Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Exemplarisches Eclipse-Plugin zur statischen C-Syntaxanalyse am Beispiel von Misra-C Regeln

vorgelegt von
Guido Kaiser *(1765261)*
am ***01.10.2007***

Betreuer: Prof. Dr. Bettina Buth
Zweitgutachter: Prof. Dr. Olaf Zukunft

Studiengang: Bachelor of Computer Science (INB)

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Guido Kaiser

Exemplarisches Eclipse-Plugin zur statischen C-Syntaxanalyse am Beispiel von Misra-C Regeln

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Informatik Bachelor (INB)
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Bettina Buth
Zweitgutachter: Prof. Dr. Olaf Zukunft

Abgegeben am **01.10.2007**

Guido Kaiser

Thema der Bachelorarbeit

Exemplarisches Eclipse-Plugin zur statischen C-Syntaxanalyse am Beispiel von Misra-C Regeln

Stichworte

Statische C-Code Analyse, Kompiliererbau, Misra-C, Programmiersprache C, Eclipse, Eclipse-Plugin Programmierung

Kurzzusammenfassung

Im Rahmen dieser Bachelorarbeit werden verschiedene Programme zur statischen C-Code Analyse betrachtet. Das Regelwerk Misra-C:2004 dient hierbei als Grundlage. Es werden sowohl Open Source, als auch kommerzielle Programme auf vollständige Abdeckung, Erweiterbarkeit und Integrierbarkeit (in die Entwicklungsumgebung Eclipse) geprüft.

Des Weiteren wird an Hand einer exemplarischen Realisierung demonstriert, wie mit Hilfe eines Eclipse-Plugins eine regelbasierte Syntaxanalyse implementiert werden kann.

Guido Kaiser

Title of the paper

Example Eclipse-Plugin for static C-Code-Analysis with rules of Misra-C

Keywords

Static C-Code analysis, Building a compiler, Misra-C, Language C, Eclipse, building an Eclipse-Plugin

Abstract

This thesis discusses several tools for static C-Code analysis. The rules of the Misra-C:2004 will be used as base. Open Source and commercial tools will be checked for extensibility, rule coverage and Integration (especially for Eclipse).

Implementing an example Eclipse-Plugin will show how to make rule-orientated static Code-Analysis.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1) Einleitung

Ein möglichst fehlerfreier Code spielt eine immer größere Rolle, da die Anwendungen immer komplexer werden und das Auffinden von potentiellen Fehlern immer schwerer wird. Ebenso leidet die Wartbarkeit am sogenannten schlechten Code. Indizien für schlechten Code sind unter anderem:

- wenig oder gar nicht kommentierter Code,
- aufgrund der Formatierung schlecht lesbarer Code,
- nicht klar ersichtliche Namen für Variablen und Objekte (Stichwort: Sprechende Bezeichner).

Um diesen Problemen vorzubeugen, kann der Programmcode mit Hilfe der statischen Syntaxanalyse überprüft und mögliche Schwachstellen gefunden werden. Eine statische Syntaxanalyse kann aber nicht alle möglichen Fehlerquellen finden. Fehler, die erst zur Laufzeit auftreten, können nicht erkannt werden (z.B. eine Division durch Null).

Fehlerfreier Code bedeutet, dass potentielle Fehler oder Fehler, die beim Kompilierungsvorgang nicht gefunden wurden, weil sie syntaktisch richtig sind (sogenannte logische Fehler), möglichst gering ausfallen. Solche Fehler sind nur durch statische Syntaxanalysen oder manuelle Code-Reviews[1] zu finden.

Es gibt eine Vielzahl von Programmen, die solche Überprüfungen durchführen können. Des Weiteren gibt es viele Regelvorschläge für „guten Code“. Guter Code heißt, dass die Anwendung folgende Aspekte möglichst gut abdeckt:

- **Verständlichkeit und Wartbarkeit**
  Vollständige und verständliche Dokumentation des Codes. Es sollten keine kryptischen Code-Konstrukte (C-Beispiel – kryptische if-else-Anweisung: `x ? a=1; : a=2;`) verwendet werden und der Code sollte sinnvolle und aussagekräftige Kommentare besitzen.

[1] Code-Review: Das Dokument nochmals betrachten

- **Erweiterbarkeit**
  Erreicht wird die Erweiterbarkeit durch gut und sauber definierte Schnittstellen, die auf jeden Fall eingehalten werden müssen. Des Weiteren muss der Code gut strukturiert, weitestgehend modular gehalten und ausführlich dokumentiert sein.
- **Zuverlässigkeit**
  Anwendungen, die ausreichend getestet wurden und sich statischen Syntaxanalysen unterzogen haben, um möglichst viele potentielle Fehler zu finden.

Die wohl bekannteste Regel lautet: „**k**eep **i**t **s**imple and **s**tupid“ (KISS). Es sollte immer einen Codingstandard bzw. Coding-Rules[2] geben, der eine Ansammlung von Regeln beinhaltet, auf die sich die am Projekt beteiligten Personen geeinigt haben, um die vorher genannten Aspekte für guten Code möglichst weiträumig abzudecken.

Erfahrungsgemäß zeigt sich in vielen Unternehmen, dass die Benutzung von Coding-Rules zwar die erste Implementierung oft etwas verlängert, aber dafür wird die Einarbeitungszeit von Programmierern, denen der Code fremd ist, und die Weiterentwicklungszeit der Software verkürzt. Dies liegt hauptsächlich an der Tatsache, dass der Code besser strukturiert, lesbarer und ausführlicher dokumentiert ist. Des Weiteren wird unter Umständen die Fehlerbehebungs- (Bug-Fixing-) Zeit verkürzt, weil weniger logische Fehler gefunden werden müssen.

Bereits existierende Sammlungen von Regeln, auch Regelwerke genannt, gibt es viele verschiedene. Oft besitzen Software-Firmen ihre eigene Sammlung, oder auch mehrere, wenn sie nur für bestimmte Projekte gültig sind. Umfangreichere Sammlungen gibt es z.B. von Todd Hoff [18] oder von der Firma Macadamian [19]. In dieser Arbeit wurde das Regelwerk Misra-C:2004 [8] gewählt, weil es ausgereift und sehr umfangreich ist. Es werden viele Aspekte bzw. Problemkategorien der Programmiersprache C abgedeckt. Siehe hierfür Kapitel **2.1** bis **2.2**.

[2] Zusammenfassung von Regeln, die in einem Projekt benutzt werden

Diese Arbeit untersucht anhand von Spezifikationen, Bildschirmfotos und Beschreibungen der Hersteller die Nutzbarkeit und die Möglichkeiten von kommerziellen und freien Programmen zur statischen C-Syntaxanalyse am Beispiel vom Misra-C:2004 Regelwerk. Die Regeln sorgen für besseren C-Code, der weniger fehleranfällig und besser wartbar ist. Sie umfassen Richtlinien für Programmierstil, Umgang mit kritischen Elementen der Programmiersprache, Ausschlüsse von Funktionen etc. Die Regeln sind allerdings keine Garantie für fehlerfreien Code, aber die Qualität der Software wird deutlich verbessert.

Für einige deutsche Automobilkonzerne gilt, dass die Misra-C:2004 Regeln strikt einzuhalten sind, d.h. dass generell alle Regeln eingehalten werden müssen. Zu den deutschen Autokonzernen, die Misra-C:2004 nutzen, gehören unter anderem: Audi, BMW, Daimler Chrysler und Porsche[3].

Folgende Auflistung zeigt die Qualitätsmerkmale von Software gemäß **DIN ISO 9126**[4] und die jeweilige Abdeckung durch das Misra-C:2004 Regelwerk:

- Funktionalität (nur auf Seiteneffekte von Fehlern bezogen)
- Übertragbarkeit (im Rahmen der Möglichkeiten der statischen Syntaxanalyse)
- Zuverlässigkeit (im Rahmen der Möglichkeiten der statischen Syntaxanalyse)
- Änderbarkeit (bezogen auf Lesbarkeit des Codes, Dokumentation)
- Benutzbarkeit (nicht abgedeckt)
- Effizienz (nicht abgedeckt)

Abschließendes Ziel dieser Arbeit ist es, ein Tool zu entwickeln, das in der Lage ist C-Coding-Rules zu verwenden, Open Source ist und sich in Eclipse integrieren lässt. Im Rahmen dieser Arbeit wird das Tool sich an den Regeln des Misra-C:2004 Regelwerks orientieren. Aufgrund der Erweiterbarkeit durch neue Regeln können später auch andere Regelwerke mit einbezogen werden. Das Plugin soll die positiven Eigenschaften der vorhandenen Programme, in einem weitestgehend eigenständigen Tool, vereinen.

[3] Dateianlage: „20070917 - HIS_SubSet_MISRA_C_2.0.pdf“
[4] Ein Modell für die Softwarequalität in Bezug auf Produktqualität

1.1) Grundbegriffe

Dieser Abschnitt beinhaltet eine Kurzeinführung in Fachbegriffe, die für das Verständnis dieser Arbeit wichtig sind. Die Definitionen stammen zum Teil aus der freien Enzyklopädie Wikipedia[5] und aus dem Wissensschatz des Autors dieser Arbeit. Daher wird auf weitere Quellenangaben verzichtet. Ausführlichere Beschreibungen können bei Wikipedia abgerufen werden.

GPL

Die **G**eneral **P**ublic **L**icence (GPL) ist eine Lizenz für Open Source Software. Sie beinhaltet die kostenlose Nutzung, Veränderung und Verbreitung von Software, solange andere (z.B. die Urheber) dieselben Freiheiten für die neu entwickelte Software haben.

Statische Syntaxanalyse

Die statische Syntaxanalyse überprüft Code auf Verstöße gegen Programmierkonventionen und mögliche Fehlerquellen, die zur Kompilierungszeit entdeckt werden können. Fehler, die erst zur Laufzeit auftreten, können nicht erkannt werden.

Es können auch Fehler gefunden werden, die eigentlich gar keine sind. Solche Situationen können durch Einhaltung von mehr oder weniger umfangreichen Programmierstandards weitestgehend ausgeschlossen werden.

Eingebettete Systeme

Eingebettete Systeme sind Computer, die in Geräte eingebettet sind, so dass sie für den Benutzer weitestgehend unsichtbar sind. Zum Beispiel finden sich solche Systeme in Waschmaschinen, Backöfen, Mikrowellen etc.

Garbage-Collection

Automatische Löschung von nicht mehr benutzten Objekten. Auf diese Weise wird regelmäßig der Speicher von nicht mehr verwendeten Objekten „aufgeräumt“. Moderne Programmiersprachen besitzen eine Garbage-Collection, um regelmäßig für freien Speicher zu sorgen.

[5] http://www.wikipedia.de

RPC	Ein **R**emote **P**rocedure **C**all (RPC) dient der Interprozesskommunikation. Er ermöglicht das Aufrufen von Funktionen in einem Dienst / einer Anwendung, die auf demselben oder auch einem anderen Rechner im selben Netzwerk liegen. Andere bekannte Dienste, die auf diesem Prinzip basieren sind z.B. CORBA und Java-RMI.
Wrapper-Klasse	Eine Wrapper-Klasse dient der Übersetzung zwischen zwei Schnittstellen. Sie entsprechen dem Adapter-Entwurfsmuster aus dem Software-Engineering. Eingesetzt werden Wrapper-Klassen meistens, wenn zwei Module miteinander kommunizieren sollen, die unterschiedliche Schnittstellen bereitstellen.
V-Modell	Das V-Modell beschreibt einen möglichen Entwicklungsprozess eines Projekts. Hierbei werden Testphasen während des gesamten Prozesses zwingend eingesetzt, um die jeweiligen Schritte zu validieren. Da das V-Modell die Möglichkeit bietet vorherige Phasen auf korrekte Umsetzungen der Spezifikationen zu prüfen, können einzelne Phasen verifiziert werden.
BBCDT	BBCDT ist ein freies Plugin der Firma IBM. Das Plugin soll als Beispiel für die Implementierung eines eigenen Editors mit C-Syntax-Hervorhebung unter Eclipse dienen.
OSGI	Die Spezifikation der **O**pen **S**ervice **G**ateway **I**nitiative (OSGi) beschreibt eine hardwareunabhängige dynamische Softwareplattform. Dienste und Anwendungen werden dort nach dem **SOA**-Paradigma ausgeführt.
SOA	**S**ervice **O**riented **A**rchitecture beinhaltet, dass Dienste nachträglich (zur Laufzeit) installiert und ausgeführt werden können.

1.2) Überblick

Das Kapitel **2** dient der Einführung in die Bereiche und Programme, die im Laufe dieser Arbeit besprochen und benutzt werden.

Im ersten Teil der Arbeit werden vorhandene freie und kommerzielle Programme untersucht, die in der Lage sind, Regeln der Misra-C:2004 und andere sicherheitsrelevante Regeln im Code zu überprüfen. Es wird auf Integrierbarkeit in die Entwicklungsumgebung Eclipse, Konfigurierbarkeit vorhandener Regeln und die Erweiterbarkeit durch neue Regeln geachtet (siehe hierzu Kapitel **3**).

Der nächste Schritt ist die Untersuchung der GCC in Kapitel **4** in Bezug auf Anwendung von Misra-C:2004 Regeln auf Seiten des Kompilierers. Es wird geprüft, ob und welche Regeln nach dem vorhandenen Standard von der GCC durchgeführt werden können.

In Kapitel **5** werden klare Anforderungen, die an das zu implementierende Eclipse-Plugin zu stellen sind, definiert. Des Weiteren werden Ziele für die Implementierung im Rahmen dieser Arbeit festgelegt.

Im Anschluss daran wird ein exemplarisches Eclipse-Plugin entwickelt, das in der Lage ist, diese Regeln aus der Eclipse Oberfläche heraus zu überprüfen. Umgesetzt werden nur drei Regeln, die die Vorgehensweise demonstrieren sollen. Die Implementierung und Problemlösungen sind beschrieben in Kapitel **6**.

Zum Abschluss wird in Kapitel **7** ein Fazit über das Eclipse-Plugin in Bezug auf die vorher untersuchten Tools gezogen und mögliche Verbesserungen und Erweiterungen angesprochen.

2) Grundlagen

Nachdem eine kurze Übersicht über diese Arbeit gegeben wurde und die wichtigsten Begriffe erklärt wurden, kann tiefer in das Thema eingestiegen werden. Hierfür müssen einige Bereiche näher erläutert werden. In diesem Kapitel werden die Programmiersprache C, das Regelwerk Misra-C:2004, die Entwicklungsumgebung Eclipse und die GCC genauer erläutert.

Die Übersicht bezieht sich auf Herkunft, Entwicklung und Eigenheiten der genannten Bereiche. Des Weiteren werden hierfür Einsatzgebiete beschrieben.

2.1) Programmiersprache C

Die Programmiersprache C ist eine imperative Hochsprache, die 1972 von Dennis Ritchie und Brian W. Kernighan entwickelt wurde [12]. Auf fast jeder Plattform befindet sich, in diversen Dialekten, ein C-Kompilierer. Für Unix / Linux Plattformen ist der bekannteste Kompilierer die GCC, siehe Kapitel **2.4**. Für die Plattform Windows existieren deutlich mehr Kompilierer. Die bekanntesten sind hier die Kompilierer von Microsoft (MS Visual-C++) und von Inprise/Borland (C++ Builder). Nicht zu vergessen die GCC Portierung MinGW, die immer mehr Beliebtheit in der Windows Welt findet, aufgrund der Konformität zur GCC.

Als die ersten objektorientierten Sprachen entstanden, wurde C erweitert, um auch mit der objektorientierten Programmierung mithalten zu können. Als Ergebnis kam 1985 die Programmiersprache C++ heraus [13]. Es ist möglich C und C++ in einer Anwendung zu mischen, zu empfehlen ist dies jedoch nicht. Ein wichtiger Grund, weshalb ein C++ Programm keine C-Fragmente enthalten sollte, ist dass in C++ viele Verbesserungen eingeflossen sind, die in C noch nicht enthalten waren. So steigt alleine durch die Verwendung von C in C++ Code die Fehlerwahrscheinlichkeit.

Die Tatsache, dass die beiden Sprachen sich mischen lassen, ist ein wichtiger Kritikpunkt, da die Objektorientierung strikt eingehalten werden sollte. Grundsätzlich gilt, dass objektorientierter Code, der mit nicht objektorientiertem Code gemischt wird schlechter **wartbar**, weniger **erweiterbar** und schwerer **verständlich** ist.

Leider gibt es Gebiete, auf denen sich das Mischen von C und C++ nicht ganz vermeiden lässt. Zum Beispiel basiert RPC nur auf C, so dass Wrapper-Klassen erstellt werden müssen, die eine klare Grenze zwischen dem eigenen objektorientierten Code und dem reinen C-Code des RPCs ziehen. Ursprünglich wurde die Mischung nur ermöglicht, damit Unternehmen mit großen Anwendungen relativ einfach Schritt für Schritt umsteigen können, ohne gleich eine komplette Neuentwicklung der Software durchführen zu müssen.

Microsoft veröffentlichte im Jahre 2000 C# [14], das aus C++, Java, Visual Basic und Delphi Komponenten bestand. Durch die sprachunabhängigen .NET-Bibliotheken [15] ist es möglich Module aus verschiedenen Sprachen zu mischen.

Im Jahre 2007 erschien dann die Programmiersprache D [16], die annähernd die Durchsatzgeschwindigkeit[6] der Programmiersprache C haben soll und zusätzlich die Vorteile von JAVA und C# vereint. Unter anderem ist hier eine Garbage-Collection eingeführt worden.

C sollte eine „besser leserliche“ Programmiersprache gegenüber Assembler werden, um Programmierern die Entwicklung von Software zu erleichtern. Aufgrund der Nähe zur Systemprogrammierung (Maschinencode) und der direkten Speichernutzung über Zeiger, ist C sehr schnell und wird daher auch bevorzugt für zeitkritische Anwendungen, wie z.B. Spiele, eingesetzt.

Für eingebettete Systeme gibt es oft keine andere Möglichkeit als Programme in C oder Assembler zu schreiben, da die Mikroprozessoren und deren Kompilierer keine anderen Sprachen unterstützen. Des Weiteren können C-Programme viel kleiner und weniger speicherintensiv als bei anderen Sprachen sein. Für eingebettete Systeme ist der Speicher oft sehr knapp und die Performanz muss möglichst hoch sein. Die Kompilierer für die Prozessoren der eingebetteten Systeme optimieren C-Code meistens auf den Befehlssatz der jeweiligen Prozessoren.

[6] Hier in Bezug auf Geschwindigkeit und Optimierung der erzeugten Maschinenbefehle

Aufgrund der Tatsache, dass C dem Programmierer sehr viele Möglichkeiten bietet direkt die Hardware anzusprechen, z.B. der direkte Zugriff auf beliebige Speicherbereiche, ist die Fehler- und Manipulationsgefahr sehr hoch. Dadurch kann bewusst und / oder unbewusst viel manipuliert werden, was sehr starke Auswirkungen auf das System haben kann.

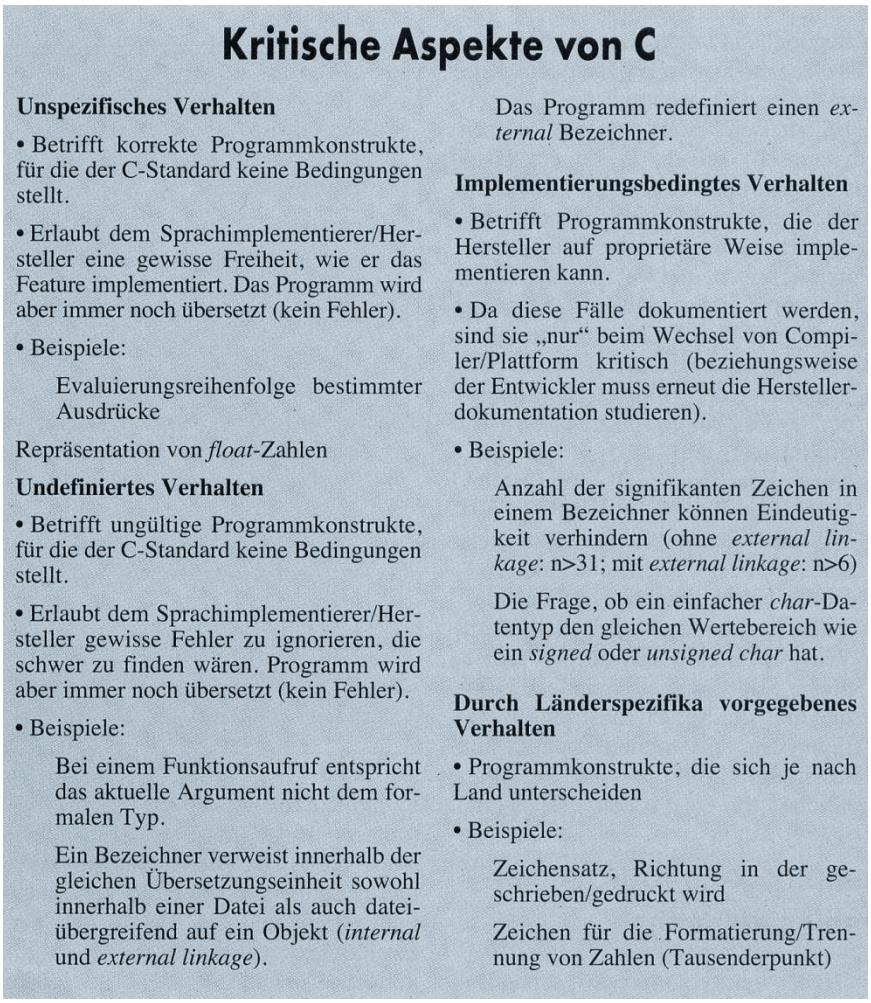
Zum Beispiel können beliebige Speicheradressen, die zum System gehören, direkt angesprochen und verändert werden. Heutzutage versuchen Betriebssysteme durch virtuelle Speicheradressen für Programme oder ähnliche Konzepte, solche Fehlerquellen bzw. Manipulationen zu vermeiden. Des Weiteren kann durch eine unvorsichtige Benutzung der Zeigerarithmetik ein ungewünschter Speicherbereich überschrieben werden. Im folgenden Beispiel wird ein Speicherbereich überschrieben, der unmittelbar hinter dem Array liegt, welches verändert werden sollte:

```
int temp[2];              // Array mit 3 Elementen
*( temp + 3 ) = 1234;     // Der maximal erlaubte Zugriff wäre temp + 2
```

Aus den oben angeführten Gründen gibt es eine große Anzahl an Regeln und Möglichkeiten, um diese Fehler bzw. Manipulationen einzuschränken, damit sie so selten wie möglich auftreten. Eine Anwendung hierfür sind z.B. Coding-Rules.

