Report on the first use of rulechecker in Atlas offline software.

Solveig ALBRAND (albrand@isn.in2p3.fr) March 2003

Introduction

The IRST tool rulechecker was made available to Atlas offline developers in December 2002.
Instructions for using it under CMT can be found at
http://atlas.web.cern.ch/Atlas/GROUPS/SOFTWARE/OO/Development/qa/Tools/RuleChecker.html
About 6 users have provides feedback – but one of them represented a group of users. The comments are both concerning the use of rulechecker, for example how to control the rules which are checked, specific comments and questions concerning how rulechecker interprets some particular rules, and also some comments on the rules themselves.
It is evident that the first use of an automatic tool for rule checking brings to light some rules of which users were not necessarily aware, and in some cases calls into question the validity of the wording of the rule, or even the complete rule.
These changes should be incorporated into a new version of the Atlas C++ rules, which will also contain some corrections and some modifications suggested during our review process. For completeness, these rules are listed in the last section.
This report was first submitted to the tool's authors, Paolo Tonella and Alessandra Potrich, and their comments have been incorporated into the final version.

General comments on the use of rulechecker

Rulechecker is currently installed in the LCG area at afs/cern.ch/sw/lcg/contrib/rulechecker. Other experiments already use it, or will use it. As far as I know only Atlas is using the above installation. The version of rulechecker is dated 02/12/2002 and concerns the first milestone of the tool. 46 rules are checked.
The installation under CMT allows recursion in package source directories, treating each CXX file. Symbolic links are made for each package source directory from the package cmt directory towards a common Atlas configuration of rulechecker. For this first installation I used the default configuration, i.e. all the rules provided in the first milestone were checked. Essentially all the Atlas user has to do is to type "gmake rulechecker". No one reported any difficulty.
In version 2 of rulechecker, the configuration directory can be pointed by a new environment variable, which will simplify the make file fragments.
Some of the first users of rulechecker have been novice C++ programmers, conscious that a lot of things can go wrong in C++, and so have welcomed the idea that a tool can tell them when they should have declared something const for example. For this reason, it is very important that we try to prevent rulechecker from returning confusing messages, or from suggesting changes in the program which will lead to an undesirable result.

How do I turn off a particular rule?

The most frequent question from users is "How do I turn off a particular rule?".
To make this possible, individual users would need to install their own copy of the configuration file which lists the rules to be checked (config_RULES). A new environment variable $IRST_CONFIG_DIR, would be defined in the CMT requirements file which would point by default to the common version of config_RULES. Users wishing to override the common version would need to redefine the environment variable. Version 2 of rulechecker supports this mechanism.
Another line of attack to give the users some choice in the rules they apply would be to provide some "typical" config_RULES files; for example, all the nomenclature rules, or all the REQUIRED rules. Some users have complained that they felt overwhelmed by the large number of rule violations that their code provoked. Breaking the rules up into groups would allow a user to first correct all the nomenclature, and then move on to the group of REQUIRED rules, which do not concern

nomenclature. Hence the nomenclature violations would not hide the other violations. To implement this mechanism we could provide different "strategies" in the rulechecker make files.

How do I turn off checking a particular file?

Rulechecker has a mechanism for choosing which files are treated, but there is no mechanism for excluding files. A particular problem with Atlas offline software is that ALL packages contain two files called PackageName_load.cxx and PackageName_entries.cxx. They do not need to be checked, but I have found no way of excluding them systematically. I have tried putting the line "MODULE_NAME $_load $_entries" in the file config_FILTERING, but could not see any effect. The files are not very big and they do not take long to. I guess the effect of leaving them is to lengthen the total time by <1.5%, so perhaps these files are not worth worrying about.

The treatment of some specific rules by rulechecker

In this section I list some specific problems people have found with some of the rules currently checked by rulechecker.

