technische universität
dortmund

Diplomarbeit

Aspektorientierte Implementierung und Bewertung von Fehlertoleranzmechanismen im eingebetteten Betriebssystem eCos

Daniel Wulfert

25. März 2013

Betreuer:
Prof. Dr.-Ing. Olaf Spinczyk
Dipl.-Inf. Christoph Borchert

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12
Arbeitsgruppe Eingebettete Systemsoftware
http://ess.cs.tu-dortmund.de

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 25. März 2013

Daniel Wulfert

Zusammenfassung

Durch immer kürzere Entwicklungszeiten und steigende Komplexität der Systeme, aber auch durch Ausreizung physikalischer Grenzen, werden sowohl Hard- als auch Softwarefehler häufiger auftreten, sodass zukünftige Systeme solche Fehler erwarten und mit ihnen umgehen müssen. Gerade im Bereich der Betriebssysteme[1] und im speziellen der Treiber[2] können Fehler unvorhersagbare, meist katastrophale Folgen bis hin zum Absturz des gesamten Systems haben.

Im Rahmen der Diplomarbeit soll zu verschiedenen Szenarien ausgewählte Mechanismen und Konzepte zur Fehlertoleranz gesichtet und mit Hilfe von Aspektorientierter Programmierung[3] exemplarisch im eingebetteten Betriebssystem *eCos*[4] implementiert werden. Die Umsetzung mit Hilfe von Aspektorientierter Programmierung ermöglicht eine für den jeweiligen Anwendungsfall hochgradig konfigurierbare Lösung zu schaffen, zeigt aber auch Stellen auf, an denen die Aspektorientierte Programmierung an ihre Grenzen stößt und eine Erweiterung der Programmiersprache nötig ist. Zusätzlich soll die Fehlertoleranzimplementierung sowohl auf ihre Effektivität als auch auf ihre Kosten in Bezug auf Performanz und Speicherbedarf evaluiert werden.

Inhaltsverzeichnis

1 Einleitung

> „Reliability remains a paramount problem for operating systems. As computers are further embedded within our lives, we demand higher reliability because there are fewer opportunities to compensate for their failure.“ [5]

Computer sind aus dem heutigen Alltag kaum mehr wegzudenken. Dabei sind es vor allem eingebettete Systeme, die uns jeden Tag umgeben und uns das Leben vereinfachen, ohne dass wir ihnen große Beachtung schenken und sie zum Teil gar nicht bemerken. Angefangen bei Fortbewegungsmitteln wie Kraftfahrzeugen oder Flugzeugen, die einen ganzen Verbund an eingebetteten Systemen enthalten, über die immer leistungsfähigeren Mobiltelefone, zu Haushalts- und Unterhaltungselektronik wie Fernsehern oder Waschmaschinen bis zu Medizin- und Haustechnik wie Fahrstühlen.

Diese Abhängigkeit hat aber auch ihre Schattenseiten, wenn durch immer kürzere Entwicklungszeiten und steigende Komplexität der Systeme, aber auch durch Ausreizung physikalischer Grenzen, das Vorkommen von Hard- als auch Softwarefehlern steigt, das zum Ausfall solcher Systeme führen kann. Zukünftige Systeme müssen deshalb mit solchen Fehlern umgehen können und diese tolerieren.

1.1 Zielsetzung

Ziel dieser Diplomarbeit ist es, verschiedene softwarebasierte Fehlertoleranzmechanismen zu implementieren und ihre Performanz anschließend zu bewerten. Im Laufe der Arbeit wurden diese Ziele, einerseits durch Voruntersuchungen, aber auch durch Prüfung der Eignung verschiedener Verfahren und Implementierungsmöglichkeiten im eingebetteten Betriebssystem *eCos*, konkretisiert.

Bei der Wahl der Fehlertoleranzmechanismen stand vor allem die Plausibilitätsprüfung („sanity check“) im Vordergrund, wobei eine speziell auf *eCos* zugeschnittene Variante, als auch ein generisches Verfahren mittels Aspektorientierter Programmierung getestet wurde. Die Ziele bei den einzelnen Konzepten waren,

- durch die **Absicherung der Treiberschnittstelle** von *eCos* zu untersuchen, inwieweit Treiberfehler in einem einfachen eingebetteten Betriebssystem mittels Fehlertoleranzmechanismen verhindert werden können, ohne den Speicherplatzverbrauch oder die Laufzeit wesentlich zu beeinflussen.

- Bei der **generischen Zeigervalidierung** ging es vor allem darum ein allgemeineres Fehlertoleranzverfahren zu entwickeln, sich auf verschiedenen Zielplattformen unabhängig einsetzen lässt und eine hohe Konfigurierbarkeit aufweist.

- Durch die **Evaluation** der beiden Verfahren kann festgestellt werden, bis zu welchem Grad diese Ziele erreicht wurden und wie performant die individuellen Lösungen sind.

1.2 Strukturierung der Arbeit

Der Einleitung folgend, wird eine kurze Einführung zu den im Titel dieser Arbeit angesprochenen Themengebieten gegeben. In Abschnitt 2 erfolgt eine Einführung zu den Grundlagen der Aspektorientierung Programmierung, wobei im Besonderen auf die aspektorientierte Programmiersprache *AspectC++* eingegangen wird, sowie deren Kombination mit der Templatemetaprogrammierung. Im folgenden Abschnitt 3 wird das Thema der Fehlertoleranz sowie Mechanismen zur Fehlererkennung und Fehlerbehebung, als auch Methoden zur Bewertung von Fehlertoleranz, in Hinblick auf Fehlerinjektionssysteme, näher betrachtet. Abschnitt 4 beleuchtet den Themenkomplex der eingebetteten Betriebssysteme, mit Fokus auf dem hier verwendetem Betriebssystem *eCos* und geht auch auf das Thema der Treiber näher ein.

Nach den Einführungsthemen folgen die Beschreibungen zur Implementierung der Fehlertoleranzmechanismen. Dabei wird in Abschnitt 5 die Absicherung des E/A-Subsystems erläutert, gefolgt von Vorüberlegungen und Exemplifikationen zur Methode der generischen Zeigervalidierung in Abschnitt 6.

Die Bewertung der implementierten Fehlertoleranzverfahren als auch der Versuch einer mathematischen Abschätzung der Bereichsprüfung findet sich in Abschnitt 7. Abschließend erfolgt in Abschnitt 8 eine kurze Zusammenfassung und Diskussion der gemachten Erfahrungen und gesammelten Ergebnisse sowie ein Ausblick auf mögliche folgende Arbeiten in diesem Gebiet.

2 Aspektorientierte Programmierung

Aspektorientierte Programmierung (kurz *AOP*) [3] bezeichnet ein Programmierparadigma welches hilft querschneidende Belange („crosscutting concerns“) modular zu implementieren.

Die folgenden Abschnitte sollen eine kurze Einführung in die Aspektorientierte Programmierung geben, sowie der für diese Arbeit genutzten aspektorientierten Spracherweiterung *AspectC++* für die Programmiersprache *C/C++*.

Da nicht nur das Paradigma der *AOP*, sondern auch spezifische Spracheigenschaften von *AspectC++* zur Einbringung von Fehlertoleranzmechanismen in dieser Arbeit von Bedeutung sind, wird auf einige dieser Spracherweiterungen wie die *JoinPoint API* näher eingegangen, sowie ein kurzer Einblick in die Template-Metaprogrammierung, die ebenfalls im späteren Verlauf bei der generischen Bereichsprüfung relevant ist, geben.

Im Anschluss soll ein Überblick über vorangegangende Arbeiten im Bereich der Aspektorientierten Programmierung, die in Beziehung zu Fehlertoleranzmechanismen und eingebetteten Systemen stehen, gegeben werden, sowie eine Überleitung zu den im nächsten Kapitel behandelten eingebetteten Betriebssystemen erfolgen.

2.1 Grundlagen der Aspektorientierten Programmierung

Bei der Funktionalen Dekomposition im Bereich der Softwareentwicklung, d.h. beim Zerlegen des Gesamtsystems in grundlegende Systemfunktionen bzw. Einzelprobleme mit bestimmten Eigenschaften und Verhalten (z. B. Objekte mit Attributen und Methoden in der Objektorientierten Programmierung oder Funktionen in der Prozeduralen Programmierung), spielen viele Faktoren wie Granularität und Abhängigkeiten, aber auch Designentscheidungen wie Adaptivität und Wiederverwendbarkeit eine Rolle. Dabei kann es oft zu Konflikten kommen, sodass nicht alle Belange als eigenständige Komponenten modelliert werden können.

Beispiele für solche querschneidende Belange sind neben der Fehlertoleranz, Funktionen wie Logging oder Authentifizierung, die zwar eigenständige Belange darstellen, sich wiederum aber auf mehrere Komponenten auswirken.

Um dennoch eine Trennung der Belange („seperation of concerns“ [6]) und so die Isolation, Komposition und Wiederverwendbarkeit von Programm- bzw. Systemfunktionen zu gewährleisten, kann das Konzept der Aspekte helfen. Im Zuge der Aspektorientierten Programmierung unterscheidet man also zwischen

- **Komponenten**, die sich als eigenständige Einheit bzw. Subeinheit darstellen lassen durch z. B. Methoden, Objekte oder Funktionen und

- **Aspekten**, die querschneidende Belange des Problemraums, also Eigenschaften oder Anforderungen der Funktionalen Dekomposition des Systems im Lösungsraum darstellen, die sich keiner eigenständigen Einheit zuordnen lassen.