Die kritischen Aspekte kann man grob in vier Kategorien einteilen. Siehe hierzu auch **Abbildung 1 - Kritische Aspekte von C [9]**.

Die vier unten angesprochenen kritischen Aspekte / Kategorien der Programmiersprache C dienen als grobe Richtlinie, um Regeln und Probleme im Code einzuordnen. Wie der Misra-C:2004 Standard mit diesen Kategorien umgeht, zeigt **Tabelle 2 - Misra-C:2004 Regeln, die die vier kritischen Kategorien von C abdecken**. Im Folgenden wird näher auf Misra und den Regelstandard Misra-C:2004 eingegangen.



Kritische Aspekte von C

Unspezifisches Verhalten

• Betrifft korrekte Programmkonstrukte, für die der C-Standard keine Bedingungen stellt.

• Erlaubt dem Sprachimplementierer/Hersteller eine gewisse Freiheit, wie er das Feature implementiert. Das Programm wird aber immer noch übersetzt (kein Fehler).

• Beispiele:

Evaluierungsreihenfolge bestimmter Ausdrücke

Repräsentation von *float*-Zahlen

Undefiniertes Verhalten

• Betrifft ungültige Programmkonstrukte, für die der C-Standard keine Bedingungen stellt.

• Erlaubt dem Sprachimplementierer/Hersteller gewisse Fehler zu ignorieren, die schwer zu finden wären. Programm wird aber immer noch übersetzt (kein Fehler).

• Beispiele:

Bei einem Funktionsaufruf entspricht das aktuelle Argument nicht dem formalen Typ.

Ein Bezeichner verweist innerhalb der gleichen Übersetzungseinheit sowohl innerhalb einer Datei als auch dateiübergreifend auf ein Objekt (*internal* und *external linkage*).

Das Programm redefiniert einen *external* Bezeichner.

Implementierungsbedingtes Verhalten

• Betrifft Programmkonstrukte, die der Hersteller auf proprietäre Weise implementieren kann.

• Da diese Fälle dokumentiert werden, sind sie „nur" beim Wechsel von Compiler/Plattform kritisch (beziehungsweise der Entwickler muss erneut die Herstellerdokumentation studieren).

• Beispiele:

Anzahl der signifikanten Zeichen in einem Bezeichner können Eindeutigkeit verhindern (ohne *external linkage*: n>31; mit *external linkage*: n>6)

Die Frage, ob ein einfacher *char*-Datentyp den gleichen Wertebereich wie ein *signed* oder *unsigned char* hat.

Durch Länderspezifika vorgegebenes Verhalten

• Programmkonstrukte, die sich je nach Land unterscheiden

• Beispiele:

Zeichensatz, Richtung in der geschrieben/gedruckt wird

Zeichen für die Formatierung/Trennung von Zahlen (Tausenderpunkt)

Abbildung 1 - Kritische Aspekte von C [9]

2.2) Misra-C Regelwerk

Die **M**otor **I**ndustry **S**oftware **R**eliabilty **A**ssociation (MISRA)[7] hat es sich zur Aufgabe gemacht ein Regelwerk zu entwerfen, das dabei helfen soll, besseren und sichereren C-Code zu schreiben. Besonders wichtig ist dies für kritische Systeme.

Entstanden sind die Regeln durch einen Zusammenschluss von der Automotive-Branche, bestehend aus Automobil- und Bahnindustrie, dem Schiffbau sowie der Luft- und Raumfahrtindustrie. Hier ist es sehr wichtig möglichst fehler- und ausfallfreie Software zu entwickeln und einzusetzen, um menschliche und materielle Schäden zu verhindern. Eine Kostenreduzierung, sowohl für den Entwicklungsprozess, als auch für eventuell vermiedene Rückrufaktionen aufgrund fehlerhafter Software, ist möglich. Ein denkbares Beispiel wäre ein Softwarefehler in der Motorsteuerung eines Autos, der einen erhöhten Spritverbrauch zur Folge hat.

[7] http://www.misra.org.uk/

Das Ziel der MISRA: *„To provide assistance to the automotive industry in the application and creation within vehicle systems of safe and reliable software.“* [8]

Die erste Version des Regelwerks wurde 1998 veröffentlicht (Misra-C:1998). Aufgrund ungenauer Definition mancher Regeln, wurde 2004 nochmals eine überarbeite Version der Regeln veröffentlicht. Manche Regeln waren mehrdeutig, schlecht erklärt oder überflüssig. Für C++ erscheint voraussichtlich im November 2007 ein neues Regelwerk (Misra-C++).

Die Misra Regelwerke umfassen Richtlinien für Programmierstil, Umgang mit kritischen Elementen der Programmiersprache, Ausschlüsse von Funktionen etc. Enthalten sind unbedingt notwendige und optionale Regeln. Es gibt aber auch Regeln, die entweder aufgrund der Migration von altem Code oder durch bestimmte Systembeschaffenheiten (z.B. bei eingebetteten Systemen) nicht eingehalten werden können.

Für diese Fälle sieht Misra eine „Deviation Procedure“, also vorgeschriebene Behandlungen von Abweichungen, vor. Es werden zwei verschiedene Behandlungen spezifiziert. Zum einen gibt es die „Project Deviation“, bei der sich auf eine Untermenge der Regeln geeinigt wird, die eingehalten werden muss. Zum anderen gibt es die „Specific Deviation“, dort wird hingegen im Projekt gekennzeichnet, an welchen Stellen und mit welcher Begründung die Regeln nicht eingehalten wurden. In beiden Fällen schreibt Misra einen Rahmen für die Dokumentation vor. Beide Behandlungen müssen folgende Punkte dokumentieren:

- Informationen über die Ausnahme (z.B. die Regel, die ausgeschlossen wird)
- Konsequenzen, die aufgrund des Ausschließens der Regel auftreten können
- Ausreichende Begründung
- Ein Beispiel, wie trotzdem die Sicherheit gewährleistet werden kann

Die Project Deviation benötigt zusätzlich noch eine Auflistung der Fälle bzw. Situationen, in denen die definierten Untermengen eingesetzt werden. [8]

Die folgenden Beispiele sollen demonstrieren, dass nicht immer alle Misra-C:2004 Regeln zu jedem Zeitpunkt des Projekts eingesetzt werden können.

Beispiel 1: Instrumentierter Code

Regel 2.4 (Codefragmente sollten nicht auskommentiert werden) und Regel 14.1 (Es sollte keinen Code geben, der im Rahmen dieser Software nie ausgeführt werden kann / wird, sogenannter Dead-Code) machen während der Implementierungsphase wenig Sinn, da zu diesem Zeitpunkt im Code viel getestet und empirisch[8] programmiert wird. Würde man diese Regeln dann einhalten, bedeutet das eine Steigerung des Zeitaufwands für die Implementierung. Dieses Beispiel gilt auch für zukünftige Weiterentwicklungen des Codes (während der Implementierungsphase).

Nach Abschluss eines Moduls können diese Regeln wieder angewendet werden, am besten bevor der Integrationstest und spätestens bevor der Systemtest durchgeführt wird.

Beispiel 2: Bestimmte Systembeschaffenheiten

Regel 1.1 (Code Konformität zu Ansi-C) kann unter Umständen für bestimmte Prozessoren und bestimmte Kompilierer nicht eingehalten werden, weil deren spezielle Schlüsselworterweiterungen benutzt werden sollen, die nicht in der Ansi-C Norm enthalten sind.

Beispiel 3: Code Migration

Die Regel 11.2 (Bezug auf Vermeidung von **void**-Zeigern) auf komplexe Anwendungen anzuwenden, kann viel Aufwand nach sich ziehen, wenn im Code viele Umwandlungen (casts) zu und von **void**-Zeigern (undefinierte Zeiger) gemacht werden.

Der Standard Misra-C:2004 besteht aus insgesamt 141 Regeln, von denen 121 benötigte und 20 optionale Regeln sind. Erfahrungen aus der Branche haben gezeigt, dass nur maximal drei Viertel der Anforderungen komplett erfüllt werden. Der Rest der Anforderungen wird entweder nur teilweise oder gar nicht erfüllt [17].

[8] Ausprobieren

Die Aufteilung der Regeln ergibt sich wie folgt (in numerischer Reihenfolge) [8]:

Nr.	Kategorie	Benötigt	Optional	Gesamt
1	Environment	4	1	5
2	Language Extensions	3	1	4
3	Documentation	5	1	6
4	Character Sets	2	0	2
5	Identifiers	4	3	7
6	Types	4	1	5
7	Constants	1	0	1
8	Declarations and Definitions	12	0	12
9	Initialisation	3	0	3
10	Arithmetic Type Conversions	6	0	6
11	Pointer Type Conversions	3	2	5
12	Expressions	9	4	13
13	Control Statement Expressions	6	1	7
14	Control Flow	10	0	10
15	Switch Statements	5	0	5
16	Functions	9	1	10
17	Pointers and Arrays	5	1	6
18	Structures and Unions	4	0	4
19	Preprocessing Directives	13	4	17
20	Standard Libraries	12	0	12
21	Runtime Failures	1	0	1

Tabelle 1 - Misra-C:2004 Kategorien

In **Tabelle 2 - Misra-C:2004 Regeln, die die vier kritischen Kategorien von C abdecken** werden alle Regeln aufgeführt, die zu den oben genannten vier Kategorien / Aspekten gehören. Dort sind 55 Regeln aufgelistet. Keine Regel des Standards bezieht sich auf Aspekt 4. Alle nicht aufgeführten Regeln beziehen sich auf andere Bereiche, auf die hier nicht weiter eingegangen wird. Aufgrund der kategorieübergreifenden (siehe Auflistung weiter unten) Regeln können die einzelnen Regeln und Kategorien nicht direkt mit den vier Kategorien in Verbindung gebracht werden. Kategorieübergreifend bedeutet, dass eine Regel sich in mehrere Kategorien einordnen lässt. Siehe hierzu auch **Abbildung 1 - Kritische Aspekte von C [9]**.

Die Aspekte wurden wie folgt durchnummeriert:

- Kategorie 1 - Unspezifisches Verhalten (*12 Regeln*)
- Kategorie 2 - Undefiniertes Verhalten (*42 Regeln*)
- Kategorie 3 - Implementierungsbedingtes Verhalten (*19 Regeln*)
- Kategorie 4 - Durch Länderspezifika vorgegebenes Verhalten (*0 Regeln*)

Regel-Nr.	Kategorie 1	Kategorie 2	Kategorie 3	Kategorie 4
1.3	x			
1.4		x	x	
2.1	x			
3.3			x	
3.4			x	
3.5	x		x	
4.1		x	x	
4.2		x	x	
6.1			x	
6.4		x	x	
8.1		x		
8.3		x		
8.4		x		
8.6		x		
8.9		x		
8.12		x		
9.1		x		
9.2		x		
11.1		x		
11.2		x		
11.3			x	
11.5		x		
12.2	x	x		
12.7			x	
12.8		x		
12.12	x		x	
16.1	x	x		
16.6		x		
16.8		x		
17.1		x		
17.2		x		
17.3		x		
17.6		x		
18.1		x		
18.2		x		
18.4			x	
19.2		x		
19.3		x		
19.8		x		
19.9		x		
19.12	x			
19.13	x			
19.14		x		

20.1		x		
20.3		x	x	
20.4	x	x	x	
20.5			x	
20.6		x		
20.7	x	x		
20.8		x	x	
20.9	x	x	x	
20.10		x		
20.11		x	x	
20.12	x	x	x	
21.1		x		

Tabelle 2 - Misra-C:2004 Regeln, die die vier kritischen Kategorien von C abdecken

Eine komplette Übersicht aller Regeln mit den von ihnen abgedeckten Bereichen, befindet sich im Anhang[9] dieser Arbeit. Genaue Definitionen der Regeln wurden nicht im Anhang vermerkt, da die Lizenz des offiziellen Standards [8] dies verbietet.

Die obige Tabelle zeigt, dass der Misra-C:2004 Standard sich zu einem großen Teil mit den kritischen Kategorien der Programmiersprache C befasst. Diese Tatsache ist ein Hauptgrund, weshalb dieses Regelwerk als Grundlage für das zu implementierte Plugin dient. Der zweite wichtige Grund ist, dass das Regelwerk gut ausgereift und wohl überlegt ist.

Die Definitionen der Regeln und des Standards stammen aus dem offiziellen Dokument „Misra-C:2004 - Guidelines for the use of the C language in critical systems“ [8] der Misra. Die Artikel „Plan C“ [9] aus der Zeitschrift *iX* und „Der Programmierstandard Misra“ [17] von *elektronik.de* dienten dem Überblick über den Standard und die Verwendung.

Im nächsten Schritt wird eine geeignete Entwicklungsumgebung benötigt. Eclipse scheint für diese Arbeit eine gute Wahl zu sein.

[9] Datei: „20070901 - Misra-C Regeltabelle.pdf“

2.3) Eclipse Entwicklungsumgebung

Eclipse[10] ist eine Open Source Entwicklungsumgebung, die auf Java basiert. Daher ist sie auch nahezu plattformunabhängig. Im Jahre 2001 wurde das Open Source Projekt gestartet.

Die Umgebung kann mit Plugins[11] erweitert werden. Dies betrifft sowohl die Benutzeroberfläche als auch andere Kompilierer oder Komponenten, die nur reine Logik enthalten (z.B. ein XML-Parser[12]). Mittlerweile gibt es eine Vielzahl von Programmiersprachen, die Eclipse mit Hilfe von Plugins verstehen und verarbeiten können. Es gibt viele Tools, die für die Entwicklung oder Pflege der Software zuständig sind, wie z.B. ein UML-Plugin. In **Tabelle 3 - Bekannte und häufig benutzte Eclipse-Plugins** werden ein paar Plugins vorgestellt.

Eclipse wurde als Entwicklungsumgebung gewählt, weil es plattformunabhängig, mithilfe von Plugins erweiterbar und Open Source ist. Ein Plugin, welches plattformunabhängig sein soll, setzt eine Umgebung, die ebenfalls plattformunabhängig ist, voraus. Eine Open Source Umgebung ist wünschenswert, aber nicht zwingend notwendig. Um ein Plugin zu integrieren, muss die Möglichkeit der Erweiterung einer Umgebung vorhanden sein.

Name des Plugins	Kurzbeschreibung
CDT	Plugin für die Entwicklung von C/C++ Programmen unter Eclipse. Enthält Editor, Debugger, Parser, etc.
JavaDoc	Dieses Plugin erstellt anhand der Kommentare im Code eine HTML basierte Beschreibung und Dokumentation der Software. Mittlerweile fester Bestandteil von Eclipse.
Log4j	Ein mächtiger Logger[13] für Java-Applikationen unter Eclipse.
Together for Eclipse	Umgebung, in der UML-Modelle erstellt werden können und anschließend kann ein Code-Gerüst daraus generiert werden. Gilt sowohl für C als auch für Java.

Tabelle 3 - Bekannte und häufig benutzte Eclipse-Plugins

[10] http://www.eclipse.org
[11] Elemente die sich in Programme integrieren lassen um deren Funktionen zu erweitern
[12] Ein Parser analysiert eine Eingabe, häufiges Anwendungsgebiet: Verifizieren von Sprachen
[13] Programm, das beliebige Ausgaben in Dateien oder auf den Bildschirm schreiben kann

In **Tabelle 4 - Eclipse Plugins für besseren Code** werden die meisten Plugins, die sich mit statischer Code Analyse und ähnlichen Bereichen befassen, aufgelistet. Es konnten keine Plugins für die Programmiersprache C gefunden werden, die sich zumindest zum Teil auf die Problemstellung dieser Arbeit befassen, nur für Java gab es entsprechendes. Aus diesem Grund werden im nächsten Kapitel Programme untersucht, die sich mit statischer Syntaxanalyse für die Programmiersprache C beschäftigen.

Name des Plugins	Sprache	Kurzbeschreibung
EclipsePro Audit	Java	Erweiterbares Plugin, das anhand von Coding-Standards Verstöße im Code meldet und automatisch reparieren kann. Ebenso können diverse Metriken generiert werden.
Klocwork Developer	Java	Bietet statische Syntaxanalyse und automatisiertes Auffinden von Sicherheitsproblemen.
Eclipse Metrics	Java	Generiert diverse Metriken während der Kompilierung. Ergebnisse können als HTML-Datei gespeichert werden.
JLint	Java	Java Code-Analysator.
FindBugs Plug-in	Java	FindBugs kann anhand von definierbaren Mustern Bugs finden und HTML-Reports erstellen.
Checkclipse Plugin	Java	Dieses Plugin überprüft Code auf Einhaltung von definierten Code-Styles.
CAP	Java	Generiert diverse Metriken mit grafischen Ausgaben, wie z.B. Tortendiagramm.
Cerp C++ Eclipse Refactoring Plugin	C++	Plugin für Refactoring von C++ Code in Eclipse.

Tabelle 4 - Eclipse Plugins für besseren Code

Am 02.09.2007 waren 928 Plugins in der Plugin-Datenbank[14], die auf der offiziellen Eclipse-Homepage zu finden ist, eingetragen.

Abschließend wird ein möglichst weit verbreiteter Kompilierer benötigt, der in der Lage ist, Regeln des Misra-C:2004 Standards zu überprüfen.

[14] http://www.eclipseplugincentral.com/

2.4) Gnu Compiler Collection (GCC)

Die **G**nu **C**ompiler **C**ollection (GCC)[15] ist eine Sammlung von Kompilierern für Linux / Unix Betriebssysteme. Unter anderem sind Kompilierer für Ada, C und Java vorhanden. Im Rahmen des Projekts wurde 1987 die erste Beta-Version veröffentlicht. Es wird sehr selten ein anderes Werkzeug zum Kompilieren unter Linux genutzt als die GCC.

Für Windows gibt es mittlerweile eine Portierung, die aber noch nicht alle Funktionen und Möglichkeiten der GCC beinhaltet. Sie trägt den Namen MinGW. Die bekanntesten Konkurrenten für die Windows Plattform sind die Kompilierer von Microsoft (MS Visual-C++) und von Inprise/Borland (C++ Builder).

Die GCC besitzt bereits eine Sammlung an möglichen Überprüfungen, die an C-Code mithilfe der statischen Syntaxanalyse durchgeführt werden können. In Kapitel **4** werden die Funktionsweise und die Abdeckung der Misra-C:2004 Regeln durch die vorhandenen Überprüfungen der GCC näher betrachtet.

Die Funktionseinschränkungen des MinGW beziehen sich auf keine der in dieser Arbeit angesprochenen Funktionen für die Kompilierung oder der statischen Syntaxanalyse.

Alternative Programme, die in der Lage sind statische Syntaxanalyse an C-Code durchzuführen, werden im Anschluss betrachtet.

[15] http://gcc.gnu.org/

3) Untersuchung vorhandener Werkzeuge

Für die statische Syntaxanalyse von C-Programmen existieren bereits einige Open Source und kommerzielle Programme auf dem Markt. Eine kurze Recherche ergab eine kleine Übersicht von Programmen, die zur statischen C-Code Analyse eingesetzt werden können.

Programm	URL
Splint	http://splint.org/
PC-Lint / FlexeLint	http://www.gimpel.com/html/products.htm
Flawfinder	http://www.dwheeler.com/flawfinder/
Mops	http://www.cs.berkeley.edu/~daw/mops/
Boon	http://www.cs.berkeley.edu/~daw/boon/
Smatch!!	http://smatch.sourceforge.net/
Oink / Cqual++	http://www.cubewano.org/oink
Axivion Bauhaus Suite	http://www.axivion.com/
Coverity Prevent SQS for C/C++	http://www.coverity.com/
Klocwork K7	http://www.klocwork.com/
Fortify Source Code Analysis	http://www.fortifysoftware.com/products/sca/
Understand	http://www.scitools.com/
QA·C	http://www.programmingresearch.com/

Tabelle 3 - Programme zur statischen C-Code Analyse

Im folgenden Abschnitt werden die bekanntesten und wichtigsten Programme genauer betrachtet. Die nicht näher erläuterten Programme sind nicht weit genug verbreitet oder zu speziell, so dass nur sehr wenige Regeln des Misra-C:2004 abgedeckt werden. Für eine detailliertere Übersicht siehe Tabelle[16] im Anhang.

Die Programme, die hier nicht weiter erläutert werden, sind für diese Arbeit weitestgehend irrelevant, weil sie keine Regeln des Misra-C:2004 Standards überprüfen können. Sie werden nur aufgeführt, weil sie gewisse Teilziele des Standards anstreben, nämlich die Sicherheit von C-Programmen weitestgehend zu gewährleisten.

[16] Datei: „20070527 - Statische Syntaxanalyseprogramme - Übersicht.pdf“

Für die Integrierbarkeit in eine Entwicklungsumgebung, z.B. Eclipse, wurde als Kriterium gewählt, dass sich das Programm als Fenster, Reiter[17] etc. in die Umgebung einbauen lässt und nicht nur im Kompilier-Befehl mit eingebaut wird, so dass die Ausgaben in der Konsole zu lesen sind. Eine Interaktion mit der Umgebung wäre auch wünschenswert, z.B. direktes Springen zu der Codezeile, in der eine Warnung oder ein Fehler aufgetreten ist. Eine reine Ausgabe in der Konsole ist unzureichend, daher wird das Programm erst einmal als „nicht integrierbar“ bezeichnet.