Rule CO2: Use forward declaration instead of a header file, if this is sufficient

Several users have said they would like to turn off this rule. One has the impression that the rule simply lists all the include files it finds and says that they could be replaced by forward declarations. It is very confusing especially for beginners. Paolo Tonella has sent this explanation of rulechecker's behaviour.
"RuleChecker does not "simply list all the include files". It suggests using a forward declaration in a given header/implementation file when the included header file defines a class A and it is **not** the case that:

- a method of class A is implemented in the analyzed files (ex.: `A::f()`)
- the analyzed code contains one of the following situations:
  - `A *p`; and successive use of `p->` or `(*p)`
  - `A &q`; and successive use of `q`
  - `A::x`; (static method or field)
- the analyzed code contains the statement `new A()`
- the analyzed code contains `class B: A {...};`
- the value of an enumerator declared in the header file is used
- the analyzed code contains declarations of the type "`A a`"

However, due to some limitations of the analysis it performs, it cannot handle:

- - global variables defined in the header file
- - library entities (ex.: cin, cout)
- - in general, the cases where type information is missing for variables/methods

As a consequence, replacement of an inclusion with a forward declaration is
only suggested, but may be not feasible."

I propose to keep the rule in the list of checked rules, and publicize this explanation. If users continue to find this rule is reporting more "false" errors than real ones, we will remove the rule from the standard list to be checked.

Rule CO4: Implementation files must hold the member function definitions for a single class (or embedded or very tightly coupled classes) as defined in the corresponding header file.

In Appendix 1 is the contents of a header file and a cxx file for the class CaloID_Exception

This class when passed through rulechecker raises the following error.

```
RULE CO4 violated:

@ Implementation files must hold the member function definitions for a
@ single class (or embedded or very tightly coupled classes) as
@ defined in the corresponding header file (REQUIRED)


--> the method:
       CaloID_Exception::operator std:: string
    refers to the class:
       CaloID_Exception that is not defined here
```

Paolo Tonella says that this is a rulechecker problem and it will be fixed.

Rule CI9 : Do not let const member functions change the state of the program

In Appendix 2 is a function from the class CaloLVL1_ID. Although one can see a lot of rule violations in this class, in particular for the variable naming, neither the author nor I can see what provoked the following output from rulechecker. I almost wondered if it were not the fact of declaring the method const which provokes the error, but I tried putting all the body of the method into comment, and no longer got the error. This implies that there is something in the source which provokes the error. As rulechecker knows what it is, could not the line number which provokes the error be given. A better indication of the line would be a great help.

```
RULE CI9 violated:

@ Do not let const member functions change the state of the program
@ (RECOMMENDED)


--> the method:
       tower_id(int, int, int, int, int): Identifier
[../src/CaloLVL1_ID.cxx 74 ]
    can change the state of program
```

Paolo Tonella's comments:
"Some invoked methods could possibly change the state of the object. In such a case, the rule would be violated: RuleChecker conservatively signals a *possible* state change.
The list of method invocations that might change the state will be given, together with the respective line numbers. If none of them actually changes the state, the violation can be safely ignored."

Applying a rulechecker recommendation causes a core dump !

The rule in question is CI8 : (*Declare as const a member function that does not affect the state of the object.*) The user (a relative beginner) had received the following message

```
--------------------------------------------------
RULE CI8 violated:
@ Declare as const a member function that does not affect
@ the state of the object (REQUIRED)
--> the method:
 finalize(): StatusCode
 in file ../src/TrigLArAlgExample.cxx [97 ]
 can be declared const
```

```
---------------------------------------------
```

When he did it, his code apparently compiled and built, but caused a core dump when running.

```
pure virtual method called
zsh: 26116 abort (core dumped)   TrigAlgExample_readZEBRA.txt
```

In Atlas oflline software all algorithms are derived from a GAUDI framework interface class. The class contains three pure virtual methods that must be implemented by a derived algorithm.

```
StatusCode initialize();
StatusCode execute();
StatusCode finalize();
```

They are not const. In declaring them const the user had no longer provided an implementation of the pure virtual method so the program could not execute. This is obviously dangerous and confusing to inexperienced users. Rule CI8 is a good rule, and rulechecker was right in detecting that the method in itself could have been declared as constant, but the tool could not know about the inheritance of pure virtual methods. Probably we should do several things to avoid this happening to other users:

- The wording of the error message could be changed to be less assertive. Instead of "..`can be declared const`" "`consider whether the method could be declared const`".
- The three methods initialize(), execute(), and finalize(), which will appear in every Atlas developers algorithms, should be listed in the default config_FILTERING file as methods to exclude.
- Any other Gaudi methods in this case should also be filtered out. (It may also be necessary to filter out some StoreGate methods, as this is another core package used by most developers)

Paolo Tonella's comments:
The following solution to this problem will be implemented in RuleChecker:
- if a member function, that does not affect the state of the object, overrides a non-const member function inherited from some superclass, the rule does not apply.
In this way it will be not necessary to filter specific cases.

Some comments on particular Atlas C++ coding rules.

These are cases where the interpretation of rulechecker is not a problem, but where the use of the tool has pointed us to some changes that should be made to our rules.

The comments rule (SC1) needs changing

We seem to have had a modest success in establishing the use of the Doxygen tool for documenting our code packages. Moreover, a growing number of people are using tools such as Together for code development. Together produces javadoc style comments, which are perfectly understood by Doxygem. It is unproductive and does not encourage the use of rulechecker if we expect developers to go through their code changing all the java doc style comments to the other acceptable Doxygen form, "///", which satisfies rule SC1.
A new version of the rule will be formulated, and rulechecker will be adapted accordingly.

Naming Conventions – rules NC1, NC2, NC6, NC7

Many users were caught out by the application of the rule NC6 to rules NC1 and NC2. For example a private variable should be called "m_name", and not "m_Name". The phrase at the end of NC6 should be copied, or moved to rules NC1 and NC2. Rulechecker is detecting this violation perfectly correctly.
The most common naming violation is the use of underscores to separate words, and not following the recommendation of the rule NC7. Our policy is to accept this in old code, if the usage is consistent, but to insist on new code following NC7. This is a task for human code reviewers.

Naming Conventions – rule NC5

Rule NC5, (*start class names, typedefs and enum types with an uppercase letter*) need some refinement. We need to state whether all of these entities MUST be named, is it acceptable to have an unnamed enum for example?
If we decide to allow unnamed enums, then the name that the preprocessor gives to them, which does not obey rule NC5, must be filtered out from rulechecking, because the error message is extremely confusing to novice programmers. This can be easily done.
If we decide that all enums must be named, that rulechecker could easily be adapted.

Const Correctness – rule CI7.

Rule CI7, (*In a class method do not return pointers or non const references to private data members*).
An exception to this rule seems to be the use of the Singleton pattern. The example of a Singleton pattern given by Scott Meyers, in "Effective C++" shows a non const reference to a static object being returned, whereas the example in "Design Patterns" (Gamma et al.) shows a pointer to a static object returned. We need to rephrase this rule to allow this exception. There is no easy way for rulechecker to detect a singleton.

Portability – rule CP10.

Rule CP10, (*Use long (not int) and double (not float)*).
This rule turns out to be particularly annoying, because of course rulechecker signals a violation for every for loop used in the program, and this was almost certainly not the intention of the authors of the rule. So, either the rule needs to be refined, to exclude for loops from detection (and rulechecker could be adapted), or perhaps dropped altogether. If the rule is not dropped, we should add some clear explanation as to why long and double are more portable than int and float.

Other Atlas C++ coding rules which need clarification.

Organizing the Code

A new rule will be introduced giving the preferred ordering of include files.

Rule CL5

Minor corrections in the text needed to bring it into line with the example.

Rule CH3 on Exception Handling

Will be re-written / expanded to try to make the explanation more didactic.

Rule CA2 on iostream functions

Needs some examples of how to replace scanf and printf.
We also need a new rule requiring the use of Gaudi message streams.

Rule CR3 on Typedefs

This rule should perhaps become a little stricter, and the explanation made clearer.

Rule CT3 on templates

This rule needs to be rewritten to emphasize that a test program MUST be written to instantiate templates. The necessity of this was brought to light by a mistake which the "codewizard" tool found in one of the Gaudi templates.

Rule SG7 on the use of spaces in the source code.