Die Stellen im Programmfluss, an denen Aspekte verwendet werden sollen, werden durch sog. Joinpoints identifiziert. Dabei können sich Aspekte auf mehrere Komponenten auswirken, sowie verschiedene Teile (z. B. Methoden eines Objektes) einer einzelnen Komponente beeinflussen. [3]

Versucht man Aspekte bspw. mit Hilfe Objektorientierter- oder Prozeduraler Programmierung zu implementieren, resultiert dies meist in verstreutem, aber eigentlich modular beschreibbarem Code, der sich innerhalb verschiedener Komponenten wieder findet oder zu großen aus eigentlich verschiedenen Komponenten bestehenden Einheiten mit sich wiederholenden Quelltextabschnitten, als Konsequenz querschneidender Belange. Dies führt dazu, dass sowohl Lesbarkeit als auch Nachvollziehbarkeit des Quelltextes leiden und Codeabschnitte durch nicht identifizierbare Komponentenzugehörigkeit schwer zu warten, zu erweitern oder wiederzuverwenden sind.

Vor allem Betriebssysteme mit ihren vielen querschneidenden Belangen können von *AOP* profitieren. Ein weiterer Vorteil der *AOP* liegt darin, dass Aspekte sehr einfach an- bzw. abschaltbar sind, welches die Konfigurierbarkeit und Variabilität von Software auch in Hinblick auf bspw. Software-Produktlininen [7][8] erhöht. Ein Beispiel, das sowohl eine Vielzahl querschneidender Belange wie auch feingranulare Konfigurierbarkeit vereint, ist das im nächsten Kapitel näher betrachtete eingebettete Betriebssystem *eCos* [4].

Aspektorientierte Programmierung stellt zudem eine Alternative zu Lösungsansätzen dar, die auf Ereignisgesteuerter Programmierung basieren, bei der im Gegensatz zur *AOP* gerade spätere Änderung bzw. nachträgliche Integration in vorhandene Komponenten kaum möglich ist.

2.1.1 Weben von Aspekten

Um Aspekt- und Komponentencode im Zuge der Komposition wieder zu vereinen wird ein sog. Aspektweber („aspect-weaver“), wie in Abbildung 2.1 zu sehen verwendet, wobei die Erzeugung von Quellcode durch Integration von Aspektcode in den Komponentencode an den Verbindungspunkten (Joinpoints) sowohl zur Übersetzungszeit als auch zur Laufzeit möglich ist. Bei dem Aspektweber handelt es sich jedoch nicht um einen nativen Compiler, sondern lediglich einen Source-to-Source Übersetzer der eine Joinpointrepräsentation des Komponentenprogramms erstellt. Vorteil ist neben der vereinfachten Implementierung des Webers, dadurch dass kompliziertere Compiler-Komponenten wie die Optimierung weggelassen werden können, eine einfachere Projektintegration durch eigene Übersetzerauswahl. Nachteil dieses Ansatzes ist die Abhängigkeit von einer festgelegten Programmiersprache (vgl. [3]).

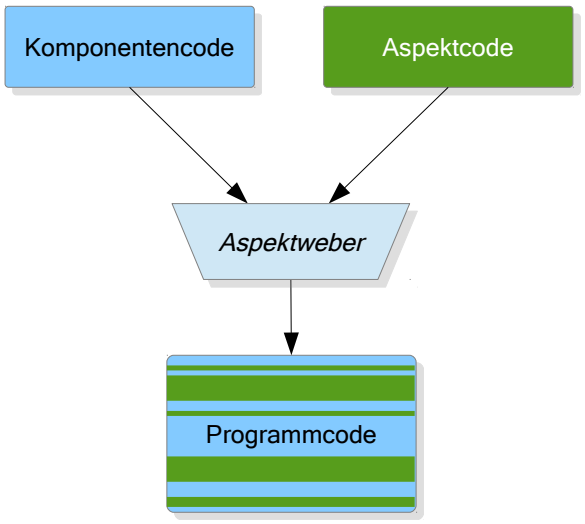


Abbildung 2.1: Veranschaulichung der Funktionalität des Aspektwebers, der Aspektcode an die durch Joinpoints ausgezeichneten Stellen im Komponentencode einfügt um so den Programmcode generiert.

2.2 AspectC++

AspectC++ (vgl. [9],[10]) ist eine aspektorientierte Erweiterung der Programmiersprache *C++*. Neben *AspectJ* [11], dass *AspectC++* vor allem im Bereich der Syntax und Semantik beeinflusst hat, stellt es eine der ersten Spracherweiterungen für *AOP* dar. *AspectC++* ist vor allem in den Bereichen der eingebetteten Systemsoftware und hardwarenahen Programmieren interessant, in denen wegen vorherrschender Ressourcenbeschränkungen, die traditionell dominierende Programmiersprache *C/C++* ist. Deshalb wurde *AspectC++* dahin gehend konzipiert unnötigen Laufzeit- oder Speicheroverhead zu vermeiden, weswegen bspw. das Weben der Aspekte zur Übersetzungszeit stattfindet, sowie die Vorteile und Stärken von *C/C++* zu erhalten.

2.2.1 Spracheigenschaften

Zu den Spracheigenschaften von *AspectC++* (vgl. [12]) zählen neben der Aspektdeklaration um querschneidende Belange zu implementieren, Pointcuts die eine Menge von Joinpoints definieren, Advices welche z. B. dafür genutzt werden, um den Code für eine Joinpointmenge festzulegen, sowie die *Joinpoint API* die diverse Laufzeit- und Compilezeitinformationen zur Verfügung stellt.

Im Folgenden sollen diese Konzepte näher betrachtet werden, mit Fokus auf die relevanten Sprachmerkmale, die für Implementierung der Fehlertoleranzmechanismen im Laufe dieser Arbeit benötigt werden.

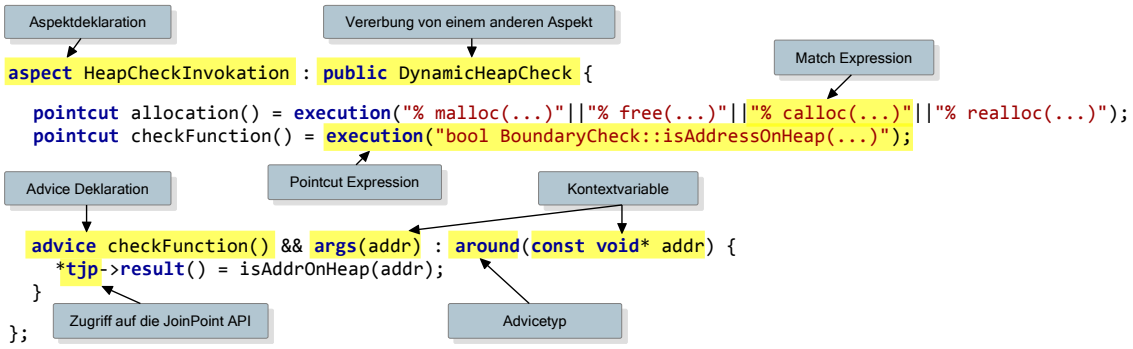


Abbildung 2.2: Beispielaspekt zur dynamischen Heapprüfung, welches eine Funktion `isAddressOnHeap(const void* addr)` durch die im Advice beschriebene Funktionalität ersetzt. Die erste Poincut-Deklaration überschreibt dabei den virtuellen Pointcut, des geerbten Aspekts `DynamicHeapCheck`.

2.2.1.1 Aspekte

Aspekte in *AspectC++* stellen eine Erweiterung des Klassenkonzeptes von *C++* dar, mit deren Hilfe querschneidende Belange implementiert werden können. Wie Klassen können sie einen eigenen Konstruktor, sowie Methoden und Attribute beinhalten und werden wie in Abbildung 2.2 zu sehen mit dem Schlüsselwort **`aspect`** deklariert.

Aspekte können zudem von anderen Aspekten oder Klassen erben, was die Wiederverwendbarkeit und Konfigurierbarkeit von Aspekten erhöht.

2.2.1.2 Pointcuts

Pointcuts definieren eine Menge von Joinpoints mit Hilfe von Pointcut-Ausdrücken („pointcut expression“). Diese wiederum können aus Match Expressions und Pointcut-Funktionen gebildet werden.

Match Expressions in *AspectC++* sind Zeichenketten die einen Typ, Klasse, Namespace oder eine Funktion beschreiben und so eine Art Suchmuster darstellen, wobei „%“ als Wildcard bzw. „...“ für beliebige Funktionsargumentlisten eingesetzt werden kann.

Pointcut-Funktionen können je nach Bezug unterschiedliche Ausprägungen besitzen:

- Um die zu filternden Joinpoints auf einen **Bereich** zu beschränken, kann die Poincut-Funktion **`within`** benutzt werden, um z. B. alle Funktionen innerhalb einer Klasse oder eines Namensraums zu erreichen.

- In Bezug zu **Funktionen** existieren **`call`**, **`execution`**, **`construction`** und **`destruction`**, die Joinpoints im Programmfluss zu Funktionsaufrufen bzw. -ausführung oder des Konstruktors bzw. Destruktors einer Klasse beschreiben.

- Um Joinpoints nach ihrem **Kontext** zu ermitteln, stehen die Funktionen **`that`**, **`target`**, **`result`**, **`args`** zur Verfügung, die bei einem Funktionsaufruf oder dessen Aus-

führung einen Abgleich mit dem angegebenen Typ oder Variablennamen durchführen, wobei `that` und `target` eventuell erst zur Laufzeit bestimmbar sind.