3.1) Splint

Splint ist ein Open Source Programm, das unter der GPL vertrieben wird. Es ist eine indirekte Weiterentwicklung des Programms **Lint**. Die aktuellste Version 3.1.2 vom 12.07.2007 ist sowohl auf Windows als auch auf Linux lauffähig.

Das Programm ist eine Konsolenapplikation, die über keinerlei grafische Oberfläche verfügt. Daher lässt sie sich nicht ohne die Programmierung von Adaptern in eine Entwicklungsumgebung, wie z.B. Eclipse, integrieren. Das Programm kann einzelne Dateien, ganze Verzeichnisse oder eine Verzeichnisstruktur durchsuchen und alle darin befindlichen Dateien überprüfen.

Das Format der Ausgaben lässt sich nach Belieben anpassen. Splint basiert auf den meisten Regeln des ISO99 / Ansi-C Standards. Ohne Definition eigener Regeln unterstützt Splint nur teilweise die Misra-C:2004 Regel 1.1 aus der Kategorie Environment (siehe **Tabelle 1 - Misra-C:2004 Kategorien**).

Es können eigene Regeln mit Hilfe von Attributen erstellt werden. Dies gilt für einige Regeln des Misra-C:2004 Standards. Des Weiteren können Überprüfungen durch sogenannte „stilisierte Kommentare“[18] gesteuert werden (z.B.: */*@null@*/* weist darauf hin, dass eine Variable eventuell **NULL** sein kann bzw. darf).

[17] Element in einer Benutzeroberfläche, welches wie ein Register von Aktenordnern funktioniert
[18] Kommentare die Anweisung für Syntaxanalyse-Programme enthalten

3.2) PC-Lint / FlexeLint

PC-Lint und FlexeLint sind kommerzielle Programme der Firma Gimpel Software. Die beiden Programme unterscheiden sich darin, dass sie auf verschiedenen Plattformen lauffähig sind. FlexeLint ist die Linux/Unix- und PC-Lint die Windows-Variante. Die aktuellste Version ist 8.0w vom 27.07.2007.

Das Programm ist wie Splint eine Konsolenapplikation. Es kann einzelne Dateien, ganze Verzeichnisse oder eine Verzeichnisstruktur durchsuchen und alle darin befindlichen Dateien überprüfen.

Die meisten Regeln des Misra-C:1998 Standards werden unterstützt. Es gibt aber auch ein paar Regeln, die bisher nicht unterstützt werden. Dafür sind Regeln und Überprüfungen mit inbegriffen, die nicht dem Misra-C:1998 Regelwerk angehören. Angepasst an das Misra-C:2004 Regelwerk wurde die Software bisher nicht.

PC-Lint und FlexeLint haben auch Überprüfungen für objektorientierte Bestandteile von C++ dabei (z.B. eine Überprüfung, ob es Klassenvariablen gibt, die nicht im Konstruktor initialisiert werden). Fehlermeldungen und Warnungen werden in Form einer HTML-Datei ausgegeben.

In **Tabelle 4 - PC-Lint / FlexeLint Unterstützung der Misra-C:1998 Regeln [21]** wird gezeigt, wie die Abdeckung der einzelnen Regel-Kategorien aussieht. Die Tabelle soll der Übersicht der abgedeckten Misra-C Kategorien dienen. Die Kategorien sind weitestgehend gleich geblieben zwischen den beiden Standards. Keine Kategorie wurde ausgelassen, aber es wurden nicht immer alle Regeln aus einer Kategorie umgesetzt. Für eine Mapping-Übersicht zwischen Misra-C:1998 zu Misra-C:2004 Regeln, siehe [8] – Anhang B.

Vollständig umgesetzt wurden beispielsweise die Kategorien: Identifiers, Constants, Initialisation und Conversions. Aufgrund der Tatsache, dass nur der alte Regelstandard unterstützt wird, ist die Abdeckung für den neuen Standard noch geringer, da die Regeln teilweise verändert, aufgeteilt oder verworfen wurden. Folgende Regelabdeckung gibt der Hersteller Gimpelsoft an[19]:

[19] http://www.gimpel.com/html/misra.htm

Kategorie	Unterstützt	Offen	Gesamt
Environment	2	2	4
Character Sets	3	1	4
Comments	1	1	2
Identifiers	2	0	2
Types	3	2	5
Constants	2	0	2
Declarations and Definitions	8	1	9
Initialisation	3	0	3
Operators	7	3	10
Conversions	3	0	3
Expressions	5	1	6
Control Flow	10	5	15
Functions	9	9	18
Preprocessing Directives	9	4	13
Pointers and Arrays	2	5	7
Structures and Unions	4	2	6
Shared Libraries	11	2	13

Tabelle 4 - PC-Lint / FlexeLint Unterstützung der Misra-C:1998 Regeln [21]

3.3) QA-C / QA-Misra

QA-C und QA-Misra sind kommerzielle Programme, welche speziell für die Überprüfungen der Misra-C:2004 Regeln entwickelt wurden [20].

Hierbei werden alle Misra-C:2004 Regeln unterstützt. Das Programm ist frei konfigurierbar in Bezug auf die Regelnutzung. Offene Schnittstellen erlauben das Implementieren von eigenen Regeln.

Die grafische Oberfläche bietet zu Fehlern oder Warnungen Verbesserungsvorschläge direkt im Code an. Des Weiteren kann dort direkt die Regel-Erklärung aufgerufen werden. Es können Reports im PDF-Format über die Softwarequalität erstellt werden.

Zusätzlich können grafische Analysen erstellt werden, die unter anderem einen Komplexitätsgraphen enthalten. Software Metriken können generiert werden. Das Programm lässt sich in manche Entwicklungsumgebungen integrieren, aber nicht in Eclipse.

3.4) Flawfinder

Flawfinder ist ein Open Source Programm, das unter der GPL vertrieben wird. Aktuell ist die Version 1.27 vom 16.01.2007. Das Programm läuft unter Linux/Unix.

Flawfinder unterstützt nur wenige, aber wichtige Überprüfungen für die Programmiersprache C. Dabei handelt es sich hauptsächlich um sicherheitskritische Aspekte im Bereich der Informationssicherheit (z.B. Buffer Overflow Risiken, Format-String Probleme). Diese Aspekte fallen unter die Problemkategorien „undefiniertes Verhalten“ und „implementierungsbedingtes Verhalten“. Die Möglichkeiten des Programms zur Überprüfung haben nur eine indirekte Verbindung zu den Misra-C:2004 Regeln: Das Ziel. Daher kann hier keine Auflistung von abgedeckten oder nicht abgedeckten Regeln erfolgen.

Über sogenannte „stilisierte Kommentare“ können gewisse Code-Abschnitte ignoriert werden (z.B.: */* Flawfinder: ignore */*). Das Programm ist eine Konsolenapplikation, die über keinerlei grafische Oberfläche verfügt. Daher lässt sie sich nicht ohne die Programmierung von Adaptern in eine Entwicklungsumgebung, wie z.B. Eclipse, integrieren. Das Programm kann einzelne Dateien, ganze Verzeichnisse oder eine Verzeichnisstruktur durchsuchen und alle darin befindlichen Dateien überprüfen.

Die Ausgabe lässt sich als HTML-Dokument generieren. Die gefundenen Sicherheitsrisiken werden von eins bis fünf, je nach Schwere des Risikos, kategorisiert. Regeln können nur schwer hinzugefügt werden, nämlich nur durch direkten Eingriff in die Datenbank des Programms.

3.5) Folgerung

Der vorhergehende Teil des Kapitels zeigt, wie groß die Unterschiede zwischen den kommerziellen und den Open Source Programmen sind. Die kommerziellen Programme kosten relativ viel, dafür besitzen sie detaillierte Auswertungen, meistens eine grafische Oberfläche und decken deutlich mehr Regeln des Misra-C:2004 Standards ab. Die Open Source Programme hingegen, sind meist nur schwer und umständlich zu bedienen und decken wenige Regeln ab.

3.5.1) Erfüllung der Anforderungen

Es gibt keine Open Source Software, die alle Misra-C:2004 Regeln abdeckt. Mit viel Aufwand kann Splint zwar dazu gebracht werden, einen Großteil der Regeln zu unterstützen, aber das erfordert viel Zeit und Arbeit. Bei den kostenpflichtigen Programmen unterstützt nur QA-C / QA-Misra alle Regeln des Standards.

Eine Integration in Entwicklungsumgebungen und Verbesserungsvorschläge bieten auch nur wenige Programme. Wenn eine Integration möglich ist, dann nur als Aufruf aus der Entwicklungsumgebung, so dass das Ergebnis in der Konsole sichtbar ist. Eine Interaktion mit Eclipse ist aber nicht möglich.

Wünschenswert sind daher folgende Punkte:

- Misra-C:2004 Regeln müssen vordefiniert und anpassungsfähig (konfigurierbar über Optionen) sein. Weiterhin müssen neue Regeln hinzugefügt werden können.
- Es müssen Prüfberichte oder auch Reports erstellt werden können, die Auskunft über die Konformität des Codes zu den Regeln gibt. Idealerweise sollten diese über Templates gesteuert werden können.
- Das Programm sollte in eine Entwicklungsumgebung, bevorzugt Eclipse, integrierbar sein.
- Es sollte eine grafische Benutzeroberfläche existieren, die direkt betroffene Code-Stellen markieren kann bzw. vorhandene Eclipse Funktionen hierfür verwendet. Die Oberfläche kann auch aus vorhandenen Eclipse-Komponenten bestehen. Wichtig ist, dass die Möglichkeit der Verlinkung zwischen Meldungen und den betroffenen Code-Stellen gegeben ist.
- Lauffähig sollte das Programm sowohl unter Windows als auch unter Linux sein.

Eine detailliertere Aufstellung der Anforderungen befindet sich in Kapitel **5.1**.

3.5.2) Vorteile des Plugins

Das hier zu entwickelnde Eclipse-Plugin führt in einer denkbaren endgültigen Version (nicht Bestandteil dieser Arbeit) alle diese Punkte zusammen. Umzusetzende Ziele im Rahmen dieser Arbeit und optionale Ziele für die Weiterentwicklung finden sich in Kapitel **5** und in Kapitel **7**. Die folgenden Punkte wären alle denkbar für spätere Versionen des Plugins.

- Das Plugin ist eine Integration in die Entwicklungsumgebung Eclipse.
- Die Möglichkeit zur Erstellung eigener Regeln ist gegeben.
- Vorhandene Regeln können verändert / angepasst werden.
- Freie Konfiguration des Plugins (Welche Regeln benutzt werden sollen, spezielle Einstellungen für einzelne Regeln).
- Verbesserungsvorschläge für den Code werden gemacht und können über die Quick-Fix[20] Funktionalität angewendet werden. Jede Regel kann eine oder mehrere Muster für die Verbesserung bekommen. Je nach Art der Regel kann vorhandener Code verbessert oder fehlender Code hinzugefügt werden. Beispielsweise könnte man auf diese Weise auch überflüssigen Dead-Code entfernen oder fehlende Klammern setzen lassen.
- Eine direkte Verlinkung der Fehlermeldungen bzw. Warnungen zum Code ermöglicht schnelles Bearbeiten.
- Lauffähig auf allen Plattformen, die Java und einen C-Kompilierer unterstützen.
- Open Source (GPL)
- Interagiert mit Kompilierer, so dass Misra-C:2004 Überprüfungen, die der Kompilierer machen kann, auch von ihm ausgeführt werden. Das Plugin verarbeitet dann das Ergebnis und vereint die Meldungen des Kompilierers mit seinen eigenen.
- Plugin kann auf verschiedene Sprachen erweitert werden. Auf diese Weise kann ein Plugin für diverse weitere Sprachen verwendet werden. Insbesondere interessant, wenn ein Projekt Module in verschiedenen Programmiersprachen besitzt.
- Es können verschiedene Metriken[21] erstellt werden.
- Konfigurierbare HTML- und PDF-Reports können auf Wunsch generiert werden.

[20] Eclipse-Funktionalität um anhand von Mustern Fehler oder Warnungen im Code zu verbessern
[21] Messungen zu bestimmten Merkmalen von Software und Prozessen

3.5.3) Fazit

Das zu implementierende Plugin vereint alle Funktionalitäten der getesteten freien und kommerziellen Programme zu einer Anwendung. Ergänzend kommen noch einige Funktionalitäten hinzu, die keine der geprüften Anwendungen anbieten. Das bezieht sich z.B. auf die Nutzung von anderen Sprachen als C oder die Integration in die Entwicklungsumgebung Eclipse.

Da das Plugin unter der GPL läuft, können je nach Belieben neue Funktionalitäten hinzugefügt werden, die bisher nicht berücksichtig wurden. Ebenso kann das Programm in eine andere Anwendung integriert werden, die durch das Plugin ergänzt wird.

4) Untersuchung der GCC

Da das Eclipse-Plugin eine plattformunabhängige Lösung ist, muss auch ein Kompilierer gefunden werden, der auf möglichst vielen Plattformen vorhanden und bekannt ist. Hier bietet sich die GCC an, da sie sehr bekannt und weit verbreitet ist. Auf Linux / Unix Plattformen werden C-Programme fast ausschließlich mit der GCC kompiliert. Für Windows gibt es eine entsprechende Portierung.

Alternativ kann jeder beliebige C-Kompilierer benutzt werden. Für andere Kompilierer müssen aber die Regeln, die hier durch die GCC bereits abgedeckt werden, noch implementiert werden, sofern der Kompilierer dafür keine Prüfungen anbietet. Bei einem Plattformwechsel muss im Zweifelsfall ein anderer Kompilierer gesucht werden, da nur sehr wenige Kompilierer so viele Plattformen abdecken wie die GCC.

4.1) Übersicht

Die GCC bietet von vornherein einige Möglichkeiten für die statische Syntaxanalyse. Vom Kompilierer können einige Misra-C:2004 Regeln angewendet werden.

Betrachtet wird die GCC in der Version 4.2.1. Ältere Versionen besitzen manche Prüfungen unter Umständen noch nicht. Zum Zeitpunkt der Entstehung dieser Arbeit war dies die aktuellste Version.

Wenn alle Warnungen der GCC aktiviert sind, werden die meisten traditionellen Überprüfungen von Lint mit ausgeführt. Um beliebige Programme während des Kompilierungsvorgangs (beispielsweise bei einem Nightly-Build[22] automatisiert mitlaufen zu lassen, muss ein entsprechender Eintrag im Makefile gemacht werden. Auf diese Methodik wird hier nicht weiter eingegangen, weil es das Ziel ist diese Einstellungen direkt in der Entwicklungsumgebung zu machen.

Die meisten Parameter, die hier aufgeführt werden, haben sowohl eine negative als auch eine positive Variante. Dadurch kann man die Regeln, je nach Belieben, explizit an- oder ausschalten (Beispiel: ‚-ffoo' und ‚-fno-foo'). Sie sind sehr nützlich, wenn man alle Meldungen bis auf eine bestimmte, haben will. Dort kann die Regel dann explizit deaktiviert werden. Auf die negativen Varianten wird hier nicht weiter eingegangen.

[22] Nächtliche Kompilierung des gesamten Projekts mit optionalen Syntaxüberprüfungen

Hinweis: Es werden keine ausführlichen Beschreibungen für die GCC-Schalter aufgeführt. Für genauere Informationen siehe Handbuch der GCC [10]. Parameter zum Unterdrücken bestimmter Warnungen werden weggelassen, da diese irrelevant sind.

4.2) Unterstützte Misra-C:2004 Regeln

Die folgende Tabelle gibt eine kurze Übersicht über die Regeln, die die GCC bereits teilweise oder sogar vollständig, abdeckt. Alle anderen Regeln müssen anderweitig, z.B. mit einem extra Analyse-Tool geprüft werden.

Regel Nr.	Kategorie	GCC Abdeckung	GCC Schalter
1.1	Environment	vollständig	-pedantic / -ansi
2.3	Language extensions	vollständig	-Wcomment
4.2	Character sets	vollständig	-Wtrigraphs
8.1	Declarations and definitions	teilweise	-Wmissing-prototypes
8.2	Declarations and definitions	vollständig	-Wimplicit
9.1	Initialisation	vollständig	-Wuninitialized
9.2	Initialisation	vollständig	-Wmissing-braces
13.3	Control statement expressions	vollständig	-Wfloat-equal
14.1	Control flow	vollständig	-Wunreachable-code
15.3	Switch statements	teilweise	-Wswitch-default
16.3	Functions	vollständig	-Wstrict-prototypes
16.8	Functions	vollständig	-Wreturn-type
19.11	Processing directives	vollständig	-Wundef

Tabelle 5 - GCC-Schalter für Misra-C:2004 Regeln

Die GCC deckt 11 Regeln des Misra-C:2004 Regelwerks vollständig und zwei teilweise ab. Die Kategorie Initialisation wird bis auf eine Regel, nämlich 9.3, vollständig abgedeckt. Aus den anderen Kategorien werden entweder gar keine Regeln oder nur ein bis zwei Regeln abgedeckt. Siehe hierzu **Tabelle 5 - GCC-Schalter für Misra-C:2004 Regeln**.

Um eine Abbruchkontrolle zu schaffen, kann die GCC so konfiguriert werden, dass sie alle oder auch ausgewählte Warnungen in Fehlermeldungen umwandelt. Folgende Parameter werden hierfür benötigt:

-Werror

Wenn die Prüfung auf Misra-Regeln streng sein soll bzw. keine Warnungen im Projekt auftauchen sollen, was meist als sinnvoll zu erachten ist, muss **–Werror** als zusätzlicher Parameter genutzt werden. Alle Warnungen werden dann als Fehler ausgegeben.

-Werror=<Rule-Name>

Lässt einzelne Warnungen auswählen, die zu einem Fehler gemacht werden sollen.

Regel 1.1 (Environment):

-pedantic in Verbindung mit -ansi

Mit diesen Schaltern weist man den Kompilierer an, sich bei der Überprüfung strikt an den ANSI-Standard zu halten. Die Option **–pedantic** wirft Fehler aus, sobald sich ein Programm nicht strikt an den ISO C-Standard hält. Die Option **-ansi** schaltet einige GNU C-Erweiterungen aus, die nicht im ANSI-C-Standard enthalten sind.

Regel 2.3 (Language extensions):

-Wcomment

Warnt, wenn die Kommentareinleitungssequenz **/*** innerhalb eines Kommentars gefunden wird.

Regel 4.2 (Character sets):

-Wtrigraphs

Warnt, wenn Trigraphs[23] gefunden werden, die das Programm verändern könnten.

Teilweise Regel 8.1 (Declarations and definitions):

-Wmissing-prototypes

Warnt, wenn eine globale Funktion ohne einen vorher definierten Prototyp definiert wird. Ziel dieser Warnung ist es, globale Funktionen zu finden, die nicht in Header-Files deklariert wurden.

Es fehlt der Aspekt der Sichtbarkeit.

Regel 8.2 (Declarations and definitions):

-Wimplicit

Warnt, wenn eine Deklaration keinen Typ angibt.

-Wimplicit-function-declaration

Gibt eine Warnung aus, wenn eine Funktion benutzt wird, bevor diese deklariert wurde.

[23] Drei-Zeichen-Sequenz die mit *??* beginnt um bestimmte ASCII-Zeichen darzustellen

Regel 9.1 (Initialisation):

-Wuninitialized

Warnt bei automatic-Variablen, die benutzt (also ausgelesen) werden, ohne vorher einen Initialwert erhalten zu haben. Diese Warnung ist nur möglich, wenn die Optimierung (**-O**) zugeschaltet ist, da sie Datenflussinformationen benötigt, die nur während der Optimierung errechnet werden.

Regel 9.2 (Initialisation):

-Wmissing-braces

Warnt, wenn eine Aggregat- oder Union-Initialisierung nicht vollständig „umklammert“ ist. Das bedeutet, dass die Struktur anhand der Klammern ersichtlich sein muss. Ein kurzes Beispiel:

```
int a[2][2] = { 0, 1, 2, 3 };                 // Nicht vollständig umklammert
int b[2][2] = { { 0, 1 }, { 2, 3 } };         // Richtig!
```

Regel 13.3 (Control statement expressions):

-Wfloat-equal

Warnt, wenn Fließkommawerte miteinander auf Gleichheit überprüft werden.

Regel 14.1 (Control flow):

-Wunreachable-code

Warnt, wenn Code gefunden wird, der niemals ausgeführt werden kann.

Teilweise Regel 15.3 (Switch statements):

-Wswitch-default

Warnt, wenn eine Switch-Anweisung keinen Default-Zweig hat.

*So kann das **default** an jeder beliebigen Stelle stehen. Laut Misra-Regel sollte der **default**-Zweig immer am Ende der Anweisung stehen.*

Regel 16.3 (Functions):

-Wstrict-prototypes

Warnt, wenn eine Methode/Funktion deklariert oder definiert wird, ohne dass die Typen der Argumente angegeben wurden.

Regel 16.8 (Functions):

-Wreturn-type

Warnt, wenn eine Funktion, die nicht vom Typ **void** definiert ist, nicht durch Return beendet wird. Außerdem warnt diese Option, wenn in einer Funktion, die nicht vom Typ **void** ist, ein **return** ohne Parameter angegeben wird.

Regel 19.11 (Functions):

-Wundef

Warnt, wenn ein undefinierter Identifier in einer '#if' Anweisung vorkommt.

Beispielaufruf mit allen unterstützten Misra-C:2004 Regeln als Warnungen:

```
gcc test.cc –o test -Wimplicit -Wimplicit-function-declaration -Wreturn-type
-Wcomment -Wtrigraphs -Wuninitialized -pedantic -Wmissing-braces -Wfloat-equal
-Wundef -Wstrict-prototypes -Wmissing-prototypes -Wunreachable-code
-Wswitch-default
```

4.3) Syntaxüberprüfungen, ergänzend zum Misra-C:2004 Standard

Der GCC C-Kompilierer kann darüber hinaus noch einige Überprüfungen vornehmen, die nicht im Misra-C:2004 Regelwerk enthalten sind. Manche davon sind auch durchaus empfehlenswert. Man sollte sich diese optional ansehen und unter Umständen auch verwenden. Da diese optionalen Überprüfungen sinnvolle Ergänzungen zum Misra-C:2004 Regelwerk sind und sich in wenigen Minuten in das Plugin integrieren lassen, werden sie kurz erläutert.

