA Testbed MISRA C Checking, version 2.0. product documentation, 2000

2 Scott Meyers. Effective C ++: 50 Specific Ways to Improve Your Programs and Design. Addison-Wesley, 1992

3 Scott Meyers. More Effective C ++: 35 New Ways to Improve Your Programs and Designs. Addison-Wesley, 1995

4 MISRA. Development Guidelines for Vehicle Based Software, MIRA, CV10 0TU, 1994. http://www.misra.org.uk/

5 陈世忠. C++编码规范. 北京：人民邮电出版社, 2002