Eine exemplarische Verwendung ist in Abbildung 2.2 zu sehen, in der auf einen bestimmten Variablennamen des Funktionsarguments gefiltert wird, wobei die angegebene Kontextvariable auch in der Argumentliste des Advice-Typs deklariert werden muss.

- Für eine **kontrollflussabhängige** Ausführung kann die Pointcut-Funktion `cflow` verwendet werden um bspw. nur Aufrufe zuzulassen die aus einem entsprechenden Kontext z. B. vom Kernel erfolgen.
- Die den **Typus** beschreibenden Funktionen `base` und `derived` geben jeweils alle Basisklassen bzw. zusätzlich zur angegebenen Klasse alle Klassen an die von dieser erben und sind deshalb auch nur für Klassennamen vorgesehen.

Um die mögliche beschreibbare Menge an Joinpoints zu erhöhen, können die angegebenen Pointcut-Funktionen außerdem mit algebraischen Ausdrücken kombiniert werden.

In *AspectC++* können Pointcuts aber auch virtuell und sogar rein virtuell sein, sofern sie mit dem entsprechenden Schlüsselwort deklariert sind, womit der Aspekt zu einem abstrakten Aspekt wird, der durch Vererbung und Redefinition des entsprechenden Poitcuts auf die gewünschten Joinpoints angewendet werden kann.

2.2.1.3 Advices

Advices dienen dazu den Code, der bei Erreichen eines Joinpoints ausgeführt bzw. eingefügt werden soll zu spezifizieren. Sie können nur innerhalb eines Aspekts definiert werden und man unterscheidet zwischen

- `before`-Advices bei denen der Code vor einem entsprechenden Joinpoint ausgeführt wird,
- `after`-Advices bei denen der Quelltext des Advice nach den im Pointcut angegebenen Joinpoints ausgeführt wird und
- `around`-Advices bei der der Advice-Code anstelle des durch die Joinpoints beschriebenen Komponentencodeabschnitte ausgeführt wird. Mit Hilfe eines Aufrufs der Joinpoint API Funktion `tjp->proceed()` innerhalb des Advice, lässt sich der Originalcode falls gewünscht trotzdem manuell ausführen.

  Diese Art von Advice eignet sich, wie beim Heapprüfungsbeispiel in Abbildung 2.2, hervorragend dafür um durch Ersetzen einer Defaultfunktion mit einem Standardverhalten durch den Aspektcode, das Systemverhalten anzupassen oder zu konfigurieren.

Neben diesen sog. Code-Advices existieren noch zwei spezielle Advice-Typen. Zum einen existiert ein Einführungs-Advice („introduction advice“), der neue Methoden oder Attribute mittels des Schlüsselworts `slice` zu einem Joinpoint hinzufügen kann, um etwa eine vorhandene Klasse um eine Zählervariable zu erweitern.

Zum anderen gibt es den `order`-Advice, der mittels des Aspektwebers versucht eine gegebene Menge von Aspekten in eine totale Ordnung zu bringen oder bei Konflikten einen entsprechenden Fehler wirft.

2.2.2 Joinpoint API

Innerhalb eines Advices können Informationen über den aktuellen Joinpoint durch die *Joinpoint API* abgefragt werden. Im Gegensatz zu *Java*, dass eher Generizität zur Laufzeit unterstützt, wird Generizität in *C++* vor allem zur Compilezeit verwendet, vor allem mit Unterstützung von Templates oder des *C-Präprozessors*, weswegen die Idee einer Laufzeit Joinpoint API aus *AspectJ*, in *AspectC++* um eine Joinpoint API erweitert wurde, die auch zur Kompilationszeit Informationen bereitstellt [10].

Die dadurch schon zur Kompilationszeit möglichen Berechnungen unterstützen die Eignung von *AspectC++* zur Implementierung eingebetteter Systemsoftware, bei der meist die Übersetzungszeit im Gegensatz zur Laufzeit eine eher untergeordnete Rolle spielt.

Statische Typinformationen und Konstanten können mit Hilfe der Pseudoklasse `JoinPoint` abgefragt werden. Hierzu zählen unter anderem

- `JoinPoint::That`, dass den Typ des aufrufenden Objektes ausgibt,
- `JoinPoint::Result`, mit dem der Ergebnistyp und
- `JoinPoint::ARGS`, mit dem die Anzahl der Argumente der aktuellen Funktion innerhalb eines Advices ermittelt werden kann.

Daneben existieren weitere statische Joinpoint-Informationen wie einer eindeutigen Identifikationsnummer für den jeweiligen Joinpoint und Typinformationen der jeweiligen Argumente.

Zur Laufzeit erfolgt der Zugriff auf die *JoinPoint API* durch den speziellen Pointer `tjp` („this join point“), welcher auf eine Instanz von `JoinPoint` zeigt. zu den statischen Informationen zugehörigen Funktionen oder die oben bereits erwähnte Funktion `tjp->proceed()` die innerhalb eines `around`-Advices den ursprünglichen Code ausführen kann.

Eine vollständige Liste der durch die *Joinpoint API* bereitgestellten Informationen findet man in [12].

2.2.3 Architektur

Um aus *AspectC++*-Quellcode validen Maschinencode zu erzeugen, wird zunächst der Source-to-Source Compiler *ac++* verwendet. Dieses auf *PUMA* (s. a. [13]) basierende Compiler-Frontend generiert hierbei *C++*-Code, welcher danach von allen hierfür gängigen Compilern verwendet werden kann (vgl. [14]).

Die Definition von Aspekten geschieht dabei in speziellen Header-Files (*.ah) um Aspekt- und „normalen“ *C++*-Code voneinander zu trennen und für den Compiler erkennbar zu machen.

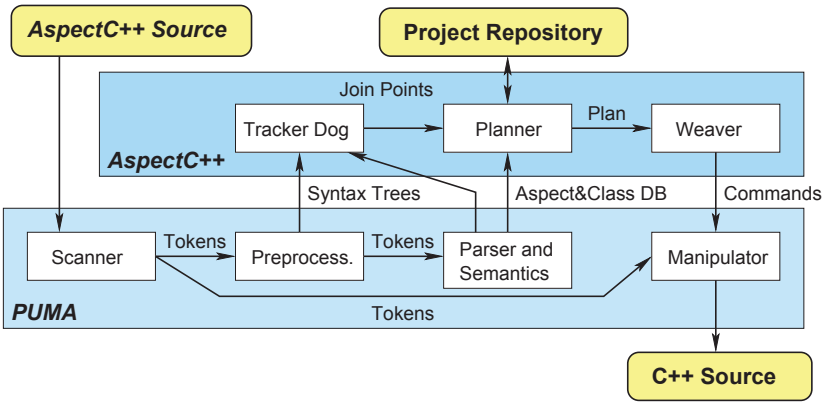


Abbildung 2.3: Architektur des *AspectC++* Webers aus [14].

In Abbildung 2.3 ist die Funktionsweise des *AspectC++* Webers, mit seinen Komponenten und dem zugehörigen Datenfluss, schematisch dargestellt. Während der Übersetzung wird in der Analysephase ein Syntaxbaum generiert und an den sog. *Tracker Dog* weitergereicht. Der *Tracker Dog* sucht alle möglichen Joinpoints und reicht diese schließlich an den *Planner* weiter, der die im Programm vorhandenen Pointcut Ausdrücke auswertet und letztendlich alle Joinpoints errechnet. Der Weber nutzt nun den Manipulator des *PUMA* Frameworks, um nach Vorgaben des *Planners* den fertigen Quelltext zu generieren.

2.3 AspectC++ und Templatemetaprogrammierung

Auch wenn Metaprogrammierung in *C/C++* bspw. auch durch C-Präprozessor-Makros erreicht werden kann, sind Templates ein mächtiges Mittel zur generativen sowie zur generischen Programmierung und im Gegensatz zum Präprozessor, der eher unabhängig von der Programmiersprache ist und wie ein reines Textersetzungswerkzeug agiert, Teil der Sprache ist.

2.3.1 Templatemetaprogrammierung

Andere Sprachen wie Java unterstützen zwar generische Typen um wiederverwendbare Datenstrukturen zu implementieren, stellen aber keine Sprachmittel zur Metaprogrammierung zur Verfügung. In *C++* werden Templates sowohl für generische Typen, wie auch für Metaprogramme eingesetzt. Templatemetaprogrammierung [15] kann dabei prinzipiell als vollständig eigene Sprache verwendet werden, da sie Turing-vollständig [16] ist.

Ein Vorteil bei der Programmierung mit Hilfe von von Templates in *C++* ist, ähnlich wie beim C-Präprozessor, dass Ausdrücke während des Kompilierens ausgewertet werden. Dadurch können Einsparungen in der Laufzeit erreicht werden, was vor allem, aber nicht ausschließlich, für eingebettete Systemsoftware interessant ist.