-fsigned-char

Der Typ **char** wird als **signed** angenommen.

-Wunused

Warnt bei lokalen Variablen, die nach ihrer Deklaration nicht benutzt werden. Warnt ebenso bei mit **static** deklarierten Funktionen, die nie definiert werden und bei Ausdrücken, die einen Wert berechnen, der nicht explizit benutzt wird.

-Wparentheses

Warnt in verschiedenen Fällen, in denen Klammern weggelassen wurden, die zum Kompilieren nicht notwendig sind, aber die die Lesbarkeit des Codes verschlechtern. Die Wahrscheinlichkeit einen Fehler zu machen, steigt durch den schlechter lesbaren Code.

-Wnonnull

Warnt bei Null-Pointern in Argumenten, die als non-null markiert sind mit dem Attribut **nonnull**.

-Winit-self

Warnt bei nicht initialisierten Variablen, die mit sich selber initialisiert wurden. Funktioniert nur in Verbindung mit **-Wuninitialized**.

-Wmissing-include-dirs

Warnt, wenn ein angegebenes Include-Verzeichnis nicht existiert.

-Wsequence-point

Warnt bei Code, der eine undefinierte Semantik aufgrund von Verstößen durch **sequence-point** Regeln aufweist.

-Wunused-function

Warnt, wenn eine Funktion als static deklariert wurde, aber nie benutzt oder definiert wird.

-Wunused-label

Warnt, wenn ein Label deklariert wurde, welches aber nicht benutzt wird.

-Wunused-parameter

Warnt, wenn ein Funktionsparameter nicht benutzt wird.

-Wunused-variable

Warnt, wenn eine lokale oder eine nicht **const static** Variable nicht benutzt wird.

-Wunused-value

Warnt, wenn eine Anweisung einen Wert errechnet, dieser aber nie benutzt wird.

-Wall

Kombiniert alle unused-Parameter.

-Wunknown-pragmas

Warnt, wenn eine **#pragma** Anweisung benutzt wird, die der GCC unbekannt ist.

-Wsystem-headers

Gibt Warnungen aus, die durch Konstrukte in System-Header-Files entstehen. Diese werden normalerweise unterdrückt.

-Wdeclaration-after-statement

Warnt, wenn eine Deklaration nach einer Anweisung gefunden wird.

-Wbad-function-cast

Warnt, wenn der Rückgabewert eines Funktionsaufrufes zu einem anderen Typ gecastet wird, als definiert wurde.

-Wcast-qual

Warnt, wenn ein Pointer so gecastet wird, dass ein Teil der Typ-Deklaration entfernt wird.

-Wsign-compare

Warnt, wenn der Vergleich zwischen vorzeichenbehafteten und nicht vorzeichenbehafteten Werten/Datentypen ein falsches Ergebnis auslösen kann, durch die Konvertierung von vorzeichenbehaftet zu nicht vorzeichenbehaftet.

-Waddress

Warnt bei der Verwendung von verdächtigen Speicher-Adressen. Zum Beispiel, wenn die Adresse einer Funktion in einer **if**-Anweisung benutzt wird.

-Waggregate-return

Warnt, sobald eine Funktion eine Struktur oder ein Union zurückgibt, oder zumindest so definiert wird.

-Wmissing-field-initializers

Warnt, wenn bei der Initialisierung einer Struktur nicht alle Werte initialisiert werden.

-Wpacked

Warnt, wenn eine Struktur das Attribut **packed** besitzt, aber dies keine Auswirkung auf den Aufbau oder die Größe der Struktur hat.

-Wnested-externs

Warnt, wenn eine **extern**-Deklaration in einer Funktion vorkommt.

-Winline

Warnt, wenn eine Funktion als **inline** deklariert wurde, aber nicht als solche behandelt werden kann.

-Wlong-long

Warnt, wenn der Typ **LongLong** benutzt wird.

-Wvolatile-register-var

Warnt, wenn eine Register-Variable als **volatile** deklariert wurde.

-Woverlength-strings

Warnt, wenn String-Konstanten größer als die “Minimum Maximum” Länge aus dem C-Standard ist.

-Wimplicit-function-declaration

Warnt, wenn eine Funktion benutzt wird, bevor sie definiert wurde.

-Wmain

Warnt, wenn die Funktion **main** mit ungewöhnlichem Rückgabewert oder ungewöhnlichen Parametern spezifiziert wurde.

-Wswitch

Warnt, wenn eine **switch**-Anweisung einen Index mit einem Aufzählungstyp hat und nicht alle Elemente dieses Aufzählungstyps durch **case**-Zweige abgearbeitet werden. Wenn ein **case**-Zweig außerhalb des Wertebereichs des Aufzählungstyps ist, wird ebenfalls eine Warnung ausgegeben.

-Wformat

Prüft, ob die Argumente in **printf**- oder **scanf**-Anweisungen zu den entsprechenden Formatstrings passen. **-Wformat** enthält **-Wnonnull**. Für eine noch bessere Kontrolle gibt es noch: **-Wformat-y2k**, **-Wno-format-extra-args**, **-Wno-format-zero-length**, **-Wformat-nonliteral**, **-Wformat-security**, and **-Wformat=2**.

-Wchar-subscripts

Warnt, wenn der Indextyp eines Arrays **char** ist.

4.4) Weitere mögliche Einstellungen

Die GCC bietet noch einige interessante Konfigurationsmöglichkeiten, die das Überprüfen der Ausgaben erleichtert. Diese sind notwendig, wenn die Ausgaben der GCC in einem anderen Programm (z.B. im zu implementierenden Plugin) geparst und umgewandelt werden müssen, damit sie ein einheitliches Bild mit eigenen Meldungen des Programms ergeben. Hierfür gibt es folgende Ausgabeformatierungen:

-fmessage-length=n

Formatiert Fehlermeldungen auf eine Länge von **n**. Wenn die Länge erreicht wird, wird ein Zeilenumbruch durchgeführt. Wenn **n=0**, dann wird kein Zeilenumbruch eingefügt.

-fdiagnostics-show-location=once/every-line

Wenn der Schalter **once** aktiviert ist, dann wird die Quelle nur einmal angegeben. Bei **every-line** wird die Quelle in jeder neuen Meldung wieder angegeben. Dies ist nur sinnvoll, wenn die vorherige Funktion aktiviert wurde (Zeilenumbrüche).

-fsyntax-only

Führt nur eine Syntax-Prüfung aus (Es wird kein Linker/Kompilierer aufgerufen).

-Wfatal-errors

Wenn dieser Schalter aktiviert ist, dann wird nach dem ersten Fehler der Kompilierungsvorgang abgebrochen und keine weiteren Meldungen ausgegeben.

5) Zielsetzung

Ein Eclipse-Plugin wird entwickelt, welches die gestellten Anforderungen erfüllt. Da es im Rahmen dieser Arbeit darum geht, wie das Plugin prinzipiell arbeitet, werden nicht alle Anforderungen, Regeln und Möglichkeiten implementiert. Im Kapitel **5.1** wird gezeigt, welche Funktionen das exemplarische Plugin aufweisen soll.

Um die Implementierungszeit möglichst gering zu halten und um „das Rad nicht neu zu erfinden“, wird ein vorhandener Parser für die Programmiersprache C genommen und kein eigener programmiert.

5.1) Anforderungen

Da im Rahmen dieser Arbeit nicht genug Zeit zur Verfügung steht alle Funktionen des Plugins zu implementieren, werden die zu implementierenden Funktionen auf ein Minimum beschränkt. Im folgenden Teil des Kapitels wird erläutert, welche Funktionen umgesetzt werden und welche nicht.

5.1.1) Musskriterien

Folgende Kriterien müssen mindestens implementiert werden, um einen funktionierenden exemplarischen Prototyp zu bekommen.

- Das Plugin muss in reinem Java geschrieben sein (keine eingebetteten Sprachen innerhalb des Java-Codes).
- Es muss unter der Entwicklungsumgebung Eclipse lauffähig sein.
- Es muss ein Interface für Regelobjekte definiert werden, so dass leicht neue Regeln hinzugefügt werden können.
- Eine oder mehr Regeln müssen implementiert werden, um exemplarisch zu zeigen, wie eine Regelüberprüfung durchgeführt werden kann.
- Die Überprüfung einer einzelnen Datei oder des gesamten Projekts muss mit dem Plugin möglich sein.

5.1.2) Sollkriterien

Folgende Kriterien müssen implementiert werden, um ein nahezu vollständiges Plugin zu bekommen, welches in der Lage ist, die meisten Funktionen auszuführen bzw. um leicht mit den fehlenden Funktionen erweitert werden zu können.

- Meldungen von Regeln müssen in einem Fenster in Eclipse gezeigt werden.
- Die GCC muss exemplarisch eingebunden werden, um zu zeigen, wie zusätzlich externe Anwendungen genutzt werden können.
- Eine Regelverwaltung muss existieren, die es ermöglicht zu entscheiden, welche Regeln in den Überprüfungsvorgang mit eingeschlossen werden können und welche nicht.
- Im Quellcode sollen Hinweise an den Stellen gezeigt werden, an denen Meldungen auftreten (unter Eclipse heißen diese Markierungen „Marker“).

5.1.3) Kannkriterien

Folgende Kriterien sind optional, aber nicht zwingend notwendig für die angestrebte Demonstration mithilfe des Plugins.

- Unterstützung anderer Sprachen als C.
- Allgemeine Einstellungen und Konfigurationen der Regeln unter den Eclipse-Einstellungen (***Window → Preferences → RuleCheck***).
- Verlinken von Meldungen mit Vorkommen im Code.

5.1.4) Abgrenzungskriterien

Folgende Kriterien werden im Rahmen dieser Arbeit nicht umgesetzt, da sie zu zeitaufwändig wären und nicht notwendig sind für den exemplarischen Prototyp.

- Quick-Fixes (automatische Codeverbesserungen) mithilfe von Mustern anbieten.
- Alle fehlenden Regeln des Misra-C:2004 Regelwerks implementieren.
- Priorisierungen für Regeln ermöglichen.
- Diverse Optimierungen.

Mehr zu diesen Punkten und weitere Ideen, zur Verbesserung des Plugins in der Implementierungsphase, finden sich in der Zusammenfassung in Kapitel **7**.

5.2) Voraussetzungen

Das Plugin benötigt folgende Software um lauffähig zu sein:

- Beliebiges Betriebssystem mit grafischer Oberfläche und einer Java VM.
- Eine Java VM, die mindestens Java 5 unterstützt.
- Eclipse in der Version 3.0 oder höher.
- Die Kern-Bibliothek des BBCDT Plugins von IBM (C-Parser).

Optional wird empfohlen das Eclipse Plugin CDT 4.x zu installieren, um eine C/C++ Unterstützung in Eclipse zu bekommen. Die zu überprüfenden Quellcodes müssen kompilierfähig sein, sonst könnten unter Umständen falsche Meldungen erstellt werden.

Hardwareanforderungen liegen bei den Mindestanforderungen für Java bzw. für Eclipse. Es wird keine weitere spezielle Hardware benötigt.

Erweitert oder verändert werden kann das Plugin in jedem beliebigen Editor, doch es ist empfehlenswert, Eclipse zu verwenden. Dafür spricht ein einfacher Export des Plugins, schnelle und einfache Tests durch JUnit und Testen des Plugins durch eine weitere Instanz der Eclipse Umgebung. Des Weiteren bietet Eclipse einige kleine Tools, die das Entwickeln von Plugins erleichtern.

5.3) Produktbeschreibung

Der folgende Abschnitt dieses Kapitels definiert klare Anforderungen an das Plugin, sowohl funktionale als auch nicht-funktionale. Des Weiteren wird kurz auf die Eigenschaften der Schnittstellen nach außen und auf die Benutzeroberfläche eingegangen.

5.3.1) Funktionale Anforderungen

Funktionale Anforderungen tragen das Prefix „**F**“ (als Beispiel: F1). Die folgenden Anforderungen definieren, was das Programm eigentlich tun soll.

1. Das Plugin soll anhand definierter Regeln eine oder mehrere Quellcodes auf Einhaltung der Regeln überprüfen.
   Es muss möglich sein eine Datei / Verzeichnis anzugeben, die mithilfe der definierten Regeln überprüft wird und jeden Verstoß der Regel unter Angabe der richtigen Position des Aufkommens, gemeldet wird.

2. Verstöße gegen Regeln sollen gemeldet und sowohl direkt im Quellcode an der entsprechenden Stelle markiert werden, als auch als detaillierte Meldung in einem eigenen Fenster dargestellt werden (ohne die Markierung im Code ist es nicht möglich dorthin zu springen).
3. Externe Programme, wie z.B. die GCC, sollen genutzt werden können, um bestimmte Regeln zu prüfen. Die Meldungen sollen genauso behandelt werden wie die intern erstellten.
   Konsolenprogramme müssen in eine Regel gekapselt ausgeführt werden können. Die Ausgaben des Programms aus der Konsole müssen im Regelobjekt zu einem allgemein gültigen Meldungs-Objekt umgewandelt und weitergeleitet werden. Somit ist dem Plugin selber egal woher die Meldungen stammen und welches Format sie vorher hatten.
4. Anhand der Meldung und den Markierungen im Quellcode soll es möglich sein, durch klicken auf die Meldung direkt zur entsprechenden Position im Quellcode zu gelangen.

5.3.2) Nicht-Funktionale Anforderungen

Nicht-Funktionale Anforderungen tragen das Prefix „**N**“ (als Beispiel: N1). Die folgenden Anforderungen definieren, welche Eigenschaften das Plugin haben soll.

1. Es soll keine Fehlertoleranz in Bezug auf nicht kompilierbare Quellcodes geben.
   Dies bedeutet, dass nicht versucht wird möglichst richtige Ausgaben zu erzeugen wenn es Probleme im Quellcode gibt (Beispielsweise eine vergessene öffnende Klammer).
2. Das Plugin ist als exemplarischer Prototyp für Demonstrationszwecke gedacht.
   Es ist auf keinen Fall Marktreif und erfüllt nicht alle Funktionen, die das Plugin haben sollte, um es effizient einsetzbar zu sein.
3. Die Leistung und Effizienz ist für dieses Plugin nicht weiter wichtig.
   Die Geschwindigkeit der Überprüfungen muss nicht besonders hoch sein.
4. Es soll leicht auf andere Plattformen mit Java und Eclipse portierbar sein.
   Plattformen auf denen Java läuft und somit auch Eclipse, muss das Plugin auch lauffähig sein (Beispielsweise Dateinamenkonventionen Windows/Linux).
5. Die Installation soll einfach und schnell gehen.

5.3.3) Benutzungsoberfläche

Die folgenden Anforderungen werden an die Benutzeroberfläche gestellt.

1. Eine einfache und übersichtliche Oberfläche soll das Plugin besitzen.
2. Das Plugin muss einfach zu starten sein.
3. Man muss zwischen einer Datei und dem gesamten Projekt wählen können (bei der Überprüfung).
4. Bedienung über möglichst wenige Elemente, nur das Notwendigste.

5.3.4) Schnittstellen

Das Plugin muss folgende Schnittstellen nach außen bereitstellen.

1. Eintrittspunkt (Startpunkt des Plugins, ähnlich wie die Main-Methode in Programmen) aufgerufen durch Event in Eclipse.
2. Meldungsfenster an Eclipse Oberfläche senden.
3. Öffnen von Dateien im Eclipse Editor und springen zu einer bestimmten Codezeile.

6) Implementierung des Prototyps

In diesem Kapitel wird kurz erläutert, wie Eclipse-Plugins aufgebaut sind und wie diese funktionieren. Anschließend wird die Architektur und die Implementierung des Plugins für die Misra-C:2004 Syntaxanalyse beschrieben.

Das Plugin deckt folgende Regeln ab:

Regel Nr.	Kategorie	Plugin Abdeckung
5.1	Identifiers	vollständig
13.6	Control Statement expressions	vollständig
14.5	Control Flow	vollständig

Tabelle 6 - Misra-C:2004 Regelabdeckung des Plugins

Zusätzlich werden die Regeln, die die GCC abdecken kann, im Plugin eingebunden, sofern GCC oder MinGW installiert und eingerichtet wurde. Hierfür bietet das Plugin eine einfache Schnittstelle zum Erweitern.

Die original Definitionen der bisher unterstützten Regeln sind wie folgt definiert worden [8]:

Regel 5.1:

Identifiers (internal and external) shall not rely on the significance of more than 31 characters.

Regel 13.6:

Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop.

Regel 14.5:

The continue statement shall not be used.

Definitionen der Regeln die von der GCC überprüft werden sind in Kapitel **4.2** aufgelistet. Im Folgenden wird die Architektur von Eclipse-Plugins erläutert.

6.1) Die Eclipse Plugin-Architektur

Eclipse ist eine Entwicklungsumgebung, die dafür ausgelegt wurde mit Plugins erweitert zu werden: „Eclipse was designed as an open extensible IDE for anything, and nothing in particular.” [5]. Es können Elemente der grafischen Oberfläche, Unterstützung für andere Sprachen, oder ganz eigenständige Anwendungen integriert werden (für einige Beispiele siehe **Tabelle 3 - Bekannte und häufig benutzte Eclipse-Plugins**). Eclipse in der Grundversion ist also auch nichts anderes als eine Sammlung von Plugins, die durch ein Framework zusammengefügt wurden. Dieses Framework, sozusagen der Kern von Eclipse, macht nichts anderes als Plugins zu laden, zu starten und zu verwalten.

Alle Plugins können über sogenannte extension points neue Funktionalitäten zu bestehenden Plugins hinzufügen. Jeder kann für sein Plugin eigene extensions points erstellen, so dass das Plugin auch wieder erweitert werden kann, und so weiter. Ein extension point definiert z.B. ein Button in der Werkzeugleiste:

```
<extension point="org.eclipse.ui.actionSets">
   <actionSet
      <action
            label="%action.label.0"
            icon="icons/multi.gif"
            class="org.bachelor.rulecheck.actions.RuleCheckAction"
            tooltip="%action.tooltip.0"
            menubarPath="ruleCheckMenu/ruleCheckGroup"
            toolbarPath="ruleCheckGroup"
            id="org.bachelor.rulecheck.actions.entireProject">
      </action>
    </actionSet>
  </extension>
```

Durch extension points können Teile von vorhandenen Plugins benutzt werden bzw. Teile eines eigenen Plugins zur Benutzung bereitgestellt werden. Das Programm, welches einen extension point benutzt, muss eine zugehörige extension definieren. Wenn man extension points mit einem Interface vergleicht, dann wäre die zugehörige extension die Klasse, die dieses Interface implementiert. An einem extension point können mehere Plugins angeschlossen werden.

Die Registrierung neuer Plugins geschieht erst beim Neustart der Umgebung. Geladen werden die Plugins sobald sie das erste Mal benötigt werden. Diese Vorgehensweise spart Zeit beim Starten von Eclipse und belegen weniger Speicher während der Laufzeit von Eclipse, daher können nahezu unendlich viele Plugins registriert werden. Dank dem OSGI-Framework können die Plugins nach Benutzung wieder entladen werden.

Bis auf diese wenigen Unterschiede ist ein Eclipse-Plugin ein Java Programm, welches statt einer *main()*-Methode einen oder mehrere Einstiegspunkte besitzt, die durch die Aktivierung von vordefinierten Aktionen (extension, siehe obigen Beispielcode) aus der Eclipse-Umgebung angesprochen werden.

Die typischen Dateien, die für ein Plugin benötigt werden, sind:

- **plugin.xml**
  Plug-in manifest (Informationen über das Plugin). Hier wird festgelegt, welche extension points das Plugin benutzt und welche es selber anbietet.
- **plugin.properties**
  Hier sind die Namen der Pluginelemente für die **plugin.xml** vorhanden. Kann z.B. benutzt werden für die Übersetzung der Namen in andere Sprachen.
- **about.html**
  In diese HTML-Datei gehören die Lizenz-Informationen.
- ***.jar**
  Bibliotheken und Plugins, die für das Plugin benötigt werden.
- **lib Directory**
  Hauptsächlicher Speicherort für die oben genannten Bibliotheken.
- **icons Directory**
  In diesem Verzeichnis liegen die Symbole für das Plugin, bevorzugt im *.gif Format.
- **(Sonstige Dateien)**
  Weitere Dateien, die für das Plugin oder die Entwicklung benötigt werden.

Für eine detailliertere Einführung siehe Eclipse-Programmier-Referenz [11]. Basierend auf dem oben erklärten Prinzip kann im nächsten Schritt das exemplarische Plugin implementiert werden.

6.2) Architektur des implementierten Plugins

Das Plugin besteht aus reinem Java und wurde speziell für die oben erwähnte Architektur von Eclipse entwickelt. Es besteht aus mehreren Paketen (Packages), die die Quellcodes kategorisieren und ordnen. Im Projekt befinden sich folgende Pakete:

- **rulecheck**
  Beinhaltet den Einstiegspunkt in das Plugin.

- **rulecheck.actions**
  Beinhaltet alle Events, die das Plugin durch Eclipse entgegennehmen kann.
- **rulecheck.core**
  Beinhaltet den Kern des Plugins.
- **rulecheck.logger**
  Beinhaltet einen Logger, der beim Debuggen des Plugins behilflich ist.
- **rulecheck.messages**
  Beinhaltet Objekte für Meldungen durch das Plugin.
- **rulecheck.preferences**
  Beinhaltet Seiten für die Eclipse Einstellungen.
- **rulecheck.rules**
  Beinhaltet das Regelinterface und implementierte Regeln.
- **rulecheck.ui**
  Beinhaltet Klassen, die die Oberfläche von Eclipse beeinflussen dürfen.