This rule will be amended so that one can use spaces before "()" in if, when and switch clauses.

Appendix 1

CaloID_Exception.h

```
/* date of creation : 10/X/2002 */

/* date of last modification : 10/X/2002 */

#ifndef CALOID_EXCEPTION_H
#define CALOID_EXCEPTION_H

#include <string>

/**
 * Exception class for Calo Identifiers <br>
 * @author Johann Collot , Fabienne Ledroit (cloning  LArID_Exception)
 * @version 00-00-00
 * @since 00-00-00
 */

class CaloID_Exception {
public:

    /**
     * default constructor
     */
  CaloID_Exception() ;

    /**
     * constructor to be used
     */
    CaloID_Exception(std::string  lMessage , int lCode) ;

    /**
     * set error message
     */
    void message(std::string lMessage) ;

    /**
     * return error message <br>
     */
    virtual std::string message() const ;

    /**
     * set error code number<br>
     */
    void code(int lCode) ;

    /**
     * return error code <br><br>
     *
     * error codes : <br>
     * 0 : no error <br>
     * 1 : CaloLVL1_ID::region_id Error <br>
     * 2 : CaloLVL1_ID::channel_id Error <br>
     * 999 : undefined error <br>
     */
    virtual int code() const ;

    virtual operator std::string();

    /**
```

```
     * destructor
     */
    virtual ~CaloID_Exception() ;

private:

    /**
     * error message
     */
    std::string m_message;

    /**
     * error code
     */
    int m_code;
};


#endif //CALOID_EXCEPTION_H
```

CaloID_Exception.cxx

```
#include "CaloIdentifier/CaloID_Exception.h"

#include <stdio.h>

CaloID_Exception::CaloID_Exception() :
m_message("No error message") , m_code( 999 )
{  }

CaloID_Exception::CaloID_Exception(std::string  lMessage , int lCode) :
m_message ( lMessage ) , m_code ( lCode )
{ }

void CaloID_Exception::message(std::string lMessage)
{ m_message = lMessage ;}

std::string CaloID_Exception::message() const
{ return m_message;}

void CaloID_Exception::code(int lCode)
{ m_code = lCode ;}

int CaloID_Exception::code() const
{ return m_code;}

CaloID_Exception::operator std::string ()

{

  char * l_str = new char[200];
  std::string newline(" \n ") ;
  std::string errorMessage ;
  sprintf(l_str,"CaloID_Exception - Error code: %d ", this->code());
  errorMessage += std::string(l_str);
  errorMessage += newline ;
  errorMessage += this->message() ;
  delete[] l_str ;
  return errorMessage ;
}

CaloID_Exception:: ~CaloID_Exception() {}
```

Appendix 2

```
Identifier CaloLVL1_ID::tower_id   ( int pos_neg_z, int sampling, int
region,
                         int eta,       int phi ) const
throw(CaloID_Exception)
{

    if( ! values_ok (pos_neg_z, sampling, region, eta, phi) ) {

      char * l_str = new char[200];
      std::string errorMessage ;
      sprintf(l_str,"Error in CaloLVL1_ID::tower_id(field values) ,
pos_neg_z: %d , sampling: %d, region: %d , eta: %d , phi: %d ", pos_neg_z ,
sampling , region, eta, phi);
      errorMessage += std::string(l_str);
      delete[] l_str ;
      CaloID_Exception except(errorMessage , 2);
      throw except ;

    }

    // Build identifier
    Identifier result = region_id (pos_neg_z, sampling, region);
    result << eta << phi;

    if( result.last_error () != Identifier::none) {

      char * l_str = new char[200];
      std::string errorMessage ;
      sprintf(l_str,"Error in CaloLVL1_ID::tower_id(field values) , values
ok but did not build , pos_neg_z: %d , sampling: %d, region: %d , eta: %d ,
phi: %d ", pos_neg_z , sampling , region, eta, phi);
      errorMessage += std::string(l_str);
      delete[] l_str ;
      CaloID_Exception except(errorMessage , 2);
      throw except ;

    }

    return result;
}
```