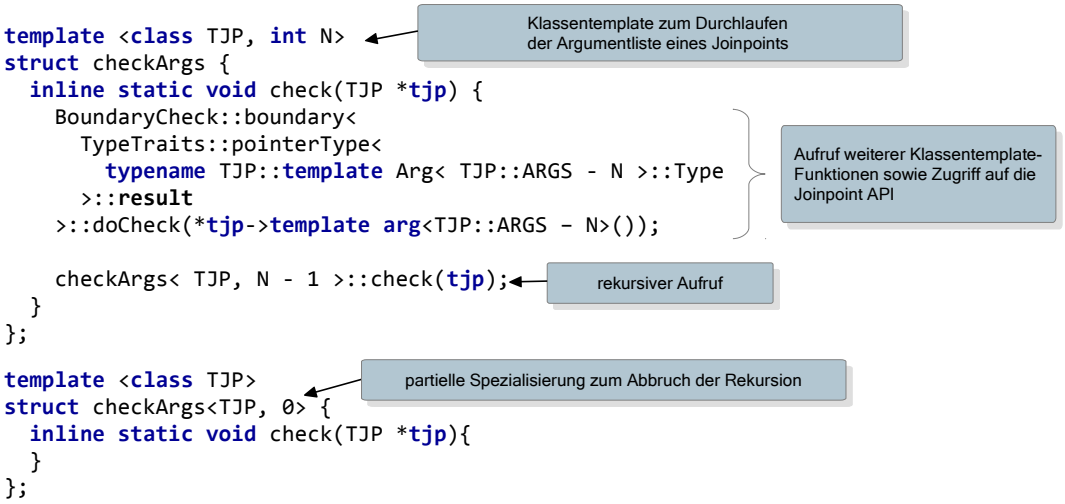


Abbildung 2.4: Typprüfung und rekursiver Durchlauf der Argumentliste eines Joinpoints mit Hilfe von Templatemetaprogrammierung.

In *C++* sind zwei Templatetypen von Bedeutung:

- Mit Hilfe von **Funktionstemplates** lassen sich generische Funktionen implementieren, die für eine Vielzahl von Argumenttypen verwendbar sind oder etwa je nach Kontext unterschiedliche Rückgabetypen besitzen.

  Beispiel für solche Funktionen sind min- oder max-Funktionen, die entweder variable, uniforme Typen miteinander vergleichen können, oder sich sogar zum Vergleich verschiedener Typen eignen.

- **Klassentemplates** dagegen werden etwa für generische Datenstrukturen wie etwa Typlisten (siehe [15]) eingesetzt, wobei neben Klassen auch Structs und Unions für Klassentemplates verwendet werden können. Klassentemplates lassen sich je nach Zweck spezialisieren, wobei diese Spezialisierungen bzw. partielle Spezialisierungen etwa zur gesonderten Behandlung bestimmter Ausprägungen genutzt werden können. Mit Hilfe von Spezialisierung und Selbstinstanziierung lassen sie sich komplette, auf Templates basierende Programme realisieren, die dann zur Übersetzungszeit ausgewertet werden.

  Ein Beispiel für Klassentemplates ist in Abbildung 2.4 zu finden. Hier wurden partielle Spezialisierung und Selbsinstanziierung dazu eingesetzt, um rekursiv die Argumentliste eines Joinpoints zu durchlaufen. Des Weiteren sieht man innerhalb des Klassentemplatefunktionsrumpfs, wie Templates zur generischen Überprüfung, abhängig vom während des Kompilierens ermittelten Argumenttyps, eingesetzt werden.

Nachteil der Templatemetaprogrammierung ist die eher der Funktionalen Programmierung ähnelnde Semantik, die die Lesbarkeit und Nachvollziehbarkeit erheblich senken

kann, sowie die zum Teil schlechte Unterstützung bei der Auswertung von Übersetzungsfehlern.

2.3.2 Aspekte und Templates

Aspektorientierte Programmierung kann zusammen mit Templateprogrammierung auf zwei verschiedene Arten eingesetzt werden. [17]

- Zum einen kann Aspektcode durch Joinpoints in generischen Komponenten eingebracht werden, um querschneidende Belange zu realisieren. Da die Implementierung von generischen Datenstrukturen und Funktionen einen häufigen Einsatzzweck für Templates darstellt, ist diese Funktion auch mit *AspectC++* möglich.

  Allerdings war das Template-Matching für die in dieser Arbeit benutzten Version von *ac++*[1] noch als experimentelles Feature deklariert, weswegen z. B. die durch Aspektcode zu ersetzende Methode für die Überprüfung der Zeigeradresse beim *GuardPointer* in Abschnitt 5.2.2 in einem eigenen Interface realisiert wurde.

- Zum anderen kann generischer Code in Aspekten benutzt werden, um auch bei Trennung der Belange generische oder sogar generative Programmiertechniken einsetzen zu können und so adaptive Lösungen zur Übersetzungszeit zu schaffen. Unterstützung für die Benutzung von Templates zur Übersetzungszeit, bietet die vorher in 2.2 besprochene *Joinpoint API*, die statische Joinpointinformationen zu bspw. Argumenttyp, Rückgabetyp oder Argumentanzahl liefert.

  Advices die solche Joinpoint-spezifische, statische Typinformationen verwenden, werden auch *generische Advices* genannt oder im Speziellen, falls sie die Informationen zur Instanziierung des Metaprogramms verwenden auch *generative Advices* [17].

Insgesamt besitzt *AspectC++*, gerade wegen der möglichen Kombination von Aspekten und Templatemetaprogrammierung, eine große Ausdrucksstärke bzw. einzigartige Sprachmöglichkeiten, die so in keiner anderen Programmiersprache vorhanden sind.

Ein Beispiel für diese Fähigkeiten ist vor allem in Kapitel 6 zu finden, in dem generative Advices benutzt werden, um bedingte und konfigurierbare Plausibilitätsprüfungen zu generieren.

[1] benutzte Version: *ac++* 1.1

3 Fehlertoleranz

Fehlertoleranz bezeichnet die Fähigkeit eines Systems zur Aufrechterhaltung der Systemstabilität bei Auftreten eines Fehlern. Zu den Merkmalen, die die Systemstabilität bzw. die Resilienz eines Systems gegenüber Fehlern charakterisieren gehören (vgl. [18])

- die **Verfügbarkeit** („availability“) eines Systems, d. h. die Funktionsbereitschaft sowie Reaktionsbereitschaft in einem gewissen Zeitraum,
- die **Zuverlässigkeit** („reliability“), d. h. die Fähigkeit über einen bestimmten Zeitraum hinweg korrekt zu funktionieren,
- die **Sicherheit** („safety“), d. h. die Verhinderung katastrophaler Folgen eines Fehlers, die Mensch oder Umwelt gefährden können,
- die **Integrität** („integrity“), die sowohl Daten- als auch Systemintegrität einschließt,
- sowie die **Wartbarkeit** („maintainability“), also die Fähigkeit Erweiterungen oder Reparaturen durchzuführen.

Neben Fehlertoleranz, also der Vermeidung von Auswirkungen einer Störung, die die Systemstabilität gefährden, gibt es noch weitere Verfahren um mit Fehlern umzugehen. Dazu zählen etwa die Fehlerprävention, d. h. das ein System oder eine Software dahin gehend entworfen wird, um das Auftreten bestimmter Fehler erst gar nicht zu ermöglichen, die Fehlerelimination, die vor allem während der Entwurfsphase durch Testen und Verifikation geschieht, sowie die Fehlervoraussage um fehleranfällige Bereiche zu identifizieren.

In den folgenden Abschnitten soll der Fehlerbegriff näher spezifiziert werden, sowie Ursachen und Propagation von Fehlern erläutert werden. Im Anschluss wird auf den Begriff der Fehlertoleranz genauer eingegangen sowie Verfahren und Beispiele zur Fehlererkennung und Fehlerbehebung gebracht. Im letzten Abschnitt dieses Kapitels werden dann Möglichkeiten der Evaluation von Fehlertoleranzmechanismen, sowie im speziellen das für diese Arbeit eingesetzte Fehlerinjektionswerkzeug *Fail** vorgestellt.

3.1 Fehlerbegriffe und -ursachen

Während in der deutschen Sprache meist nur der Begriff Fehler verwendet wird, werden hier drei grundlegende Fehlerbegriffe nach [18] unterschieden, wobei der Fokus des Fehlerbegriffs hier auf Computersystemen liegt.

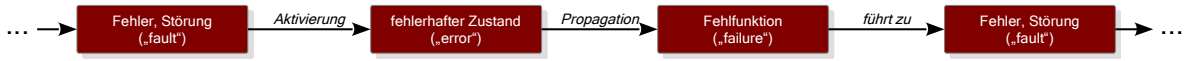


Abbildung 3.1: Grundlegende Fehlerbegriffe und Reihenfolge ihres Auftretens in einer Fehlerkette nach [18].

1. Fehler im Sinne von Störung („**fault**“) oder Defekt, ist die Ursache eines fehlerhaften Zustands. Faults sind entweder aktiv (active), wenn sie zu einem Error führen, oder können ruhend (dormant) sein.

   Beispiele solcher internen Störungen sind Bitfehler durch defekte Schaltkreise oder fehlerhafte Instruktionen bzw. Software-Bugs, durch Fehler bei der Softwareentwicklung. Zu den externen Störungen zählen unter anderem Bitfehler durch Strahlung oder Störungen als Folge des Betriebes außerhalb von Systemspezifikationen.

   Die Aktivierung eines ruhenden Defekts muss dabei nicht immer systematisch reproduzierbar sein oder bspw. an derselben Stelle im Programmfluss erfolgen, sondern kann durch Zusammenspiel mehrere interner und externer Faktoren aktiviert werden. Man spricht dabei von harter bzw. weicher Reproduzierbarkeit. Gerade Fehler mit weicher Reproduzierbarkeit führen bei Verfahren wie Fehlerelimination und Fehlerprävention häufig zu Problemen.