Um nicht einen neuen C-Scanner zu entwickeln, wurde ein freies Plugin der Firma IBM benutzt. Der auf CDT 3.x basierende C-Code-Scanner aus dem Plugin BBCDT[24] wurde ins RuleCheck Plugin mit eingebunden. Das Plugin ist ein Beispiel für einen eigenen Editor für C-Programme unter Eclipse mit Syntaxhervorhebung. Es besteht aus zwei Teilen. Einem Front-end und einem Back-end. Das Front-end beinhaltet den eigentlichen Editor und das Back-end den Parser für die Syntaxhervorhebung. Für das Rule-Check Plugin wird nur das Back-End (BBCDT-Core) benötigt, da dort der Scanner untergebracht wurde.

Der Logger ist nur zum Debuggen gedacht und sonst nicht weiter relevant für dieses Projekt. Daher wird er hier auch nicht näher beschrieben bzw. erklärt. Eine kurze Anleitung für Änderungen des Log-Level sind in Kapitel **6.4.1** zu finden. Der Logger ist während des Studiums durch den Autor dieser Arbeit entstanden. Für dieses Projekt braucht er nicht näher erläutert werden, da er als Hilfestellung dient und nicht zwangsläufig dazugehört. Man könnte ihn auch komplett aus dem Plugin entfernen.

✓ Nicht-Funktionale Anforderung **N4** erfüllt.

[24] http://www.ibm.com/developerworks/library/os-ecl-cdt1/index.html

6.2.1) Wichtige Klassendiagramme

In diesem Abschnitt werden die wichtigsten Klassen als Diagramme vorgestellt. Als wichtig werden die Kern-Komponenten des Plugins angesehen. Die **Abbildung 2 - Klassendiagramm Rules-Package** zeigt, dass alle implementierten Regeln von der abstrakten Regelklasse erben.

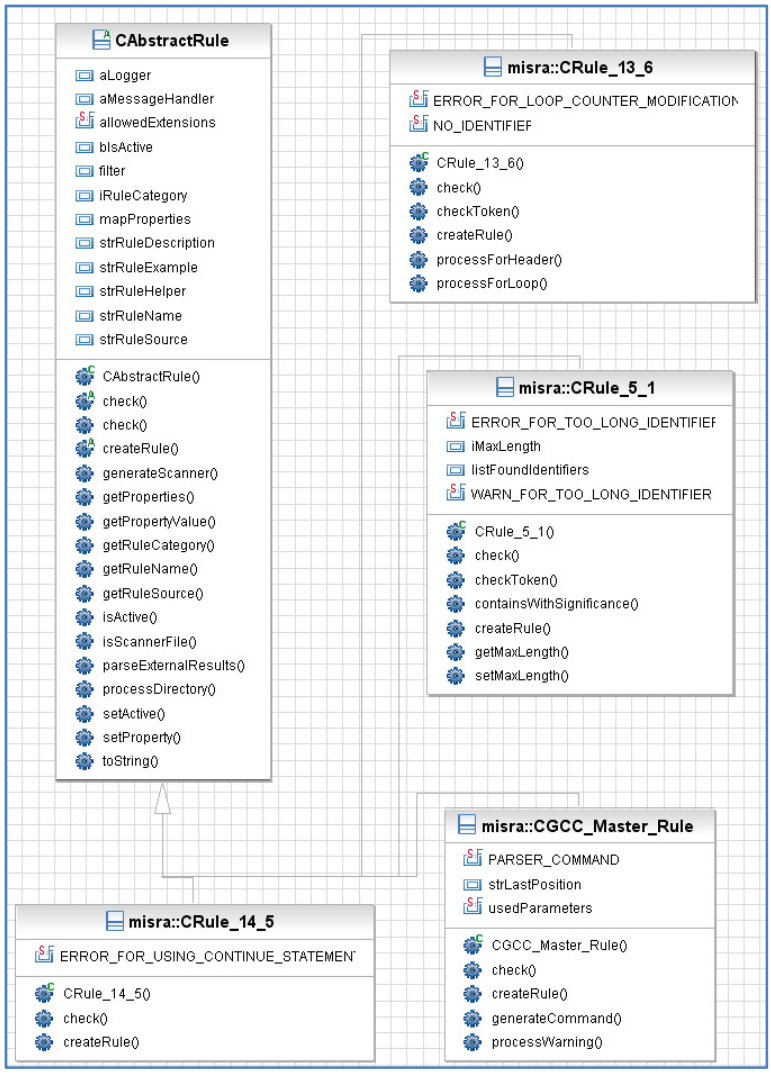


Abbildung 2 - Klassendiagramm Rules-Package

Da die restlichen Klassen keine besonderen Verknüpfungen zueinander haben, werden diese hier nicht aufgeführt. Sie werden in den folgenden Kapiteln noch genauer erläutert in Bezug auf Zusammenspiel.

Das Plugin besteht aus einem Kern, der den Manager für die Regeln enthält. Der Manager beinhaltet alle registrierten Regeln und das Interface zum Ausgeben von Meldungen und dem Erstellen von Markern. Im folgenden werden die einzelnen Pakete näher erläutert.

6.2.2) Interaktions-Paket

Das Interaktions-Paket besteht aus folgenden Modulen:

- Eclipse-Interactor (**CEclipseInteractor** - *Singleton*)
  Diese Klasse bildet das Interface zwischen dem Plugin und der Eclipse-Oberfläche (Kommunikation nur vom Plugin nach Eclipse, nicht andersherum). Hier werden Quellcode Marker erstellt / gelöscht und referenzierte Dateien im Editor geöffnet.
- Meldungs-Fenster (**CMessageViewer**)
  Das Fenster, welches in Eclipse neben der Konsole erscheint (mit Namen RuleCheck) wird durch diese Klasse realisiert. Intern besteht die Ausgabe aus einer Tabelle, die hier definiert wird.
- Und Hilfsklassen für das Meldungs-Fenster, um die Tabelle zu erstellen und zu füllen (dazu gehören **CMessageResultLabelProvider** und **CMessageResultContentProvider**). Mehr zu diesen beiden Klassen findet sich in Kapitel **6.4.3**.

Der Eclipse-Interactor sollte die einzige Klasse sein, die mit Eclipse kommuniziert. In dieser Version des Plugins meldet sich das Meldungs-Fenster noch selber bei Eclipse an und aktualisiert sich auch selbst. In späteren Versionen sollte dies geändert werden. Für eine Übersicht der Kommunikation siehe **Abbildung 4 - Interaktion von Eclipse und RuleCheck**.

Dieses Paket wurde anhand des Metrik-Beispiels [22] erstellt. Zusätzlich wurden noch die Quellcode-Marker hinzugefügt [11].

6.2.3) Logik-Paket

Das Logik-Paket besteht aus folgenden Modulen:

- Plugin-Kern (**CRuleCheckCore** – *Singleton*[25])
  Der Kern wird von den Eintrittspunkten des Plugins angefordert und gestartet. Als Parameter bekommt er eine Datei oder ein Verzeichnis. Er besitzt eine Referenz zum Regel-Manager. In der derzeitigen Version leitet der Plugin-Kern nur die Anforderung des Eintrittspunkts an den Regel-Manager weiter.

[25] Eine Klasse, von der es nur eine Instanz geben darf / kann

- Regel-Manager (**CRuleManager** – *Singleton*)
  *Der Regel-Manager enthält eine Liste aller registrierten Regeln. Diese werden in eine **HashMap** gelegt um schnell über den eindeutigen Namen darauf zugreifen zu können. Er ist in der Lage die Überprüfungen für alle Regeln auf eine Datei oder ein Verzeichnis zu starten. Dies geschieht sequentiell anhand der Reihenfolge der Regeln in der **HashMap**.*
- Meldungs-Manager (**CMessageHandler**)
  Der Manager für die Ausgaben des Plugins beinhaltet eine Liste aller Meldungsobjekte, die in einem gesamten Durchlauf des Plugins generiert wurden. Meldungen kommen hier als einzelne Variablen an, so dass der Meldungs-Manager daraus ein Meldungs-Objekt generieren kann. Nach dem Einfügen einer Meldung wird eine Aktualisierung der Ausgabe in Eclipse ausgelöst.
- Des Weiteren existieren noch ein paar Klassen, die Daten repräsentieren. Dazu gehören die generalisierten Regeln und ein Meldungs-Objekt (**CMessage**).

Mit der Methode *add()* kann im Regel-Manager eine neue Regel hinzugefügt und mit *remove()* gelöscht werden. Der Name der Regel ist hierbei eindeutig zu wählen. Das komplette Interface[26] kann in der JavaDoc-API des Projekts eingesehen werden.

Alle Regeln sind von der Klasse **CAbstractRule** abgeleitet, die das Interface und einige vorgefertigte Funktionen für eine Regel definiert. Jede neue Regel muss von dieser Klasse abstammen. Die Implementierung neuer Regeln wird in Kapitel **6.4.2** behandelt.

Des Weiteren ist in dem Logik-Paket noch ein Message-Handler (**CMessageHandler**) vorhanden, der Warnungen und Fehlermeldungen von Regeln in Bezug auf die Überprüfung des Quellcodes verarbeitet und weiterleitet. Die Meldungen können entweder in der Konsole oder in einem eigenen Fenster in der Eclipse Oberfläche ausgegeben werden. Mit dem Schalter *bConvertWarningToError* wird der Message-Handler angewiesen, alle Warnungen als Fehlermeldungen auszugeben. Vorbereitete Meldungen werden an das Interaktions-Paket geleitet, damit diese in der Eclipse Oberfläche angezeigt werden können.

[26] Schnittstelle

Die folgende Grafik zeigt, wie die Objekte im Plugin gehalten werden. Die RuleCheckActions-Klasse holt sich eine Instanz der RuleCheckCore-Klasse. Hier hingegen gibt es eine Instanz des RuleManagers und der beinhaltet Regeln und einen MessageHandler. Die Struktur ist hierarchisch aufgebaut.

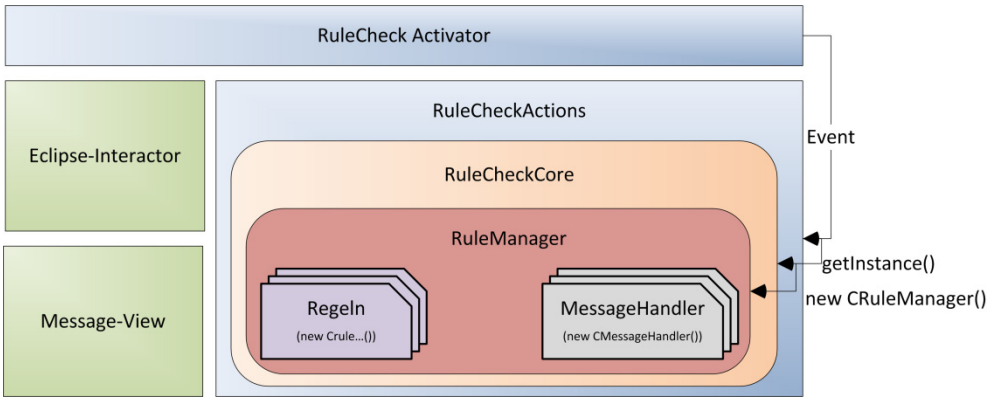


Abbildung 3 - Interne Objekthaltung

Um zu demonstrieren, wie man einen Parsing-Vorgang einer Regel implementieren könnte, siehe folgendes Listing:

```
/**
 * Methode sucht Identifier, die länger sind, als die maximale
 * Signifikanz-Länge und speicher diese zwischen.
 *
 * @param aToken Aktueller Token
 */
private void checkToken( IToken aToken ) {
     String strFunctionName = "checkToken( IToken aToken )";
     aLogger.enter( this.getClass().getName(), "unique", strFunctionName,
                    "Checking token: " + aToken.toString() );

     // Prüfen, ob Token ein Identifier und größer als iMaxLength ist
     if ( IToken.tIDENTIFIER == aToken.getType()
        && iMaxLength < aToken.getLength() ) {
             // Prüfen, ob der reduzierte Token schon existiert
             if ( containsWithSignificance( aToken.toString() ) ) {
                     // Prüfen ob der gesamte Token schon existiert
                     if ( !listFoundIdentifiers.contains( aToken.toString() ) ) {
                             listFoundIdentifiers.add( aToken.toString() );
                             aMessageHandler.addError( aToken.getLineNumber(),
                                     aToken.getFilename(), aToken.toString(),
                                     ERROR_FOR_TOO_LONG_IDENTIFIER,
                                     getRuleSource(), getRuleName() );
                     } else { // Gesamter Token existiert schon
                             aMessageHandler.addWarning( aToken.getLineNumber(),
                                     aToken.getFilename(), aToken.toString(),
                                     WARN_FOR_TOO_LONG_IDENTIFIER + " |
                                     MaxSignificantLength = " + iMaxLength,
                                     getRuleSource(), getRuleName() );
                     }
             } else { // Reduzierter Token exisitert noch nicht
                     listFoundIdentifiers.add( aToken.toString() );
                     aMessageHandler.addWarning( aToken.getLineNumber(),
                                     aToken.getFilename(), aToken.toString(),
                                     WARN_FOR_TOO_LONG_IDENTIFIER + " |
                                     MaxSignificantLength = " + iMaxLength,
                                     getRuleSource(), getRuleName() );
             }
     }

     aLogger.leave( this.getClass().getName(), "unique", strFunctionName, "" );
}
```

Die Methode *containsWithSignificances()* prüft, ob vorher bereits ein Bezeichner gefunden wurde, der dieselbe Zeichenfolge vor der Signifikanzschwelle besitzt (jeder neuer Bezeichner wird gespeichert).

Dieses Paket basiert komplett auf den Überlegungen und Erfahrungen des Autors.

6.2.4) Der C-Scanner

Der Scanner des CDT-Projekts kann nicht einfach so verwendet werden. Er muss herausgefiltert werden, damit er einsatzfähig ist. Um diese Arbeit zu vermeiden, wurde das IBM-Plugin BBCDT benutzt, welches bereits der herausgefilterten Scanner besitzt und nutzbar macht.

Die momentane Implementierung sieht vor, dass jedes Regelobjekt seine eigene Scanner-Instanz besitzt. Nachdem er für eine Datei erstellt wurde, kann die Datei Token-weise verarbeitet werden. Um die Regel zu realisieren muss jede Regel einen eigenen Parser implementieren, der mithilfe des Scanners die Datei überprüft. Bei Regeln, die über mehrere Dateien geprüft werden, muss anhand von passenden Datenstrukturen die nötigen Informationen gespeichert und verwendet werden.

Um den Scanner nutzen zu können muss eine neue Instanz der Klasse **Scanner2** erstellt werden. Folgende Methode aus der Klasse **CAbstractRule** generiert eine Scanner-Instanz:

```
/**
 * Methode generiert einen Scanner für die angegebene Datei
 *
 * @param strFilename Datei, die mit Scanner geprüft werden soll
 * @return Generierter Scanner
 */
protected IScanner generateScanner( String strFilename ) {
    String strFunctionName = "generateScanner( String strFilename )";
    aLogger.enter( this.getClass().getName(), "unique", strFunctionName,
                   "Trying to generate Scanner for file: " + strFilename );

    // Vorarbeit, um den Scanner zu erstellen
    IScannerInfo info                       = new ExtendedScannerInfo();
    ISourceElementRequestor requestor       = new NullSourceElementRequestor();
    ParserMode parserMode                   = ParserMode.COMPLETE_PARSE;
    ParserLanguage language                 = ParserLanguage.C;
    IParserLogService log                   = ParserUtil.getScannerLogService();
    ArrayList<IWorkingCopy> workingCopies= new ArrayList<IWorkingCopy>();

    // Leeren Scanner erstellen
    IScanner aScanner = null;
    try {
           // Scanner generieren
           aScanner = ParserFactory.createScanner( strFilename, info,
                                                   parserMode, language,
                                                   requestor, log,
                                                   workingCopies );
    } catch ( Exception e ) {
           aLogger.exception( this.getClass().getName(), "unique",
                              strFunctionName, e, "Error while generating
                              scanner!" );
```

```
        }

        aLogger.leave( this.getClass().getName(), "unique", strFunctionName, "" );
        return( aScanner );
    }
```

Mit diesem Objekt kann man sich nun alle Token mithilfe der Funktion *nextToken()* liefern lassen, bis die Methode mit einer **EndOfFileException** die Ausführung abbricht, weil das Ende der zu scannenden Datei erreicht wurde. Folgendes Beispiel generiert mithilfe der oben genannten Methode einen Scanner und gibt jedes Token aus der angegebenen Datei in der Konsole aus:

```
// Einen Scanner für die gewünschte Datei erstellen
IScanner sourceScanner = generateScanner( "c:/test.c" );

// Datei durchscannen und parsen
try {
    while ( true ) {
            // Nächsten Token besorgen
            IToken aToken = sourceScanner.nextToken();

            System.out.println( aToken.getFilename() + ":"
                                + aToken.getLineNumber() + " => "
                                + aToken.toString() );
    }
} catch ( EndOfFileException e ) {
    System.out.println( "Ende der Datei erreicht!" );
}
```

Wichtig zu wissen ist, dass der Scanner automatisch die eingefügten Dateien (Header, Quelldateien), mittels dem Schlüsselwort **#include**, mit überprüft. Diese brauchen also nicht nochmal extra geprüft zu werden. In dieser Version des Plugins dies nicht beachtet.

6.2.5) Das Plugin und Eclipse

Das Plugin kommuniziert hauptsächlich mithilfe des EclipseInteractor mit der Eclipse Oberfläche. Einzige Ausnahme ist hier die Message-View. Mit etwas Aufwand kann man die Registrierung und die Aktualisierungen der Message-View auch über den EclipseInteractor laufen lassen. Aus Zeitgründen ist dies nicht geschehen.

Der EclipseInteractor kann Marker in Quellcode Dateien erstellen, sofern diese im aktuellen Workspace liegen. Diese können durch Doppelklick auf die Meldung (im vorhandenen Eclipse-Problem-Fenster) angezeigt werden. Die Funktionalität des Doppelklicks wurde aus Zeitgründen nicht für die Plugin-eigene Ausgabe implementiert. Der EclipseInteractor kann nur alle Marker aus dem aktuellen Workspace löschen.

Eclipse selber kommuniziert nur indirekt mit dem Plugin, indem es eine Methode im Activator aufruft und diese dann beginnt das Plugin zu erstellen. Da es als Singleton implementiert wurde, wird für das Plugin nur beim ersten Startvorgang einmal eine Instanz erstellt. Folgende Grafik zeigt die möglichen Interaktionen zwischen Eclipse und dem RuleCheck Plugin.

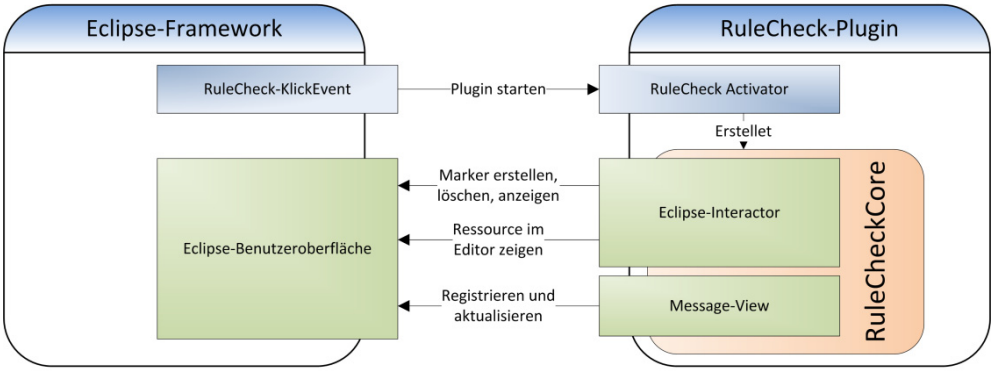


Abbildung 4 - Interaktion von Eclipse und RuleCheck

Hinweis zur Weiterentwicklung: Da das Plugin Open Source ist, wurde in diesem Prototyp vorerst darauf verzichtet eigene extension points einzubauen. Im Moment muss das Plugin selber angepasst und kompiliert werden. Dies betrifft hauptsächlich neue Regeln. Mit entsprechenden extension points könnten Regeln in einem Fremd-Projekt / Plugin erstellt und eingespeist werden.

6.3) Tests

Der Entwicklertest für das Plugin wurde mithilfe des Eclipse-Debuggers[27] und einer selbstgeschriebenen Logging-Klasse durchgeführt. In manchen Bereichen wurde JUnit4[28] eingesetzt. JUnit wurde nur im Anfangsstadium verwendet, da es dem Plugin zu diesem Zeitpunkt noch nicht möglich war, mit der Eclipse-Umgebung zu interagieren.

Die einzelnen implementierten Regeln des Misra-C:2004 Regelwerks wurden durch eigens dafür geschriebene C-Programme verifiziert. Es wurden gezielt Verletzungen der Regeln herbeigeführt. Teilweise wurden ähnliche, aber regelkonforme Anweisungen eingebaut.

[27] Programm zum Schrittweisen durchlaufen eines Programms zur Ausführungszeit
[28] Framework zum automatisierten Testen unter Java

Im Endstadium des Plugins wurde ein Workspace angelegt, in dem ein C-Projekt existierte, welches alle Regeltests beinhaltete. Dort war es dann möglich, alle Regeln sowohl auf das gesamte Projekt als auch auf einzelne Dateien anzuwenden.

Der benutzte Scanner wurde nicht extra getestet, da er dem offiziellen CDT-Projekt[29] entstammt und dort vermutlich ausreichend getestet wurde.

Die folgenden Abschnitte verdeutlichen die Vorgehensweise beim Testen des Plugins.

6.3.1) JUnit

Im ersten Stadium der Implementierung fungierte das Programm noch nicht als Plugin, sondern wurde als eigenständiges Programm benutzt. In dieser Zeit wurden die Regelobjekte und der RuleManager implementiert und getestet.