2. Durch Aktivierung führt diese Störung in ihrer Folge zu einem fehlerhaften Systemzustand („**error**“). Diese Manifestation einer Störung kann z. B. eine fehlerhafte Instruktion, wie eine Division durch Null, infolge eines Softwarebugs sein und kann sich über Komponenten hinweg räumlich oder zeitlich fortpflanzen.

   Die Erkennung solcher fehlerhafter Zustände („error detection“) stellt dabei die Voraussetzung für die Fehlertoleranz („fault tolerance“) dar, die versucht nach der Entdeckung eines fehlerhaften Zustands mittels Fehlerbehandlung („error handling“) und eventuellen Maßnahmen die das zukünftige Auftreten der Störungen verhindern („fault handling“), das System wieder in einen Zustand zu überführen, der vor dem Auftreten des Fehlers vorlag.

3. Die Auswirkung eines fehlerhaften Systemzustands bzw. die Fehlerpropagation führt zur Fehlfunktion („**failure**“). Dies kann bspw. den Absturz eines Betriebssystems oder das Versagen einer Komponente bedeuten.

   Fehlfunktionen können sich dabei auf den Inhalt („content failures“), durch Liefern fehlerhafter Werte, das Timing („timing failures“), durch Lieferung zu einem falschen Zeitpunkt, oder beides auswirken.

Wie in Abbildung 3.1 zu sehen können Fehlfunktionen zu weiteren Störungen führen und so die Fehlerkette verlängern. Fehlerketten müssen dabei nicht alle Stationen durchlaufen und jedes Mal zum Ausfall einer Komponente führen, sondern können z. B. durch Isolationsmechanismen, die die Propagation eines Fehlers verhindern oder Nichtausführung eines ruhenden Defekts, unterbrochen werden.

3.1.1 Ursachen und Klassifikation von Fehlern

Für Störungen, also Fehlern im Sinne von „faults“, kann eine Vielzahl von Ursachen verantwortlich sein, wobei man eine grobe Klassifizierung wie die folgende vornehmen kann.

- **Softwarefehler**, also Fehler, die beim Softwaredesign oder durch Fehler bei der Implementierung entstehen, sind meist interne Fehler. Selbst wenn negative Faktoren wie zeitlicher Druck bei der Softwareentwicklung keine Rolle spielen, können Fehler während der Implementierung oder auch durch nachträglich eingebrachte Änderungen nie ausgeschlossen werden.

  > „Whether executable specifications or specification implementations, programs are really not much more than the programmer's best guess about what a system should do.“ [19]

  Beispiele für solche Softwarefehler können nachlässige Implementierungen mit daraus hervorgehenden Speicherlecks, fehlerhafte Instruktionen wie Division durch Null, Zugriffe auf nicht initialisierte Zeiger oder nicht-terminierende Schleifen sein.

- **Hardwarefehler** sind physische Defekte, deren Ursache äußere Einflüsse wie z.B. Strahlung oder Schwankungen der Versorgungsspannung sind, oder ihren Ursprung bei inneren Defekten wie beschädigten Kondensatoren oder Kurzschlüssen haben. Daneben können auch Fehler im Entwurf ausschlaggebend sein. Im Gegensatz zu Software können bei Hardware Defekte im Laufe der Einsatzzeit eines Systems durch Alterung oder Verschleiß entstehen.

  Während Planungs- oder Designfehler permanent sind, können Fehler durch etwa äußere Einflüsse auch transienter Natur sein. Permanente Fehler können häufig zum Ausfall einer Komponente führen, wobei meist nur das Ausweichen auf eine andere Komponente als Lösungsmöglichkeit bleibt.

  Gerade in Zukunft werden transiente Fehler durch Strahlung, sog. Soft Errors (vgl. [20]) immer häufiger vorkommen, die sich meist durch Bit-Flips oder Cross-Talk manifestieren. Die Anfälligkeit moderner Hardware ist dabei vor allem den immer kleineren Packungsdichten und niedrigeren Versorgungsspannungen geschuldet.

- **Interaktionsfehler** gehören zu den externen Fehlern, die bei der Benutzung des Systems oder der Software auftreten. Dabei muss hier zwischen fahrlässigen Fehlern, wie das Betreiben eines Systems außerhalb der zulässigen Spezifikationen, und Fehlern die durch schlechtes Design oder mangelnde Sorgfalt bei der Beschränkung möglicher Inputs ermöglicht werden. So ist etwa eine falsche Eingabe, die zum Absturz eines Softwaresystems führt, eher mangelnder Sorgfalt bei der Implementierung zuzurechnen.

3.1.2 Fehlerpropagation

Durch die Korrelation und Verbundenheit von Komponenten, z. B. Prozesse, die in einem gemeinsamen Speicher laufen oder Systeme, die über ein Bussystem kommunizieren,

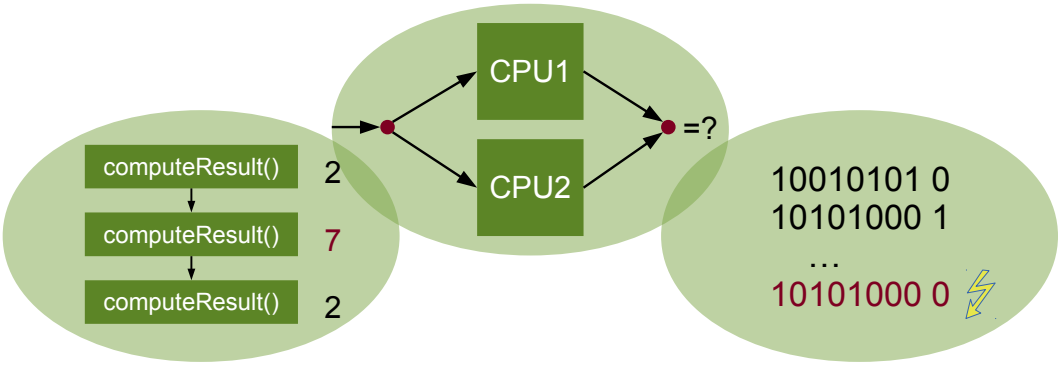


Abbildung 3.2: Verschiedene Arten der Fehlererkennung nach Redundanzart. Links: Zeitredundanz durch wiederholte Ausführung; Mitte: Ortsredundanz durch parallele Ausführung auf mehreren physichen CPUs; Rechts: Informationsredundanz mittels Paritätsprüfung.

können Fehler in einer Komponente zur Fortpflanzung eines Fehlers oder zum Entstehen neuer Fehler in einer anderen Komponente führen.

Diese Fehlerpropagation (vgl. [21]) kann dabei sowohl den Programmfluss anderer Komponenten beeinflussen, etwa durch einen korrumpierten Instruktionszeiger oder eine fehlerhafte Rücksprungadresse, den Datenfluss korrumpieren, durch das Überschreiben von Daten eines anderen Prozesses oder das Timing stören, wie das Ignorieren von Zeitschlitzzuteilungen bei Netzwerkprotokollen.

Zur Vermeidung von Fehlerfortpflanzung kommen verschiedene Isolationsmechanismen zum Einsatz, deren wichtigste Anforderungen ein modularer Aufbau des Systems zur Isolation der einzelnen Komponenten ist. Gerade die Schnittstellen zwischen den Komponenten müssen dabei mit Sorgfalt implementiert werden, um eine Propagation von einer Komponente zur nächsten zu unterbinden.

Isolationsmechanismen können vor allem räumlich, also durch Segmentierung des Adressraums etwa durch Schutzdomänen, Sandboxing oder zur Übersetzungszeit durch Analyse, aber auch zeitlich, wie etwa der Zuteilung von Zeitschlitzen durch geeignetes Scheduling oder mit Hilfe von entsprechenden Protokollen, wirken.

3.2 Fehlertoleranzmechanismen

Fehlertoleranz schließt zum einen die Fehlererkennung als auch die Fehlerbehebung mit ein. Im Folgenden soll auf beide Mechanismen näher eingegangen werden. Fehlertoleranz kann sowohl zur Laufzeit erfolgen oder durch Härtung der Software (u. a. [5]) erreicht werden, wobei hier vor allem Ersteres behandelt wird.

3.2.1 Fehlererkennung

Fehlererkennung basiert auf einer der folgenden Redundanzarten bzw. einer Kombination dieser.

- **Zeitredundanz** erfolgt dabei durch redundante Ausführung derselben Berechnung bzw. desselben Dienstes. Die Ausführung findet dabei sequenziell statt und verursacht so höhere Laufzeitkosten. Zeitredundanz eignet sich dabei hauptsächlich zum Auffinden transienter Fehler, da permanente Fehler aber auch periodisch auftretende Fehler, die mit derselben Frequenz wirken mit der die Erkennung erfolgt, mit Hilfe redundanter Ausführung nicht gefunden werden können.
- **Ortsredundanz** bzw. Replikation erfolgt durch die parallele Berechnung, die im allgemeinen zusätzliche Hardware erfordert. Damit wird zwar die Geschwindigkeit der Ausführung nicht beeinträchtigt, aber die Kosten für Hardware mindestens verdoppelt.
- **Informationsredundanz** nutzt zusätzliche Informationen zur Fehlererkennung, wie bspw. fehlererkennende aber auch fehlerkorrigierende Codes (z. B. Hamming-Code) bzw. Prüfsummen. Dabei ist die Anwendung der Informationsredundanz üblicherweise auf Einsatzgebiete beschränkt wo Informationen übertragen oder gespeichert werden, aber auch das Konzept der Diversität kann zur Informationsredundanz gerechnet werden.