Um die implementierten Regeln zu testen, wurden Instanzen der Regeln erstellt und auf die jeweiligen C-Beispiel-Dateien angewendet, die für diese Regeln implementiert wurden. Anschließend wurden von Hand die Ausgaben aus dem Log-File bzw. aus der Konsole mit den Stellen im Beispiel-Code verglichen. Folgender Code startete den Test:

```
@Test
public void testRuleWithSourceAndHeader() {
     String strSourceFile = "";
     // ----------------------------------------
     aRule  = new CRule_5_1();
     System.out.println( "=========== Testing rule " + aRule.getStrRuleSource()
                    + " - " + aRule.getRuleName() + "("
                    + CMisra.getNameByID(aRule.getRuleCategory()) + ")"
                    + " ===========" );
     strSourceFile= "I:/Bachelorarbeit/Implementierung/1 - Erweiterter
                     Prototyp/TestCases/rule_5_1.c";
     aRule.check(strSourceFile);
     // ----------------------------------------
     aRule  = new CRule_13_6();
     System.out.println( "=========== Testing rule " + aRule.getStrRuleSource()
                    + " - " + aRule.getRuleName() + "("
                    + CMisra.getNameByID(aRule.getRuleCategory()) + ")"
                    + " ===========" );
     strSourceFile= "I:/Bachelorarbeit/Implementierung/1 - Erweiterter
                     Prototyp/TestCases/rule_13_6.c";
     aRule.check(strSourceFile);
     // ----------------------------------------
     aRule  = new CRule_14_5();
     System.out.println( "=========== Testing rule " + aRule.getStrRuleSource()
                    + " - " + aRule.getRuleName() + "("
                    + CMisra.getNameByID(aRule.getRuleCategory()) + ")"
                    + " ===========" );
     strSourceFile= "I:/Bachelorarbeit/Implementierung/1 - Erweiterter
                     Prototyp/TestCases/rule_14_5.c";
     aRule.check(strSourceFile);
     // ----------------------------------------
     aRule  = new CRules_gcc();
```

[29] http://wiki.eclipse.org/index.php/CDT

```
System.out.println( "=========== Testing rule " + aRule.getStrRuleSource()
                    + " - " + aRule.getRuleName() + "("
                    + CMisra.getNameByID(aRule.getRuleCategory()) + ")"
                    + " ===========" );
    strSourceFile= "I:/Bachelorarbeit/Implementierung/1 - Erweiterter
                    Prototyp/TestCases/rules_gcc.c";
    aRule.check(strSourceFile);
    // ----------------------------------------
}
```

Der RuleManager wurde im nächsten Schritt auf dieselbe Weise getestet. Die Überprüfungen wurden auf das gesamte Verzeichnis, welches die implementierten Beispiel-Codes enthielt, angewendet. Es wurde dann geprüft, ob alle Ausgaben, die im vorherigen Test gemacht wurden, auch wieder auftreten. Der Vergleich wurde auch hier per Hand durchgeführt, da die Regeln und der Manager die Meldungen nicht als Resultate liefern können. Der Test des RuleManagers wurde wie folgt gestartet:

```
@Test
public void testRuleManager() {
    String strSourceDirectory = "I:/Bachelorarbeit/Implementierung/1 -
                                 Erweiterter Prototyp/TestCases/";
    // ----------------------------------------
    theRuleManager = new CRuleManager();
    theRuleManager.startAllRules( strSourceDirectory, true, true );
}
```

Nachdem die implementierten Regeln richtige Ergebnisse lieferten, wurden die Schnittstellen zu Eclipse (siehe Kapitel **6.2.4**) erstellt und das Programm nur noch als Plugin ausgeführt. Ab hier wurden Tests nur noch mithilfe von Debugging und Tracing[30] durchgeführt.

6.3.2) Debugging, Tracing und empirisches Testen

Im Log-File des selbstgeschriebenen Loggers werden alle relevanten Ausgaben des Plugins festgehalten. Je nach Ausgabe-Level des Loggers werden Fehler (*error*, *fatal*, *exception*), Informationen (*info*) oder Daten über den Programmfluss (*trace*) in die Datei geschrieben. Für mehr Informationen zu den Level siehe Kapitel **6.4.1**. Im Anhang befindet sich ein Beispiel Log-File[31], welches in der Endphase des Plugins generiert wurde.

[30] Ablaufverfolgung, hier speziell Ablauf des Programmflusses
[31] Datei: „20070916 - Beispiellogfile.log“

Das Debugging und Tracing wurde zusammen mit empirischen Tests (mögliche Programmabläufe) kombiniert. Die normalen Programmabläufe wurden durchgespielt und anschließend im Log-File verifiziert, ob das Plugin das macht, was es soll. Ist dies nicht der Fall gewesen oder waren Auffälligkeiten (Aufruf von Methoden, Werte etc.) / Fehlverhalten des Programms vorhanden, wurde mithilfe des Debuggers mögliche Ursachen gesucht. Nach Auffindung der Ursache wurde diese beseitigt und das Plugin anschließend erneut getestet.

Der Scanner wurde durch Tracing und empirisches Testen kurz auf seine Funktionstüchtigkeit geprüft. Dies geschah durch Scannen einer Datei und Ausgabe der Tokens in der Konsole (siehe Minimal-Beispiel aus Kapitel **6.2.4**). Anschließend wurde die Ausgabe von Hand mit dem gescannten Quellcode verglichen.

Im Folgendem wird genauer auf die Implementierung der C-Beispiel-Codes eingegangen, die die Funktionalität der implementierten Regeln des Misra-C:2004 Standards testen sollen.

✓ Funktionale Anforderung **F4** erfüllt.

6.3.3) Testen der implementierten Regel 5.1

Um die Regel 5.1 zu testen, werden mehrere Bezeichner erstellt, die verschiedene Längen haben. Drei Fälle werden bei dem folgenden Test überprüft:

- Bezeichner die länger sind als 31 Zeichen und sich in den ersten 31 Zeichen unterscheiden. *Es sollte eine Warnung pro Aufkommen der einzelnen Bezeichner geben.*
- Bezeichner die länger sind als 31 Zeichen und sich nicht in den ersten 31 Zeichen unterscheiden. *Es sollte eine Warnung pro Aufkommen der einzelnen Bezeichner geben und einen Fehler, sobald ein weiterer Bezeichner mit denselben ersten 31 Zeichen gefunden wird.*
- Bezeichner die kürzer als 31 Zeichen sind und sich minimal unterscheiden. *Sollten ignoriert werden.*
- Bezeichner der genau 31 Zeichen lang ist und Bezeichner, der mehr als 31 Zeichen lang ist und dieselben ersten 31 Zeichen haben. *Es sollte eine Warnung pro Aufkommen der einzelnen Bezeichner geben und einen Fehler, sobald ein weiterer Bezeichner mit denselben ersten 31 Zeichen gefunden wird.*

Der folgende C-Quellcode soll die oben genannten Fälle abdecken.

Datei: *rule_5_1.h*

```
#ifndef RULE_5_1_H
#define RULE_5_1_H

// Diese zwei sollten für die Ueberpruefung als gleich gesehen werden
int ich_bin_ein_langer_identifier_mit_der_laenge_47              = 47;
int ich_bin_ein_langer_identifier_mit_der_noch_mehr_laenge_57    = 57;

// Diese zwei sollten problemlos unterschieden werden koennen
char ich_bin_auch_ganz_lang                                      = 'a';
char ich_bin_nicht_ganz_lang                                     = 'b';

// Grenzfall, aber beide sollten als gleich gesehen werden
long genau_31_zeichen_lang_werde_ich                             = 31;
long genau_31_zeichen_lang_werde_ich_plus_mehr                   = 31 + 10;

// Grenzfall, aber sollte nicht als gleich gesehen werden
long genau_31_zeichen_lang_werde_ick_plus_mehr                   = 31 + 10;

#endif
```

Datei: *rule_5_1.c*

```
#include <stdio.h>
#include "rule_5_1.h"


int main( int argc, char* argv[] ) {
    int iRetCode = 0;

    printf( "\nErgebnis: %d" ,
          ich_bin_ein_langer_identifier_mit_der_laenge_47
        + ich_bin_ein_langer_identifier_mit_der_noch_mehr_laenge_57 );

    if ( genau_31_zeichen_lang_werde_ich
       < genau_31_zeichen_lang_werde_ich_plus_mehr ) {
          ich_bin_ein_langer_identifier_mit_der_laenge_47 =
          genau_31_zeichen_lang_werde_ick_plus_mehr;
    }

    return( iRetCode );
  }
```

Wenn das Plugin den oben aufgeführten C-Quellcode überprüft bekommt man die folgenden Ausgaben:

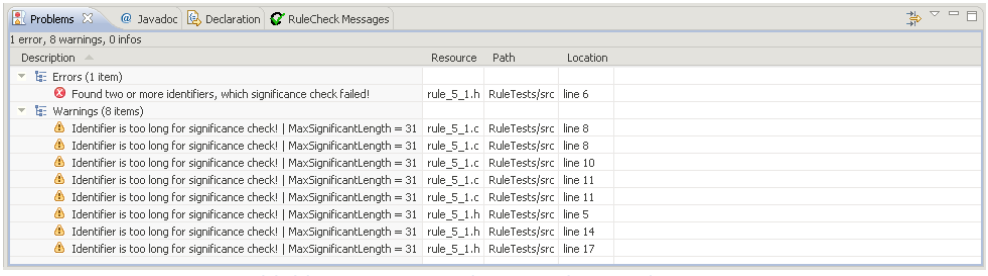
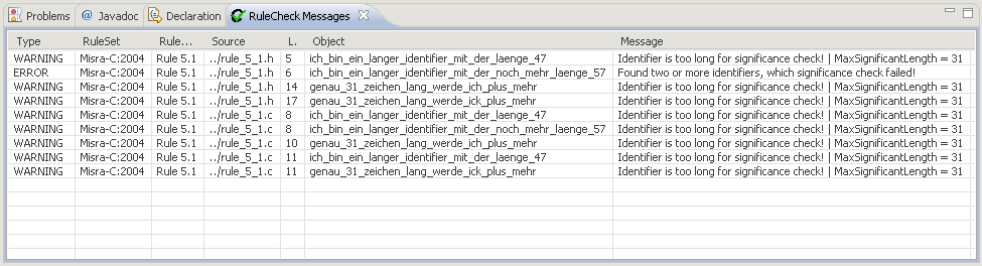


Abbildung 5 - Testergebnis Regel 5.1 - Eclipse Fenster



Problems | Javadoc | Declaration | RuleCheck Messages

Type	RuleSet	Rule...	Source	L.	Object	Message
WARNING	Misra-C:2004	Rule 5.1	../rule_5_1.h	5	ich_bin_ein_langer_identifier_mit_der_laenge_47	Identifier is too long for significance check! | MaxSignificantLength = 31
ERROR	Misra-C:2004	Rule 5.1	../rule_5_1.h	6	ich_bin_ein_langer_identifier_mit_der_noch_mehr_laenge_57	Found two or more identifiers, which significance check failed!
WARNING	Misra-C:2004	Rule 5.1	../rule_5_1.h	14	genau_31_zeichen_lang_werde_ich_plus_mehr	Identifier is too long for significance check! | MaxSignificantLength = 31
WARNING	Misra-C:2004	Rule 5.1	../rule_5_1.h	17	genau_31_zeichen_lang_werde_ick_plus_mehr	Identifier is too long for significance check! | MaxSignificantLength = 31
WARNING	Misra-C:2004	Rule 5.1	../rule_5_1.c	8	ich_bin_ein_langer_identifier_mit_der_laenge_47	Identifier is too long for significance check! | MaxSignificantLength = 31
WARNING	Misra-C:2004	Rule 5.1	../rule_5_1.c	8	ich_bin_ein_langer_identifier_mit_der_noch_mehr_laenge_57	Identifier is too long for significance check! | MaxSignificantLength = 31
WARNING	Misra-C:2004	Rule 5.1	../rule_5_1.c	10	genau_31_zeichen_lang_werde_ich_plus_mehr	Identifier is too long for significance check! | MaxSignificantLength = 31
WARNING	Misra-C:2004	Rule 5.1	../rule_5_1.c	11	ich_bin_ein_langer_identifier_mit_der_laenge_47	Identifier is too long for significance check! | MaxSignificantLength = 31
WARNING	Misra-C:2004	Rule 5.1	../rule_5_1.c	11	genau_31_zeichen_lang_werde_ick_plus_mehr	Identifier is too long for significance check! | MaxSignificantLength = 31

Abbildung 6 - Testergebnis Regel 5.1 - RuleCheck Fenster

Die erste Abbildung zeigt die Marker, die von dem Plugin gesetzt wurden. In der zweiten Abbildung wird die Ausgabe des Plugins gezeigt. Das Ergebnis des Tests zeigt, dass alle erwähnten Fälle richtig erkannt wurden.

- ✓ Funktionale Anforderung **F1** erfüllt.
- ✓ Funktionale Anforderung **F2** erfüllt.

6.3.4) Testen der implementierten Regel 13.6

Um die Regel 13.6 zu testen, werden alle Modifikationen der Schleifenzähler-Variable die möglich sind durchgeführt. Dabei werden die Operanden für eine Manipulation der Variable aufgeführt. Der folgende C-Quellcode soll die Modifikationen abdecken.

Datei: ***rule_13_6.c***

```c
#include <stdio.h>

int main( int argc, char* argv[] ) {
    int iRetCode = 0;

    int flag = 1;

    for ( int i = 0; (i<5) && (1==flag); i++ ){ // Normal nicht ok
        flag = i;                               // Zulaessig, Verkuerzung

        i = i + 3;                              // Nicht zulaessig
        i--;                                    // Nicht zulaessig
        i++;                                    // Nicht zulaessig
        i*=1;                                   // Nicht zulaessig
        i/=2;                                   // Nicht zulaessig
        i-=2;                                   // Nicht zulaessig
        i+=2;                                   // Nicht zulaessig
    }

    i = -1;                                     // ok, nicht in Schleife

    for ( i = 0; i < 10; i++ )
       i++;                                     // Nicht zulaessig

    return( iRetCode );
}
```

Wenn das Plugin den oben aufgeführten C-Quellcode überprüft bekommt man die folgenden Ausgaben:

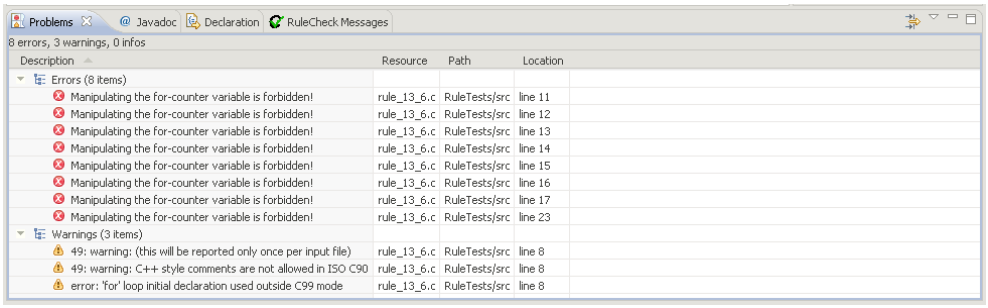


Abbildung 7 - Testergebnis Regel 13.6 - Eclipse Fenster

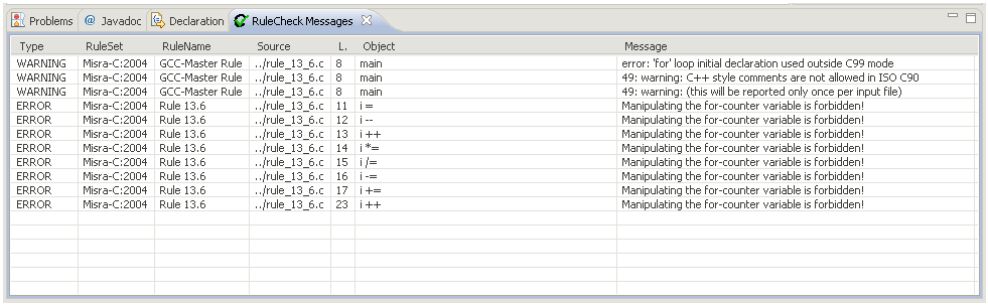


Abbildung 8 - Testergebnis Regel 13.6 - RuleCheck Fenster

Die erste Abbildung zeigt die Marker, die von dem Plugin gesetzt wurden. In der zweiten Abbildung wird die Ausgabe des Plugins gezeigt. Das Ergebnis des Tests zeigt, dass alle getesteten Fälle richtig erkannt wurden.

- ✓ Funktionale Anforderung **F1** erfüllt.
- ✓ Funktionale Anforderung **F2** erfüllt.

6.3.5) Testen der implementierten Regel 14.5

Um die Regel 14.5 zu testen, wird an einer beliebigen Stelle ein **continue** eingefügt. In diesem Beispiel wird es innerhalb einer **for**-Schleife eingefügt. Der folgende C-Quellcode beinhaltet die Anweisung:

Datei: ***rule_14_5.c***

```c
#include <stdio.h>

int main( int argc, char* argv[] ) {
    int iRetCode = 0;
    int i;

    for ( i = 0; i < 5; i++) {
        continue;
        int j = i + 3;                              // Nicht zulaessig
    }

    return( iRetCode );
}
```

Wenn das Plugin den oben aufgeführten C-Quellcode überprüft bekommt man die folgenden Ausgaben:

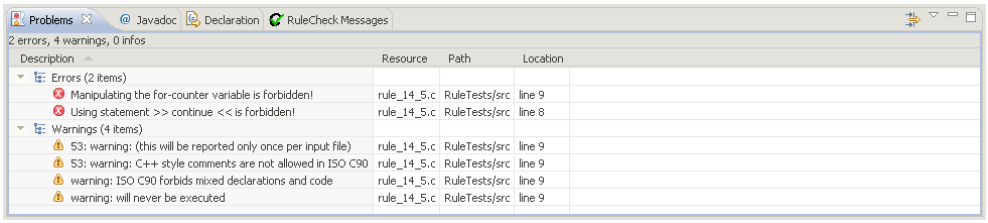


Abbildung 9 - Testergebnis Regel 14.5 - Eclipse Fenster

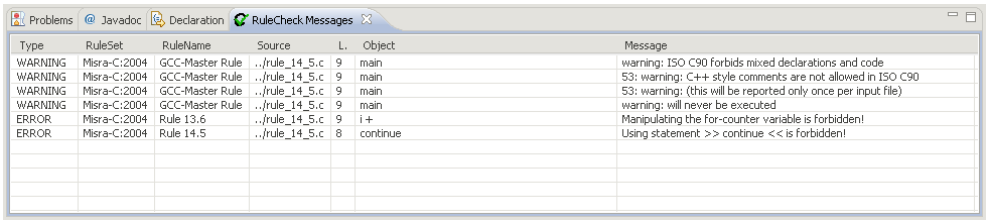


Abbildung 10 - Testergebnis Regel 14.5 - RuleCheck Fenster

Die erste Abbildung zeigt die Marker, die von dem Plugin gesetzt wurden. In der zweiten Abbildung wird die Ausgabe des Plugins gezeigt. Das Ergebnis des Tests zeigt, dass alle getesteten Fälle richtig erkannt wurden.

- ✓ Funktionale Anforderung **F1** erfüllt.
- ✓ Funktionale Anforderung **F2** erfüllt.

6.3.6) Testen der Abdeckung durch die GCC

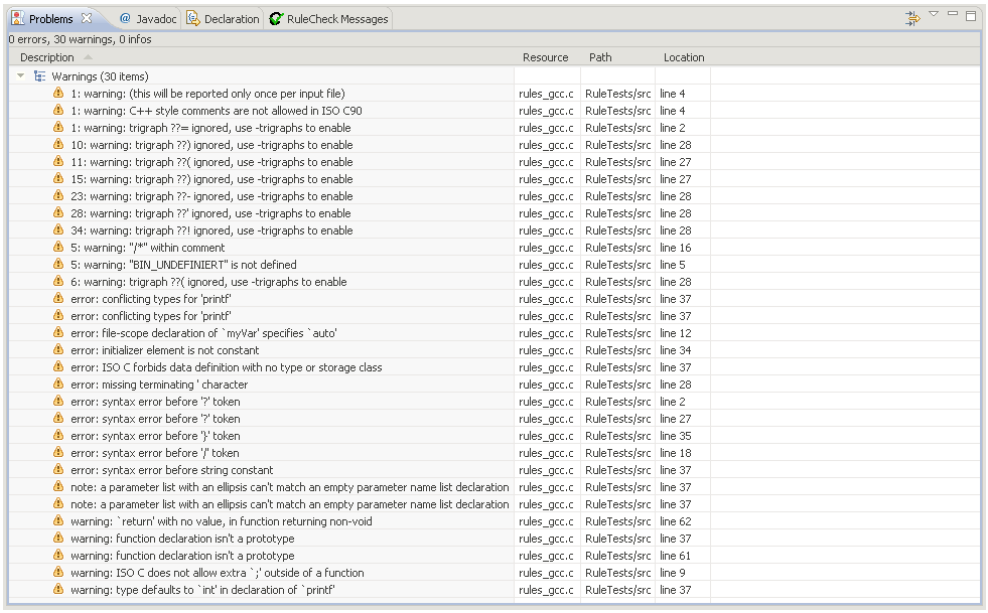
Um die Regeln zu testen, die die GCC abdecken, wurden Verstöße gegen diese Regeln in die folgende Datei eingebaut. Für manche Regeln wurden zusätzlich ähnliche Konstrukte eingebaut, die den Regeln entsprechen.

Datei: ***rules_gcc.c***

```
#include <stdio.h>
??=include <time.h>                          /* # */

// Nicht zulässig nach 19.11
#if BIN_UNDEFINIERT
    int xyz = 3;
#endif

int a[2][2] = { 0, 1, 2, 3 };          // Nicht vollständig umklammert 9.2
int b[2][2] = { { 0, 1 }, { 2, 3 } }; // Richtig!

auto int myVar;
float aFuntion( int a, int b );
/*
   Meine kleine Main-Methode
    /* int a = 3;            */ /* Nicht zulässig 2.3 */

*/
int main( int argc, char* argv[] ) {
    int iRetCode = 0;
```

```
    int i;

    float fVar1 = 1.03;
    float fVar2 = 1.03000001;

    // Folgende zwei Zeilen: Nicht zulässig nach 4.2
    char n??(5??);                          /* [ and ]    */
    n??(4??) = '0' - (??-0 ??' 1 ??! 2); /* ~, ^ and | */

    // ----------------------

    for ( i = 0; i < 5; i++) {
        continue;
        int j = i + 3;                  // Nicht zulaessig nach 1.1 und 14.1
    }

    printf( "%d", myVar );              // Nicht zulässig nach 9.1

    if ( i != i ) {
        i = 3;                          // Nie erreichbarer Code 14.1
    } else {
        i = myFuntion();
    }

    // Nicht zulässig nach 13.3
    if ( fVar1 == fVar2 ) {
        printf( "Sind gleich!" );
    }

    // Nicht zulässig nach 15.3
    switch( i ) {
        case 1: printf( "1" ); break;
        case 3: printf( "3" ); break;
        case 12: printf( "12" ); break;
    }

    return( iRetCode );
}

// Nicht zulässig nach 16.8 und ..
int myFunction() {
    return;
}

// Nicht zulässig nach 16.3 und 16.8
float aFuntion( a, b ) {
    printf( "%d, %d", a, b );
}
```

Wenn das Plugin den oben aufgeführten C-Quellcode überprüft bekommt man die folgenden Ausgaben:

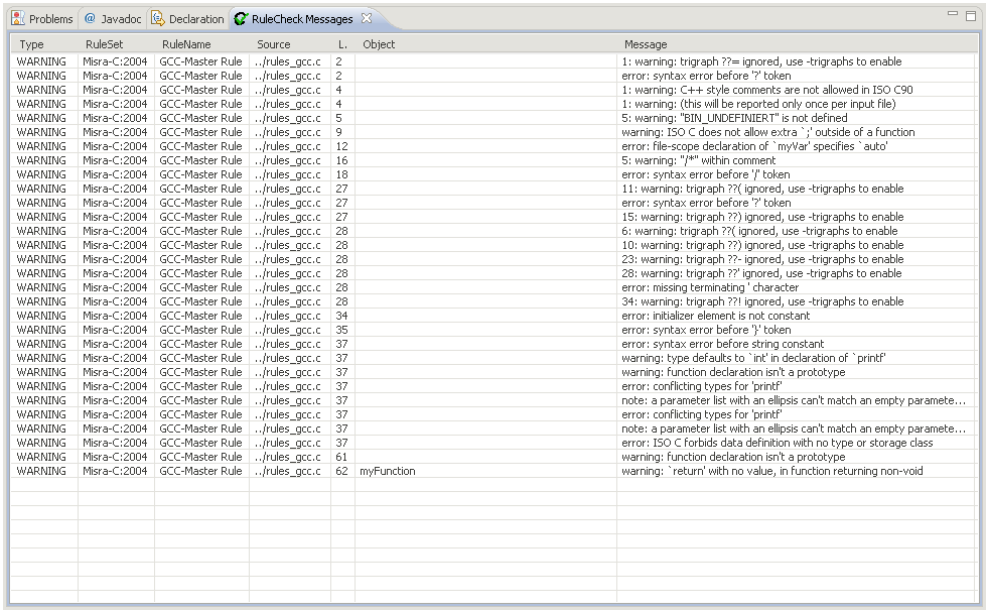


Problems | Javadoc | Declaration | RuleCheck Messages

0 errors, 30 warnings, 0 infos

Description	Resource	Path	Location
Warnings (30 items)			
1: warning: (this will be reported only once per input file)	rules_gcc.c	RuleTests/src	line 4
1: warning: C++ style comments are not allowed in ISO C90	rules_gcc.c	RuleTests/src	line 4
1: warning: trigraph ??= ignored, use -trigraphs to enable	rules_gcc.c	RuleTests/src	line 2
10: warning: trigraph ??) ignored, use -trigraphs to enable	rules_gcc.c	RuleTests/src	line 28
11: warning: trigraph ??(ignored, use -trigraphs to enable	rules_gcc.c	RuleTests/src	line 27
15: warning: trigraph ??) ignored, use -trigraphs to enable	rules_gcc.c	RuleTests/src	line 27
23: warning: trigraph ??- ignored, use -trigraphs to enable	rules_gcc.c	RuleTests/src	line 28
28: warning: trigraph ??' ignored, use -trigraphs to enable	rules_gcc.c	RuleTests/src	line 28
34: warning: trigraph ??! ignored, use -trigraphs to enable	rules_gcc.c	RuleTests/src	line 28
5: warning: "/*" within comment	rules_gcc.c	RuleTests/src	line 16
5: warning: "BIN_UNDEFINIERT" is not defined	rules_gcc.c	RuleTests/src	line 5
6: warning: trigraph ??(ignored, use -trigraphs to enable	rules_gcc.c	RuleTests/src	line 28
error: conflicting types for 'printf'	rules_gcc.c	RuleTests/src	line 37
error: conflicting types for 'printf'	rules_gcc.c	RuleTests/src	line 37
error: file-scope declaration of `myVar' specifies `auto'	rules_gcc.c	RuleTests/src	line 12
error: initializer element is not constant	rules_gcc.c	RuleTests/src	line 34
error: ISO C forbids data definition with no type or storage class	rules_gcc.c	RuleTests/src	line 37
error: missing terminating ' character	rules_gcc.c	RuleTests/src	line 28
error: syntax error before '?' token	rules_gcc.c	RuleTests/src	line 2
error: syntax error before '?' token	rules_gcc.c	RuleTests/src	line 27
error: syntax error before '}' token	rules_gcc.c	RuleTests/src	line 35
error: syntax error before '/' token	rules_gcc.c	RuleTests/src	line 18
error: syntax error before string constant	rules_gcc.c	RuleTests/src	line 37
note: a parameter list with an ellipsis can't match an empty parameter name list declaration	rules_gcc.c	RuleTests/src	line 37
note: a parameter list with an ellipsis can't match an empty parameter name list declaration	rules_gcc.c	RuleTests/src	line 37
warning: `return' with no value, in function returning non-void	rules_gcc.c	RuleTests/src	line 62
warning: function declaration isn't a prototype	rules_gcc.c	RuleTests/src	line 37
warning: function declaration isn't a prototype	rules_gcc.c	RuleTests/src	line 61
warning: ISO C does not allow extra `;' outside of a function	rules_gcc.c	RuleTests/src	line 9
warning: type defaults to `int' in declaration of `printf'	rules_gcc.c	RuleTests/src	line 37

Abbildung 11 - Testergebnis GCC Regeln - Eclipse Fenster



Problems | Javadoc | Declaration | RuleCheck Messages

Type	RuleSet	RuleName	Source	L.	Object	Message
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	2		1: warning: trigraph ??= ignored, use -trigraphs to enable
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	2		error: syntax error before '?' token
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	4		1: warning: C++ style comments are not allowed in ISO C90
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	4		1: warning: (this will be reported only once per input file)
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	5		5: warning: "BIN_UNDEFINIERT" is not defined
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	9		warning: ISO C does not allow extra `;' outside of a function
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	12		error: file-scope declaration of `myVar' specifies `auto'
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	16		5: warning: "/*" within comment
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	18		error: syntax error before '/' token
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	27		11: warning: trigraph ??(ignored, use -trigraphs to enable
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	27		error: syntax error before '?' token
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	27		15: warning: trigraph ??) ignored, use -trigraphs to enable
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	28		6: warning: trigraph ??(ignored, use -trigraphs to enable
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	28		10: warning: trigraph ??) ignored, use -trigraphs to enable
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	28		23: warning: trigraph ??- ignored, use -trigraphs to enable
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	28		28: warning: trigraph ??' ignored, use -trigraphs to enable
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	28		error: missing terminating ' character
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	28		34: warning: trigraph ??! ignored, use -trigraphs to enable
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	34		error: initializer element is not constant
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	35		error: syntax error before '}' token
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	37		error: syntax error before string constant
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	37		warning: type defaults to `int' in declaration of `printf'
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	37		warning: function declaration isn't a prototype
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	37		error: conflicting types for 'printf'
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	37		note: a parameter list with an ellipsis can't match an empty paramete...
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	37		error: conflicting types for 'printf'
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	37		note: a parameter list with an ellipsis can't match an empty paramete...
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	37		error: ISO C forbids data definition with no type or storage class
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	61		warning: function declaration isn't a prototype
WARNING	Misra-C:2004	GCC-Master Rule	../rules_gcc.c	62	myFunction	warning: `return' with no value, in function returning non-void

Abbildung 12 - Testergebnis GCC Regeln - RuleCheck Fenster

Die erste Abbildung zeigt die Marker, die von dem Plugin gesetzt wurden. In der zweiten Abbildung wird die Ausgabe des Plugins gezeigt. Das Ergebnis des Tests zeigt, dass alle getesteten Fälle richtig erkannt wurden.

- ✓ Funktionale Anforderung **F1** erfüllt.
- ✓ Funktionale Anforderung **F2** erfüllt.
- ✓ Funktionale Anforderung **F3** erfüllt.

6.3.7) Geschwindigkeitstest

Abschließend wird ein Blick auf die Geschwindigkeit des Plugins während der Überprüfung geworfen. Die Geschwindigkeit ist für die exemplarische Implementierung unerheblich aber dennoch interessant. Daher wird ein größeres Projekt mithilfe der bereits implementierten Regeln durchlaufen. Das Projekt setzt sich wie folgt zusammen:

- 223 Dateien, bestehend aus Header- und Code-Dateien
- 73.585 Zeilen Gesamt
- 41.360 Zeilen Code
- 9.461 Leerzeilen
- 22.764 Zeilen Kommentare

Der gesamte Durchgang dauerte 01:03:53 (64 Minuten), entspricht ca. 19,16 Sekunden pro Zeile bzw. 0,052 Zeilen pro Sekunde. Für einen Durchlauf „mal eben“ während der Implementierung ist die Ausführungszeit (bei ähnlichen Projektgrößen und ähnlicher Regelauswahl) viel zu hoch. Für einen Nightly-Build könnte man diese Zeit notfalls noch vertreten. Es wurden 173 Fehler und 2.506 Warnungen in dem Projekt gefunden, wobei nur wenige Regeln des Misra-C:2004 Standards geprüft wurden.

Als Gegenüberstellung dient das Projekt mit den Testdateien für die Regeln aus den vorigen Abschnitten. Das Projekt setzt sich wie folgt zusammen:

- 5 Dateien, bestehend aus Header- und Code-Dateien
- 153 Zeilen Gesamt
- 92 Zeilen Code
- 47 Leerzeilen
- 14 Zeilen Kommentare

Der gesamte Durchgang dauerte 00:00:07 (sieben Sekunden), entspricht ca. 0,046 Sekunden pro Zeile bzw. 22,86 Zeilen pro Sekunde. Um kleine Projekte oder ein aktuelles Modul während der Implementierung überprüfen, ist die Geschwindigkeit akzeptabel.

Bei der momentanen Implementierung steigt die Ausführungszeit der Überprüfungen exponentiell mit der Anzahl der Regeln und der Anzahl von Dateien. Dabei ist weder der Scanner noch das Parsing der einzelnen Regeln schuld. Der Grund für diese Steigerung liegt in der Kombination der Regeln mit den Dateien. Mögliche Ansätze um die Geschwindigkeit bei vielen Regeln und Dateien zu steigern befinden sich in Kapitel **7.3.2**.

Getestet wurde auf einem Dual-Core Rechner mit 2,13 GHz pro Kern, 2 GB DDR-2 RAM, SATA II Festplatte. Während der Tests lag die durchschnittliche Last eines Kerns bei 40% Auslastung. Die Festplattenzugriffe waren sehr gering.

✓ Nicht-Funktionale Anforderung **N3** erfüllt.

6.4) Erweitern / Anpassen des Plugins

Im Anhang dieser Arbeit befindet sich eine vollständige JavaDoc Dokumentation mit allen Klassen und deren Objekten. Es wurden alle aufgeführt, nicht nur die **public**-Schnittstellen. Da das Plugin unter der GPL geführt wird, darf es erweitert und oder geändert werden.

In diesem Kapitel wird beschrieben, wie man neue Regeln hinzufügen kann, wie der Logger wieder auf höhere Ausgabe-Level gesetzt wird und wie die Ausgabe im „RuleCheck Messages“-Fenster angepasst werden kann.

6.4.1) Logger-Ausgaben-Filter

Es gibt sieben Log-Level, die die Ausgaben des Loggers steuern können. Sie können getrennt für das Log-File und für die Konsole eingestellt werden. Die Level sind numerisch geordnet, so dass immer alle Level, die größer oder gleich dem angegebenen Level sind, ausgegeben werden. Die kleineren werden ignoriert und verworfen.

In der Klasse **Logger** muss im Default-Konstruktor der gewünschte Level in den folgenden beiden Zeilen eingetragen werden:

```
logLevelConsOutput = eLogLevel.NONE;          // Log-Level der Konsole
logLevelFileOutput = eLogLevel.EXCEPTION;     // Log-Level der Log-Datei
```

Die Log-Datei findet sich unter *C:\rulecheck.log*. Ändern kann man die Position in der Klasse **FileOperations**. Dort muss folgende Konstante auf den gewünschten Speicherort gesetzt werden:

```
public static final String LOG_FILE = "c:/rulecheck.log";
```

Log-Level, die gewählt werden können, sind in **Tabelle 7 - Logger Level** aufgelistet.

Wertigkeit	Level	Beschreibung
0	None	Es wird nichts geloggt.
1	Fatal	Logausgaben, die fatale Fehlermeldungen enthalten
2	Exception	Logausgaben, die Exceptions sind
3	Error	Zur Laufzeit aufgetretene Fehler
4	Info	Infomeldungen
5	Debug	Debugmeldungen
6	Trace	Es wird alles geloggt

Tabelle 7 - Logger Level

6.4.2) Hinzufügen von neuen Regeln

Die Regeln sind im Package **org.bachelor.rulecheck.rules** in Regelwerke aufgeteilt. Beispielsweise werden Regeln des Misra-C:2004 Standards in dem Package **org.bachelor.rulecheck.rules.misra** untergebracht. Wenn das Ziel-Package gewählt wurde, muss eine neue Klasse erstellt werden, die von der abstrakten Klasse **CAbstractRule** erbt. Dort müssen nur zwei Methoden implementiert werden, um eine fertige Regel für C-Code zu haben. Wenn eine andere Sprache geparst werden soll, muss nur der Scanner ausgetauscht werden. Die Methode *generateScanner()* müsste dann überschrieben werden.

Zusätzlich müssten noch im RuleManager die neuen Datei-Erweiterungen aufgenommen werden, damit die entsprechenden Dateien nicht ausgeblendet werden. Des Weiteren muss ein Dateierweiterungsfilter in die abstrakte Regelklasse eingebaut werden, damit nicht aus Versehen versucht wird, Dateien zu prüfen, die in einer anderen Sprache geschrieben wurden. Dies könnte über eine **HashMap** geschehen, die als Schlüssel die Sprache und als Wert ein Array mit den Dateierweiterungen beinhaltet.

Zu implementieren sind die beiden Methoden *createRule()* (sie dient der Erstellung / Initialisierung einer Regel) und die Methode *check(String strSourceName)* (startet bzw. beinhaltet den Parse-Vorgang).

In der check-Methode muss ein Scanner erstellt und anschließend ein Parsing-Vorgang implementiert werden, der die Überprüfung der gewünschten Regel durchführt. Als Minimalbeispiel könnte die Methode wie folgt aussehen:

```
// Einen Scanner für die gewünschte Datei erstellen
IScanner sourceScanner = generateScanner( strSourceName );

// Datei durchscannen und parsen
try {
    while ( true ) {
            checkToken( sourceScanner.nextToken() );
    }
} catch ( EndOfFileException e ) {
    // EOF reached
} catch( Exception ex ) {
    // Fehler
}
```

In der Methode *checkToken()* würde dann der eigentliche Parsing-Vorgang vorgenommen werden.

Zum Schluss muss die Regel noch im Regel-Manager registriert werden. Hierfür wird in der Klasse **CRuleManager** in der Methode *initialize()* eine Instanz der Regel hinzugefügt:

```
this.add( new <Neue-Regel-Klasse>( aMessageHandler ) );
```

Nun ist die neue Regel registriert und wird beim Überprüfen von einer Datei oder dem gesamten Projekt mit ausgeführt.

Es ist auch möglich mehrere Regeln in einem Objekt unterzubringen. Dafür muss nur der Parse-Vorgang entsprechend programmiert werden. Um eine einheitliche und übersichtliche Struktur zu bekommen, sollte aber möglichst jede Regel einzeln implementiert werden.

6.4.3) Plugin Ausgaben anpassen

Um die Ausgaben des Plugins im Fenster „RuleCheck Messages“ anzupassen, müssen einige Klassen angepasst werden. Für dieses Beispiel wird davon ausgegangen, dass ein weiteres Attribut der Message hinzugefügt wird. Andere Anpassungen der Ausgabe sollten dann leicht erarbeitet werden können.

Zunächst muss das Attribut in der Klasse **CMessage** hinzugefügt und eine getter-Methode geschrieben werden. Das Attribut muss dann beim Erstellen gefüllt werden, da es, sobald es einmal erstellt wurde, nicht mehr geändert werden soll.

Als nächstes muss in der Methode *createPartControl()* in der Klasse **CMessageViewer** eine Spalte hinzugefügt werden, die später dieses Attribut aufnehmen soll. Die Spalten sind von **null** bis **n** nummeriert. Die Reihenfolge der Erstellung ist egal, solange darauf geachtet wird, dass die Reihenfolge der übergebenen Positionsnummern richtig ist.

Dann muss die Methode *getColumnText()* in der Klasse **CMessageResultLabelProvider** angepasst werden. Hier muss die Nummer der zuvor erstellten Spalte zusammen mit dem neuen Attribut angegeben werden.

Zum Schluss müssen die Methoden, die eine Nachricht weiterleiten bis das Objekt erstellt wird, noch mit dem neuen Attribut angepasst werden. Dies könnte z.B. die Methode *addWarning()* im **MessageHandler** übernehmen.

6.5) Installation des Plugins

Um das Plugin zu installieren, muss die Eclipse Entwicklungsumgebung zunächst geschlossen werden, da das Plugin-Verzeichnis immer nur beim Start von Eclipse durchsucht wird.

Dann müssen das RuleCheck-Plugin und das BBCDT-Core-Plugin (benötigt für den C-Scanner) in das Plugin-Verzeichnis von Eclipse kopiert werden. Die Dateien:

- **org.bachelor.rulecheck_1.0.0.jar**
  Das eigentliche RuleCheck-Plugin
- **org.dworks.bbcdt.core_1.0.0.jar**
  Die Logik-Komponente des BBCDT-Plugins, die den CDT 3.x Scanner enthält, den das RuleCheck-Plugin zum Scannen von C-Quellcode benötigt.

müssen in das Verzeichnis **%Eclipse-Install-Root%/plugins** kopiert werden. Anschließend wird Eclipse wieder gestartet. Um herauszufinden, ob das Plugin wirklich installiert wurde, kann man die installierten Plugins einsehen (***Help → About Eclipse SDK → Plugin Details***). Hier sollte sich folgendes Bild zeigen, wenn das Plugin richtig installiert wurde:

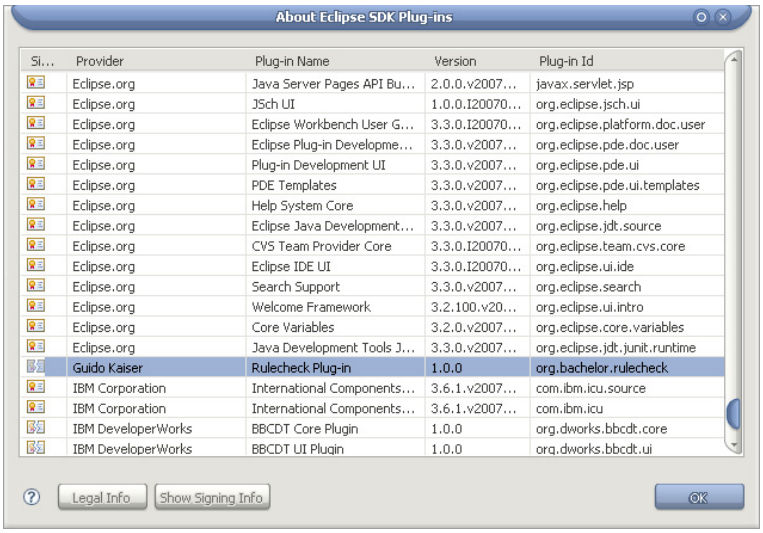


Abbildung 13 - Erfolgreich installiertes Plugin

Wenn der markierte Eintrag vorhanden ist, dann sollte das Plugin in der Eclipse Oberfläche zu sehen und zu benutzen sein. Es wird empfohlen das CDT Plugin in der Version 4.x zu installieren. Es ist nicht notwendig für die Funktionsweise des Plugins, jedoch für die Bearbeitung und Kompilierung von C-Projekten bzw. Dateien.

Damit die Überprüfungen der GCC mit einbezogen werden können, muss der Kompilierer MinGW[32] installiert werden. Das **bin**-Verzeichnis des MinGW muss im Ausführungspfad liegen oder alternativ muss der Aufruf in der GCC-Master-Rule angepasst werden, da der Kompilierer sonst nicht gestartet werden kann. Für eine kurze Bedienungsanleitung des RuleCheck Plugins siehe Kapitel **6.6**.

✓ Nicht-Funktionale Anforderung **N5** erfüllt.