Dabei kann keine dieser Redundanzen eine vollständige Fehlererkennungsabdeckung leisten, die Zahl möglicher latenter Fehler aber zumindest drastisch reduzieren.

In Abbildung 3.2 sind die verschiedenen Möglichkeiten der Redundanz exemplarisch dargestellt, wobei dies im Beispiel der Zeitredundanz durch wiederholte Ausführung und den anschließenden Vergleich der Ergebnisse geschieht, im Beispiel der Ortsredundanz durch parallele Ausführung jeder Instruktion und Vergleich der Ergebnisse und im Beispiel der Informationsredundanz durch Pritätsprüfung.

3.2.1.1 Verfahren zur Fehlererkennung

Die möglichen Verfahren zur Fehlererkennung sind so zahlreich, dass hier nur einige exemplarisch genannt werden. Zudem sind einige dieser Verfahren zusätzlich fähig, Fehler nach der Entdeckung zu beheben und werden im folgenden Abschnitt behandelt.

- Wie schon erwähnt, kommen **Prüfsummen** bzw. fehlererkennende Codes gerade dort zum Einsatz, wo Informationen gespeichert oder übertragen werden. So wird bspw. die Zyklische Redundanzprüfung (CRC) bei Ethernet oder Hammingcodes bei diversen Speichercontrollern verwendet um Fehler zu erkennen. Prüfsummenverfahren werden dabei sowohl in Hard- als auch in Software umgesetzt und bieten ein gutes Verhältnis zwischen Kosten und Fehlerabdeckung.
- Durch **Speicherschutz** wie bspw. MMU[1], MPU[2] aber auch ohne spezielle Hard-

[1] Memory Management Unit
[2] Memory Protection Unit

ware, wie etwa durch Sandboxing (siehe [22]) können Komponenten isoliert und von ihnen ausgehende Fehler erkannt werden.

- Erkennung durch den Prozessor (processor exceptions) oder **spezielle Hardware** wie etwa ein Watchdog-Timer (vgl. [23]) um Timingfehler wie Verklemmungen aufzuspüren.

- Durch **redundante Ausführung** oder replizierte Recheneinheiten können ebenfalls Fehler leicht erkannt werden, obwohl hier der Aufwand größer als bei anderen Verfahren ist und permanente Fehler etwa im Code nur bedingt gefunden werden können.

- Die vor allem in dieser Arbeit eingesetzte **Plausibilitätsprüfung** bzw. Bereichsprüfung, bei der Daten dahin gehend geprüft werden, ob sie sich in einem validen Wertebereich befinden, stellt ein weiteres Fehlererkennungsverfahren dar. Ein großer Vorteil dieses Erkennungsverfahrens sind die geringen Anforderungen bzgl. Metadaten, die universelle Einsetzbarkeit durch Betriebssystem- und Hardwareunabhängigkeit sowie die Möglichkeit sowohl transiente als auch permanente Fehler zu entdecken. Auch wenn wenige Informationen über die zu prüfenden Daten gegeben sein müssen, um das Verfahren anzuwenden, helfen solche allerdings die Fehlerabdeckung zu erhöhen. Die eher niedrigen Kosten dieses Verfahrens erlauben aber leider auch nur eine geringere Fehlerabdeckung gegenüber anderen Verfahren.

Auch wenn einige dieser Fehlererkennungsverfahren in Hardware erfolgen oder zusätzliche Hardware erfordern, konzentriert sich diese Arbeit ausschließlich mit Fehlertoleranzmechanismen in Software. Allerdings sind die Grenzen zwischen Fehlertoleranz in Software und Hardware manchmal fließend, gerade wenn Software Hardwaregegebenheiten ausnutzt oder Hardware auf eine Unterstützung durch Software angewiesen ist.

3.2.2 Fehlerbehebung

Unter Fehlerbehebung bzw. Fehlerbehandlung versteht man die Wiederherstellung eines Systemzustands, der vor dem Eintreten eines Fehlers vorgefunden wurde. Zusätzlich zur Wiederherstellung eines fehlerfreien Systemzustands zählen auch Maßnahmen zur Vermeidung der Wiederholung eines entdeckten Fehlers dazu.

3.2.2.1 Verfahren zur Fehlerbehebung

Bei den Verfahren, die zur Fehlerbehebung eingesetzt werden, kann grob zwischen Einzelversions- und Multiversionsverfahren (vgl. [21],[19]) unterschieden werden. Im Gegensatz zu Einzelversionsverfahren existieren bei Multiversionsverfahren mehrere unterschiedliche Versionen einer Systemkomponente oder eines Programms, die parallel oder sequenziell ausgeführt werden und im Normalfall für dieselben Inputdaten zum selben Output führen.

Da sich Multiversionsverfahren zwar hervorragend zur Kompensation von Fehlern eignen, in der Praxis aber eher schwer zu realisieren sind, werden hier nur einige dieser Verfahren am Rande erwähnt.

- Einzelversionsverfahren
  - Durch **Checkpoints**, die entweder statisch, d. h. an bestimmten festgelegten Kontrollpunkten oder dynamisch angelegt werden, kann im Falle eines detektierten Fehlers ein **Rollback** stattfinden. Checkpoints sind nur dort anwendbar, wo keine Seiteneffekte auftauchen.
  - Durch Einnahme eines neuen, fehlerfreien Zustandes wird das System mit einem sog. **Rollforward** wieder lauffähig gemacht. Dies ist meist nur bei Systemen möglich die irgendwann einen Zustand erreichen, der nicht von Zustandsinformationen des vorigen abhängig ist.
  - Durch **Replikation** kann beim Ausfall einer Komponente eine andere Komponente deren Aufgaben übernehmen.
  - Fehlerkorrigierende Codes und andere **Kompensationsmaßnahmen** können eingesetzt um auftretende Fehler zu maskieren.
  - Zur Vermeidung des zukünftigen Auftretens entdeckter Fehler können **Isolation** oder Abschaltung einzelner Komponenten beitragen oder eine **Reinitialisation** des Systems, etwa durch einen Neustart, vorgenommen werden.
- Multiversionsverfahren
  - Bei den **Recovery-Blocks** wie in Abbildung 3.3 zusehen, wird ein Akzeptanztest zur Validitätsprüfung des Ergebnisses der aktuellen Version eingesetzt, wobei im Falle eines entdeckten Fehlers eine alternative Version benutzt wird. Herausforderung bei diesem Verfahren ist die Realisation des Akzeptanztests.

    Zu beachten ist, dass evtl. Checkpoints für einen Rollback verwendet werden müssen, falls durch die Versionen Änderungen des Systemzustands erfolgen, die durch einen Wechsel auf eine andere Version rückgängig gemacht werden müssen.
  - Beim **N-Version Programming** wird aus den Ergebnissen aller Versionen ein Ergebnis mittels eines Selektionsalgortihmus bestimmt. Die Bestimmung des Ergebnisses kann z. B. durch Voting, also die Wahl des häufigsten Ergebnisses, geschehen.
  - Neben den beiden bereits erwähnten Multiversionsverfahren existieren noch eine Reihe weiterer Verfahren, die auf verschiedenen Kombinationen bzw. Variationen dieser beruhen (s. a. [21]).

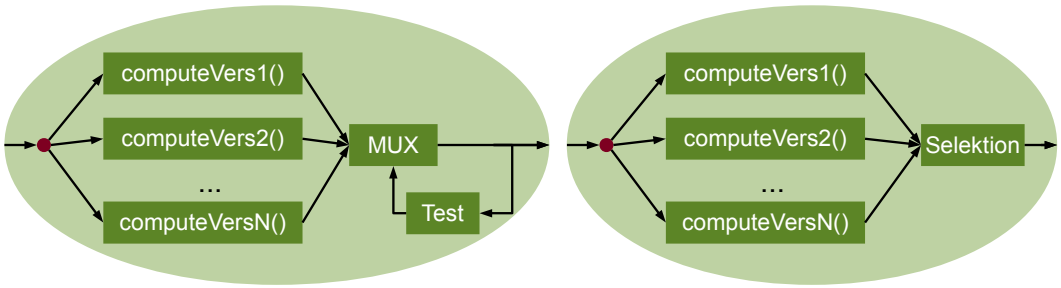


Abbildung 3.3: Zwei Multiversionsverfahren zur Fehlerbehebung. Recovery-Blocks (links) die mit Hilfe eines Akzeptanstests eine korrekte Version auswählt und N-Version (rechts) die mittels Selektionsalgorithmus die wahrscheinlich korrekte Version auswählt.

3.2.3 Fehlertoleranz und AOP

Gerade weil Fehlertoleranz einen querschneidenden Belang darstellt, eignet sich *AOP* sehr gut um konfigurierbare Fehlertoleranzmechanismen umzusetzen. Deshalb sollen hier einige vorangegangene und verwandte Arbeiten, aus dem Bereich der Softwarefehlertoleranz zusammen mit *AOP*, vorgestellt werden.

In der Arbeit von Alexandersson und Karlsson [24] wird *AspectC++* verwendet um Fehler im Kontrollfluss zu entdecken („Double Signature Control Flow Checking“). Hierzu wird vor jedem Funktionsaufruf und jeder Funktionsausführung eine Identifikationsnummer der jeweiligen Funktion auf den Stack gelegt, die nach Rücksprung wieder vom Stack geholt wird, um sie mit der aktuellen Funktionsnummer zu vergleichen. Ein Fehler im Kontrollfluss, wie z. B. die Ausführung einer anderen Funktion, würde zu einer Falsifikation des Vergleichs und somit zur Detektion führen.

Ein weiterer auf Mehrfachausführung basierender Fehlertoleranzmechanismus ist das ebenfalls von Alexandersson und Karlsson [24] vorgestellte Verfahren der TTR-FR („Triple Time-Redundant Execution with Forward Recovery“), bei der durch dreifache hintereinanderfolgende Ausführung derselben Funktion und einem anschließenden Vergleich der Ergebnisse, Fehler im Falle einer einzelnen Abweichung sogar korrigiert werden können. Allerdings hat dieser FT-Mechanismus einen sehr hohen Performanzoverhead und lässt sich nur auf einfache Funktionen ohne Seiteneffekte anwenden, die für einen gegebenen Input ein Ergebnis berechnen. Neben dem Laufzeitoverhead sind zudem globale Ergebnisvariablen für die Zwischenspeicherung bis zum Vergleich und Statusvariablen, um bspw. die verbleibende Anzahl an nötigen Wiederholungen und aktuelle Ergebnisvariable zu erfassen, was den Einsatzzweck dieses Mechanismus stark einschränkt.

Ein ähnlicher, aber für einen etwas spezielleren Einsatzzweck entwickelter Ansatz zur Fehlertoleranz mittels *AOP*, nämlich dreifache Redundanz mit Vorwärtsfehlerkorrektur, wird in der Arbeit von Borchert et al. [25] verfolgt, in dem der Zeiger auf die Tabelle virtuelle Methoden („vptr“) für jedes Objekt zweimal kopiert und gespeichert und vor jedem Aufruf überprüft wird („Triple-modular Redundant Vptrs“). Die Validität des *vptr* kann dabei dadurch ermittelt werden, in dem geprüft wird ob dieser in einen, speziell

für diese Art von Zeiger durch den Linker ausgewiesenen Speicherbereich, zeigt.

Ein weiteres von Borchert et al. [26] vorgestelltes Verfahren, wird zur generischen Objektüberwachung eingesetzt. Bei dieser Objektüberwachung werden alle Membervariablen einer kritischen Klasse per ECC geschützt. Bevor ein Methodenaufruf dieser Klasse stattfindet, werden die Variablen überprüft und können evtl. korrigiert werden. Dabei folgt nach Beendigung des Methodenaufrufs eine Neuberechnung des Fehlerkorrekturcodes. Gerade dieses Fehlertoleranzverfahren lässt sich äußerst universell einsetzen und verfügt über eine hohe Fehlererkennungsrate bei niedrigem Overhead.

3.3 Bewertung von Fehlertoleranz

Um das Verhalten eines Systems im Falle eines Fehlers zu beobachten und zur Bewertung von Fehlertoleranzlösungen, ist man auf die gezielte Fehlerinjektion („fault injection“) angewiesen. Mit Hilfe von Fehlerinjektion (kurz FI) ist es möglich eine bestimmte Fehlerart, an einem bestimmten Ort und zu einem bestimmten Zeitpunkt in das System einzubringen.

3.3.1 Fehlerinjektionstechniken und -modelle

Die Techniken zur Fehlerinjektion (vgl. [27]) lassen sich dabei in hardwarebasierte, softwarebasierte, simulationsbasierte und hybride FI-Techniken einteilen.

- **Hardwarebasierte** FI kann dabei einerseits zur Simulation möglicher, grenzwertiger Umweltbedingungen z. B. durch Strahlung, Temperatur oder Versorgungsspannung eingesetzt werden, oder zur Pin-Level-FI d. h. zur Injektion von Fehlern direkt an den Kontakten des Prozessors. Nachteil dieser Methode sind einerseits die Kosten durch evtl. Zerstörung der Zielhardware und die Kosten der benötigten, komplexen Hardware zur Fehlerinjektion und Auswertung andererseits.

- **Softwarebasierte** FI kann gezielte Manipulation von Teilen der Software einsetzen, wie das Ersetzen einzelner Instruktionen zur Übersetzungszeit oder Fehlergeneratoren innerhalb eines Programms zur Laufzeit, um Fehler zu erzeugen. Die manipulierte Software läuft dabei auf der Zielhardware und die Auswertung kann entweder durch Software oder Hardware erfolgen. Ein Nachteil dieses Verfahrens ist der beschränkte Fehlerinjektionsraum, da bestimmte Fehler nur schwer durch Software zu erreichen sind.

- Bei der **simulationsbasierten** FI wird die Zielhardware simuliert, um eine größtmögliche Flexibilität bei der FI zu erreichen. Sowohl Ort, Timing als auch gewünschte Fehlerart für die Einbringung können je nach Simulator bei dieser Methode frei gewählt werden. Auch können bei diesem Verfahren nur Teile des Systems oder bestimmte Systemzustände simuliert und so Testzeiten verkürzt werden. Nachteil dieser Technik sind die Kosten der Implementierung und die möglichen Einschränkungen in der Genauigkeit eines Simulationsmodells.

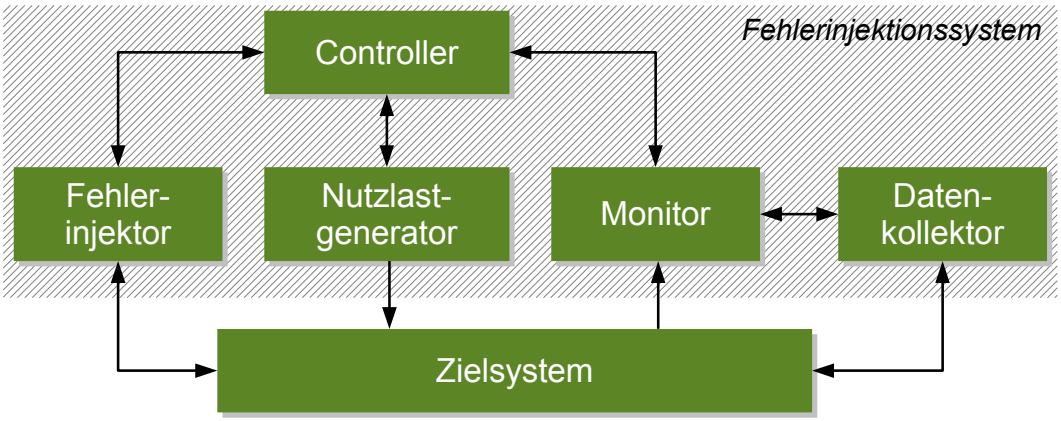


Abbildung 3.4: Modell eines Fehlerinjektionssystems nach [29].

3.3.1.1 Fehlermodell

Je nach zu erwartenden Fehlern und abhängig von der Betrachtungsebene können verschiedene Fehlermodelle (s. a. [28]) eingesetzt werden. Diese spezifizieren, wann, wo und welche Art von Fehler injiziert werden soll.

- Als **Ort** kommen z. B. Register, Bus, Speicher bzw. Speichersegmente oder ALU[3] infrage.
- Der **Zeitpunkt** einer FI kann sich dabei auf die Systemzeit oder -zyklen beziehen, z. B. eine bestimmte Zeit nach Systemstart oder durch Eintreffen eines bestimmten Ereignissen ausgelöst werden, wie das Erreichen einer festgelegten Instruktionsadresse oder der Zugriff auf eine spezielle Speicheradresse.
- Um die Auswirkungen eines bestimmten Fehlers auf die eingesetzte Software, wie Soft-Errors oder permanente Defekte, zu simulieren bzw. abzuschätzen, können verschiedene **Fehlertypen**, wie Bit-Flips (single/multiple), Stuck-At oder Parameter Fuzzing im Fehlermodell eingesetzt werden, wobei jeder Fehlertyp bestimmte Fehler repräsentieren kann, mit unterschiedlich guter Fehlerabdeckung.

3.3.2 Fehlerinjektionssysteme

Zur Umsetzung von FI-Testreihen und der Aufzeichnung der Ergebnisse werden Fehlerinjektionssysteme verwendet, die grundlegend aus den in Abbildung 3.4 gezeigten Komponenten bestehen (vgl. [29]).

Um die einzelnen Fehlerinjektionsexperimente zu verwalten, sowie das eingesetzte Fehlermodell zu bestimmen, wird ein Controller eingesetzt, der die benötigten Informationen an den Fehlerinjektor, den Nutzlastgenerator und den Monitor propagiert.

Der Fehlerinjektor ist für die direkte Injektion des Fehlers ins Zielsystem, abhängig vom verwendeten Fehlermodell, verantwortlich, während der Nutzlastgenerator bspw.