6.6) Benutzung des Plugins

Das zuvor installierte Plugin sollte sich nun in der Eclipse Oberfläche zeigen. Man kann es durch die Symbole in der Werkzeugleiste oder über den Menüeintrag starten. Folgende Bilder zeigen das Plugin in der Eclipse-Oberfläche:

[32] http://www.mingw.org/

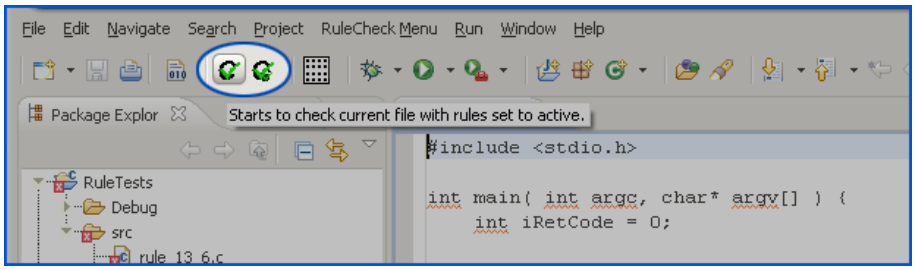


Abbildung 14 - Plugin-Buttons in der Werkzeugleiste

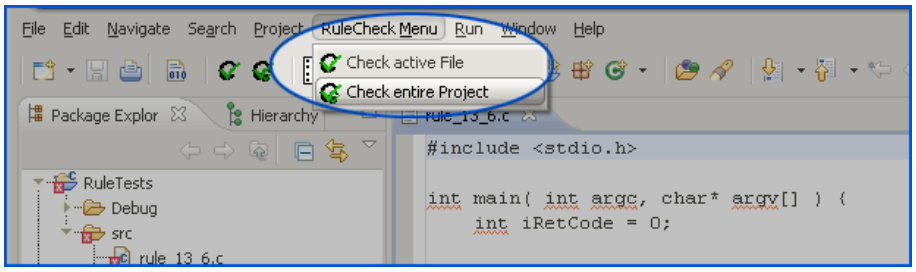


Abbildung 15 - Plugin-Menü

Das Plugin unterstützt zwei verschiedene Modi zum Überprüfen von C-Quellcode. Es kann die Datei, die im Eclipse-Editorfenster geöffnet und aktiv (sich im Vordergrund befindet) ist, mit der Funktion „**Check active File**“ überprüft werden.

Die zweite Variante ist das Überprüfen des gesamten Projekts. Hierfür wird wieder die aktuell geöffnete und aktive Datei im Editorfenster benötigt, um das Projekt zu ermitteln. Gestartet wird die Funktion über „**Check entire Project**“.

Wichtig für beide Funktionen ist, dass sich die zu prüfenden Dateien im aktuellen Workspace befinden, da für die Interaktion mit Eclipse Datei-Handler benutzt werden müssen, die nur mit Dateien im aktuellen Workspace funktionieren. Eine weitere Voraussetzung ist, dass die Dateien kompilierfähig sind, da die Überprüfungen sonst unter Umständen fehlerhafte Meldungen produzieren können.

Nachdem dann die Überprüfungen durchgelaufen sind, werden Warnungs- und Fehlermarker mit den jeweiligen Meldungen in den Quellcodes angezeigt. Sie werden im Standard Eclipse Fenster (unter „problems“) für Probleme gezeigt und verlinkt.

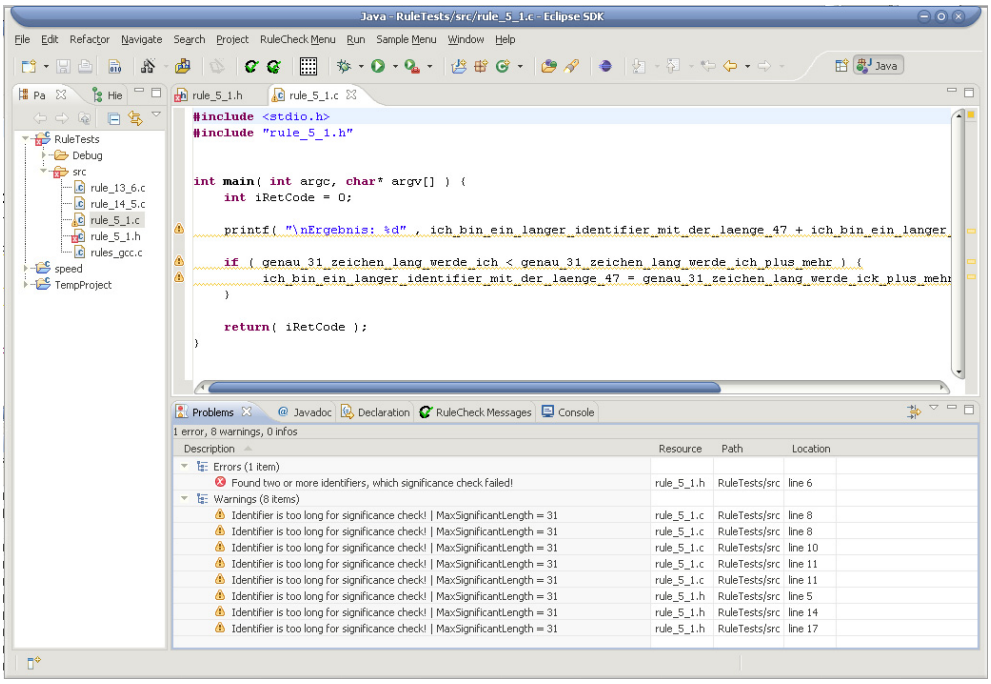


Abbildung 16 - Quellcode Marker

Zusätzlich finden sich etwas detailliertere Ausgaben im Plugin-eigenen Fenster „RuleCheck Messages“. Dort werden weitere Informationen, wie der Name der Regel, welche die Meldung ausgelöst hat und das Regelwerk, zu dem die Regel gehört, angegeben.

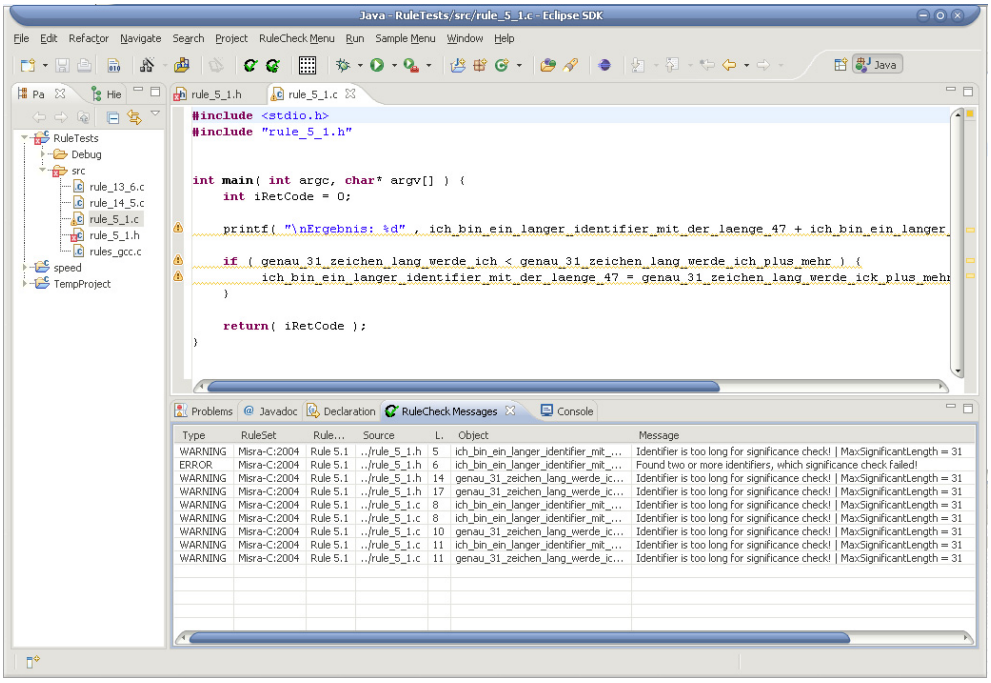


Abbildung 17 - Ausgaben des Plugins

Auf dem aktuellen Stand dieses Plugins können Regeln nur direkt im Quellcode ein- bzw. ausgeschaltet werden (für die Überprüfung). Da bisher nur wenige Regeln implementiert sind, wird diese Funktion noch nicht wirklich benötigt.

6.7) Schwierigkeiten / Erfahrungen

Eine Schwierigkeit war es, sich an das Plugin-Konzept bzw. die Wizard-Oberfläche zu gewöhnen. Die Einstellungen für das Plugin, die im nachhinein das Framework generieren, sind nicht sofort erklärend und nicht allzu übersichtlich angeordnet.

Wenn man das Prinzip der extension points verstanden hat, dann braucht man nur noch die richtigen bzw. passenden extension points zu suchen. Hierfür muss sehr tief in die Eclipse-Entwicklerdokumentation eingestiegen werden.

Ein wenig komplexes Plugin zu schreiben, wie z.B. ein „Hallo-Welt“ Plugin, stellte sich als einfach heraus. Beispiele hierfür waren in Eclipse und in der Literatur viele zu finden. Man muss dann allerdings etwas tiefer einsteigen und spezielle Kommunikationen mit Eclipse aufbauen. Denn hierfür gibt es keine weiterführende Literatur mehr. Das erfordert viele Möglichkeiten auszuprobieren und eine intensive Internetrecherche zu betreiben. Dabei ist zu hoffen, dass die Erklärungen und Beispiele, die gefunden werden, auch mit Eclipse 3.x anwendbar sind. Es gibt viele Dinge, die seit dem Versionswechsel nicht mehr funktionieren, wie z.B. diverse Zugriffe auf die Eclipse-Oberfläche.

Wirklich schwierig war es mit der Eclipse-Oberfläche zu kommunizieren, also die Dateinamen und Pfade der geöffneten Datei zu bekommen oder eine View anzeigen zu lassen, die man nach Belieben füllen kann.

Wenn man allerdings die Hürden mit den extension points und Eigenheiten der Plugin-Programmierung gemeistert hat, kann man relativ schnell gute und sinnvolle Plugins schreiben. Es wird sogar eine JUnit Plugin Testsuite angeboten, so dass man auch JUnit Tests auf ein Plugin anwenden kann.

7) Zusammenfassung

Die Arbeit zeigt, dass es schwer ist, wenn nicht sogar beinahe unmöglich, vernünftige Open Source Tools zu finden, um die Einhaltung der Misra-C:2004 Regeln zu prüfen. Man muss dafür schon selber etwas programmieren oder programmieren lassen.

Aufgrund der unzureichenden Dokumentation und den wenigen Beispielen im Internet zu spezifischen Problemen, erschwert die Entwicklung noch zusätzlich. Viele Funde beziehen sich auf veraltete Versionen von Eclipse, die gar nicht mehr funktionieren können, da mit Eclipse 3.0 ein großer Teil der Plugin Strukturen umgestellt wurde.

7.1) Endzustand des Plugins

Nach Abschluss der endgültigen Testphase des Plugins steht fest, dass alle Muss- und Sollkriterien umgesetzt wurden und funktionieren. Das Kannkriterium der Verlinkung von Verstoß-Meldungen und den betroffenen Code-Zeilen wurde implementiert und funktioniert. Die Unterstützung anderer Sprachen als C ist aufgrund der Architektur implizit gegeben. Es müssen nur minimale Anpassungen vorgenommen und ein entsprechender Scanner eingebunden werden.

Die Möglichkeit das Plugin über die Eclipse-Einstellungen zu konfigurieren ist nicht vollständig implementiert. Es existieren die nötigen Einträge in den Einstellungen, aber es steht keine Funktion dahinter. In Kapitel **7.3.1** wird kurz darauf eingegangen wie diese Funktionalität implementiert werden könnte.

Das Plugin ist weit genug ausgereift um es zu demonstrieren und um es in kleineren Projekten (aufgrund der Geschwindigkeit) zu benutzen. Sinnvoll wäre es noch einige oder alle weiteren Misra-C:2004 Regeln zu implementieren, damit das gesamte Regelwerk angewendet werden kann. Auch wenn einige Funktionen noch nicht implementiert sind, die das Verwalten der Ausgaben oder die Steuerung des Plugins vereinfacht, ist es trotzdem voll funktionsfähig.

- ✓ Nicht-Funktionale Anforderung **N1** erfüllt.
- ✓ Nicht-Funktionale Anforderung **N2** erfüllt.

7.2) Fazit

Der implementierte Prototyp zeigt, dass es möglich ist Eclipse mit eigenen Plugins auf einen deutlich höheren Standard für die Entwicklung von C-Programmen in Bezug auf Erweiterbarkeit, Wartbarkeit und Fehleranfälligkeit anzuheben.

Es können auf diese Weise alle Anforderungen erfüllt werden, die die Werkzeuge zur statischen Syntaxanalyse nicht erfüllen können oder sollen (siehe hierzu auch Kapitel **3**). Aufgrund der Tatsache, dass schon wenige Regeln in größeren Projekten eine Vielzahl an Warnungen erzeugen (vergleich siehe Kapitel **6.3.7**) macht es Sinn ein Werkzeug zu haben, mit dem man den Quellcode überprüfen lassen kann. Da die Auswahl an Open Source Software diesbezüglich sehr mager ist macht es durchaus Sinn das exemplarische Plugin zur Marktreife zu bringen.

Der Nutzen des Plugins wäre im Vergleich zum Aufwand der Implementierung von fehlenden Funktionalitäten, sehr hoch. Speziell auf die Anwendung der Misra-C:2004 Regeln für HAW-Hamburg Lehre-Projekte wäre es sinnvoll ein Werkzeug zu haben, mit dem die Studenten in der Lage sind ihren C-Quellcode überprüfen zu lassen. Dies gilt speziell für die Entwicklung von Software für eingebettete Systeme. Die Studenten sollten mit Eclipse-Plugin-Programmierung und mit Werkzeugen, die mittels statischer Syntaxanalyse Quellcode überprüfen, während des Studiums mindestens einmal in Berührung gekommen sein.

Guter Quellcode wird heutzutage immer wichtiger, weil immer mehr Software in Bereichen eingesetzt wird, bei denen ein Fehlverhalten menschliche oder finanzielle Schäden nach sich ziehen können. Aus diesem Grund sollten Studenten wenigstens ansatzweise mit dieser Problematik auseinandergesetzt werden.

Aufgrund der Tatsache, dass das hier entwickelte Plugin Open Source ist, eignet es sich gut für Studenten, da Einsicht in den Quellcode genommen werden kann und bei Bedarf neue Funktionalitäten hinzugefügt werden können.

7.3) Ausblick

Erweitern lässt sich der Prototyp durch Implementierung aller Misra-C:2004 Regeln. Es können auch andere sinnvolle Regeln implementiert werden, die nicht im Misra-Standard enthalten sind. Z.B. können Regeln für die objektorientierte Sprache C++ erstellt werden. Weitere Möglichkeiten finden sich in Kapitel **7.3.1**.

Welche Regeln später in einem Projekt wirklich überprüft werden, liegt im Normalfall im Ermessen der Projektleiter. Auf jeden Fall sollte darauf geachtet werden, dass sicherheitskritische Anwendungen durch möglichst viele Überprüfungen laufen.

Es können auch weitere Funktionalitäten und Optimierungen hinzugefügt werden, siehe hierzu Kapitel **7.3.2**.

7.3.1) Mögliche weitere Funktionalitäten

Die Quick-Fix Funktion von Eclipse, bei der Verbesserungen im Code mithilfe von Mustern vorgenommen werden können, wäre für einige Regeln eine hilfreiche Funktionalität, die dem Programmierer einiges an Arbeit abnehmen kann. Beispielsweise könnte bei einer vergessenen / weggelassenen Deklaration einer Funktion diese automatisch in die zugehörige Header-Datei geschrieben werden. Ein allgemeines Interface für die Muster würde mehrere Wochen in Anspruch nehmen. Für jede Regel dann die nötigen Muster zu erstellen, dürfte nur noch wenig Zeit in Anspruch nehmen. Die benötigte Zeit ist abhängig von der Komplexität der Regel.

Die Eclipse-Einstellungen beinhalten schon ein leeres Feld für das Plugin. Hier könnte man dynamisch die Regeln ein- und ausschalten und konfigurieren. Zum Beispiel könnte für Regel **5.1** (Länge der Signifikanz von Bezeichnern) die Länge konfiguriert werden. Da die nötigen Schnittstellen bereits existieren, sollte die Implementierung in ein bis zwei Tagen erfolgen können.

Eine Reporting-Funktion, die Metriken, Grafiken und HTML-Listen erstellen kann, wäre eine gute und durchaus nützliche Erweiterungsmöglichkeit. Beispielsweise könnten in einem Report alle ausgegebenen Meldungen mit Position und Verbesserungsvorschlag aufgelistet werden. Um die Report-Funktion zu erstellen, sollten wenige Tage Implementierungszeit genügen. Die Metriken zu erstellen, würde je nach Auswahl ungefähr eine Woche dauern.

Das Erstellen von Regeln kann man mithilfe von extension points aus dem Plugin herausholen, so dass nicht das Plugin verändert werden muss, wenn neue Regeln implementiert werden. Um die nötigen extension points zu erstellen, benötigt man ein bis zwei Tage.

Es könnten Regelwerk-Mengen gebaut werden, die jeweils einen RuleManager beinhalten. Ein globaler Manager würde steuern, welche Regelwerk-Mengen benutzt werden. Theoretisch würde nur eine Manager-Ebene darüber gesetzt werden, damit man zwischen den verschiedenen Regelwerken für verschiedene Projekte wechseln kann. Diese Implementierung sollte innerhalb von ein bis zwei Tagen vorgenommen werden können.

Eine Priorisierung von Regeln könnte dabei helfen die höher gewichteten Verstöße aus einem Projekt (zum Beispiel Aspekt der Sicherheit wichtiger als Aspekt der Wartbarkeit) herauszufiltern um diese zuerst zu bearbeiten. Dafür kann im Regelobjekt ein Prioritätswert implementiert werden, der später beim Report oder bei der Sortierung der Ausgabe in Eclipse für die Positionierung zuständig ist. Implementierungszeit wäre ca. ein Tag.

Zum Schluss ist noch anzumerken, dass die Implementierung aller Regeln des Misra-C:2004 Standards einige Monate dauern würde, da schon bei wenig komplexen Regeln ein erheblicher Parse-Aufwand entsteht. Zum Beispiel gibt es Regeln, für die das gesamte Projekt durchsucht werden muss (**extern**-Deklarationen und zugehörige Definitionen).

7.3.2) Mögliche Optimierungen

Eine mögliche Optimierung der Parsing-Vorgänge wäre, wenn die Überprüfungen parallel mithilfe von Threads stattfinden können. Laut dem Ergebnis des Geschwindigkeitstest von Kapitel **6.3.7** würde die Nutzung aller Kerne / CPUs schon einen Geschwindigkeitsvorteil bringen. Alternativ kann hier auch ein mächtigerer Parser entwickelt werden, der mit Satzanfängen arbeitet und die Möglichkeit hat zurückzuspringen, um die nächste Regel auf denselben Satz anzuwenden. Diese Optimierung sollte bei einer größeren Anzahl von Regeln eine deutlich höhere Geschwindigkeit während der Überprüfung bringen. Die Implementierung sollte für beide Varianten nur wenige Tage in Anspruch nehmen.

Literaturverzeichnis

1. Clayberg, Eric und Rubel, Dan. Eclipse Building Commercial-Quality Plug-ins. Amsterdam : Addison-Wesley Longman, 2006.

2. Carlson, David. Eclipse Distilled. Amsterdam : Addison-Wesley Longman, 2005.

3. Budinsky, Frank, et al. Eclipse Modeling Framework: A Developer's Guide. Amsterdan : Addison Wesley Longman, 2003.

4. Beck, Kent und Gamma, Erich. Contributing to Eclipse: Principles, Patterns, and Plug-Ins. Amsterdam : Addison-Wesley Longman, 2003.

5. Galllardo, David. Eclipse In Action. s.l. : Manning, 2003.

6. Moore, Bill, et al. Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework. s.l. : IBM, 2004.

7. Daum, Berthold. Professional Eclipse 3 for Java Developers. Heidelberg : dpunkt.verlag GmbH, 2004.

8. (MISRA), The Motor Industry Software Reliability Association. Misra-C:2004 - Guidelines for the use of the C language in critical systems. Warwickshire : MIRA Limited, 2004.

9. Merkle, Bernhard. Plan C. iX. Februar 2006, S. 52-58.

10. Stallman, Richard M. Using the GNU Compiler Collection. Boston : GNU Press, 2003.

11. Eclipse Foundation. Eclipse Javadoc-API. [Online] http://help.eclipse.org/.

12. Kernighan, Brian W. und Ritchie, Dennis. The C Programming Language. s.l. : Prentice Hall, 1978.

13. Stroustrup, Bjarne. Die C++-Programmiersprache. Amsterdam : Addison-Wesley Longman, 1984.

14. Ecma International. C# Language Specification. Geneva : Ecma International, 2006.

15. Schwichtenberg, Holger. Microsoft .NET 3.0 Crashkurs. Unterschleißheim : Microsoft Press, 2007.

16. digitalmars. Spezifikation der Programmiersprache D. [Online] digitalmars. http://www.digitalmars.com/d/.

17. Würtenberg, Jens. Elektronik automotive. [Online] Februar 2006. http://www.elektroniknet.de/home/embeddedsystems/fachwissen/uebersicht/software/entwicklungssoftware/der-programmierstandard-misra/.

18. Hoff, Todd. C++ Coding Standard. [Online] 17. Juli 2007. http://www.possibility.com/Cpp/CppCodingStandard.html.

19. Coding Conventions for C++ and Java. [Online] 22. Januar 1998. http://www.macadamian.com/index.php?option=com_content&task=view&id=34&Itemid=37.

20. QA Systems GmbH. QA-C/MISRA - Holen Sie sich den MISRA-Standard in Ihre C-Programme. [Online] QA Systems GmbH. http://www.qa-systems.de/html/deutsch/produkte/qacm/qacm.php.

21. Gimpel Software. MISRA C (1998) checking provided by PC-lint/FlexeLint. [Online] Gimpel Software. http://www.gimpel.com/html/misra.htm.

22. Bäumer, Dirk; Megert, Daniel; Weinand, André. Eclipse Plugins – Entwickeln und Publizieren. [Online] Sigs-Datacom. http://www.sigs.de/publications/os/2004/01/weinand_OS_01_04.pdf.

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel und im Sinne der Prüfungsordnung nach §25(4) erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Hamburg, den **01.10.2007**

(Guido Kaiser)