[3] Arithmetisch-logische Einheit

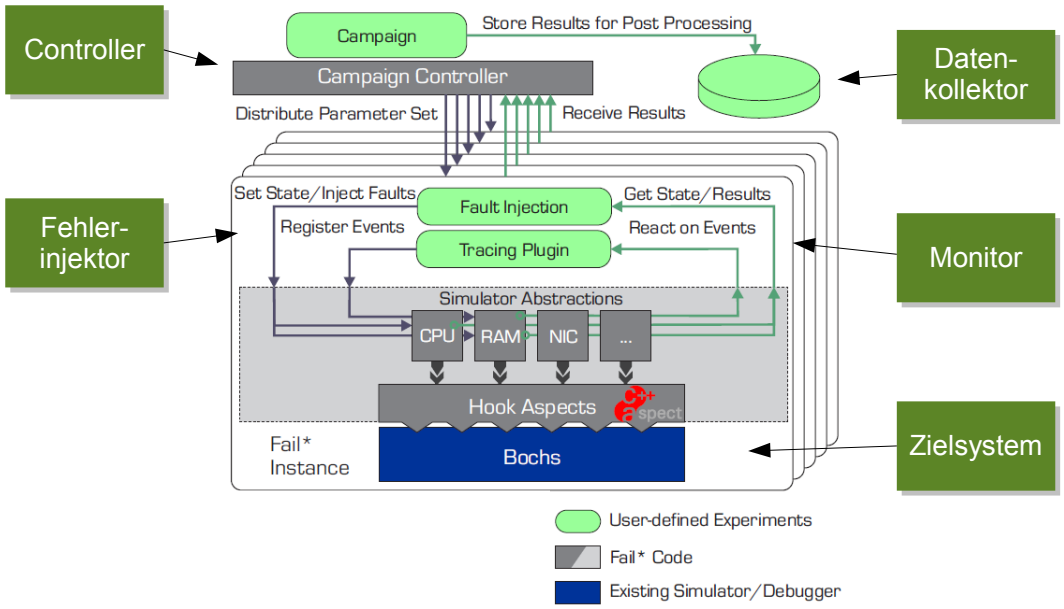


Abbildung 3.5: Architektur von *FAIL** aus [30] mit Vergleich zum allgemeinen Fehlerinjektionssystem aus Abschnitt 3.3.2.

eine Testapplikation ausführt. Die durch den Controller spezifizierten und durch die FI gewonnenen Daten werden vom Monitor zusammen mit Daten über das Verhalten des Zielsystems an den Datenkollektor weitergereicht und können in einem möglichem, folgendem Speicher- oder Analyseschritt weiterverarbeitet werden.

3.3.2.1 FAIL*

Eine Implementierung eines Fehlerinjektionssystems stellt das ebenfalls auf *AOP* basierende und für den Evaluationsteil in dieser Arbeit benutzte Fehlerinjektions Framework *FAIL**[4][30] dar.

Neben diversen Fehlerinjektionssystemen, die man grob in Generalisten, d. h. FI-Tools, die zwar ein flexibles Back-End für den einfachen Austausch des Zielsystems bieten, deren Zugriff aus dem Experimentcode auf das Zielsystem und damit die Möglichkeiten der Fehlerinjektion aber eingeschränkt sind, und Spezialisten, die das gegensätzliche Problem aufweisen, einteilen kann, versucht *FAIL** die Vorteile beider Typen zu vereinen (vgl. [31]).

In Abbildung 3.5 ist der schematische Aufbau von *FAIL** zu sehen. Zum Vergleich mit dem allgemeinen Aufbau eines Fehlerinjektionssystems aus dem vorigen Abschnitt sind die jeweiligen zugehörigen Komponenten beschriftet. Die Vorteile von *FAIL** sind dabei vor allem die Austauschbarkeit des Zielsystems wie bspw. dem in dieser Arbeit benutzten x86-Emulator *Bochs* (siehe [32]), sowie die Möglichkeit Experimente verteilt und parallel durchführen zu können.

[4] Fault Injection Leveraged; * für das austauschbare Zielsystem

Um FI-Experimentreihen mit *FAIL** durchzuführen, muss eine Kampagne erstellt werden, die als Rahmen und zur Spezifizierung bspw. der gesammelten Informationen der einzelnen Experimente dient. Um aus dem Experimentcode auf das Zielsystem zugreifen zu können, stellt *FAIL** eine Framework API bereit, die Zugriff auf Metainformationen zu bspw. Registern oder dem Speicher bietet, Statusinformationen zum Zielsystem bereitstellt, sowie zur Registration von Ereignissen dient. Durch diese Abstraktion von Funktionen und Eigenschaften des Zielsystems ist der erstellte Experimentcode auch für andere Zielsysteme wiederverwendbar.

Da die Anzahl der benötigten Experimente von den drei Dimensionen des Fehlermodells abhängig ist, kann diese schnell sehr groß werden. Da viele der möglichen Experimente zwangsläufig zum selben Ergebnis führen, können einige dieser Experimente eingespart werden und so die Laufzeit einer kompletten Kampagne reduziert werden. Diese auch von *FAIL** eingesetzte Ausdünnung des Fehlerraums („Fault-Space-Pruning“) sorgt z. B. dafür, dass Fehlerinjektionen in eine Speicherstelle die bis zum nächsten Schreibzugriff nicht mehr gelesen wird, ausgelassen werden und die benötigten Experimente reduziert werden.

4 Eingebettete Betriebssysteme

> „Embedded systems can be defined as information processing systems embedded into enclosing products such as cars, telecommunication or fabrication equipment. Such systems come with a large number of common characteristics, including real-time constraints, and dependability as well as efficiency requirements.“ [33]

Betriebssysteme für eingebettete Systeme sind für einen speziellen Einsatzzweck geschaffen und müssen, im Gegensatz zu Vielzweckbetriebssystemen wie z. B. *Microsoft Windows*, strenge Anforderungen in Bezug auf den Ressourcenverbrauch erfüllen.

Zu den typischen Komponenten von eingebetteten Betriebssystemen gehören z. B.

- Task- bzw. Prozessor-Management,
- Unterbrechungsbehandlung,
- Synchronisation,
- grundlegende E/A-Operationen bzw. Treiber,
- Speichermanagement
- und Zeitdienste.

Auch harte oder weiche Echtzeitanforderungen sowie die Verlässlichkeit des Systems sind in diesem Bereich ein häufiges Kriterium.

4.1 Konfigurierbare Betriebssysteme

Da maßgeschneiderte Betriebssysteme für einzelne Lösungen häufig viel zu teuer und aufwendig sind, stellt die Konfigurierbarkeit eine weitere Anforderung an eingebettete Betriebssysteme dar. Dabei sollen vor allem nicht benötigte Teile weggelassen werden können, um deren verursachten Overhead einzusparen.

Zu den Möglichkeiten diese Variabilität für eingebettete Betriebssysteme effizient umzusetzen, können Konzepte wie die der Software-Produktlinien (vgl. [8]) verwendet werden. Derweil kann die Implementierung der Konfigurierbarkeit etwa durch

- konditionale Übersetzung bzw. Bindung mittels des *C-Präprozessors*, wie in *eCos*,
- die in dieser Arbeit verwendete Aspektorientierte Programmierung (*AOP*),

- durch Konzepte wie Polymorphie bei der Objektorientierte Programmierung (*OOP*)
- sowie generischer bzw. generativer Programmierung

erfolgen.

AOP hat im Bereich der konfigurierbaren, eingebetteten Betriebssysteme durchaus seine Vorteile, durch die Trennung von Belangen und seine Verwendbarkeit, durch niedrigen Overhead bei der Implementierung, unter Beweis gestellt. So wurden bspw. ohne signifikante Performanzverluste zu erreichen, Fehlertoleranzmechanismen mittels Aspekten, in das Echtzeitbetriebssystem *BOSS* eingebracht (siehe dazu [34], [35]). Auch bei der Untersuchung von Laufzeit- und Speicherkostenbedarf, durch Umsetzung von Betriebssystemkomponenten in *eCos* mithilfe von *AspectC++*, wurden positive Resultate erzielt (vgl. [4]).

Auch wenn die Aspektimplementierung nicht immer so reibungslos verlaufen muss und von der Struktur des vorhandenen Systems abhängig ist, die evtl. Änderung an der Codebasis verlangt, so ist es möglich, *AOP* direkt beim Systementwurf zu berücksichtigen. Der sog. Aspektgewahre Systementwurf („aspect-aware operating-system development“), der bei der Entwicklung des Betriebssystems *CiAO* (siehe dazu [36]) verfolgt wurde, zeigt, dass *AOP* in allen Bereichen der Betriebssystementwicklung sowohl in Bezug auf Konfigurierbarkeit, als auch bei der Effizienz in Bezug auf Laufzeit und Speicherplatzverbrauch, bei gleichzeitiger Trennung der Belange, eingesetzt werden kann.

4.2 Das eingebettete Betriebssystem eCos

Ein Beispiel eines eingebetteten Betriebssystems, welches zudem als Zielsystem zur Einbringung von Fehlertoleranzmechanismen diente, ist das Open Source Echtzeitbetriebssystem (RTOS[1]) *eCos* (s. a. [37], [38]).

Wie bereits vom Namen *eCos* (embedded Configurable operating system) abzuleiten ist, lag der Hauptschwerpunkt beim Entwurf dieses eingebetteten Betriebssystems, bei der Konfigurierbarkeit und Variabilität.

Ziel beim Design war es, dem Benutzer zu ermöglichen ein System zusammenzustellen, das nur die benötigten Features für einen geplanten Einsatzzweck enthält und dessen Speicherplatzverbrauch dementsprechend mitskaliert. Herausgekommen ist ein Betriebssystem, das mit mehr als 200 Optionen feingranular konfigurierbar ist und dessen Größe von einigen hundert Byte bis zu mehreren Kilobyte reicht.

eCos ist zudem für viele Hardwareplattformen verfügbar, unter anderem *Intel x86*, *ARM*, *MIPS*, *PowerPC*, *SPARC* sowie einigen anderen.

4.2.1 Kernkomponenten von *eCos*

Die Kernkomponenten von *eCos*, die hauptsächlich in *C* bzw. *C++* (Kernel) geschriebenen sind, werden in Abbildung 4.1 dargestellt.

[1] Real Time Operating System

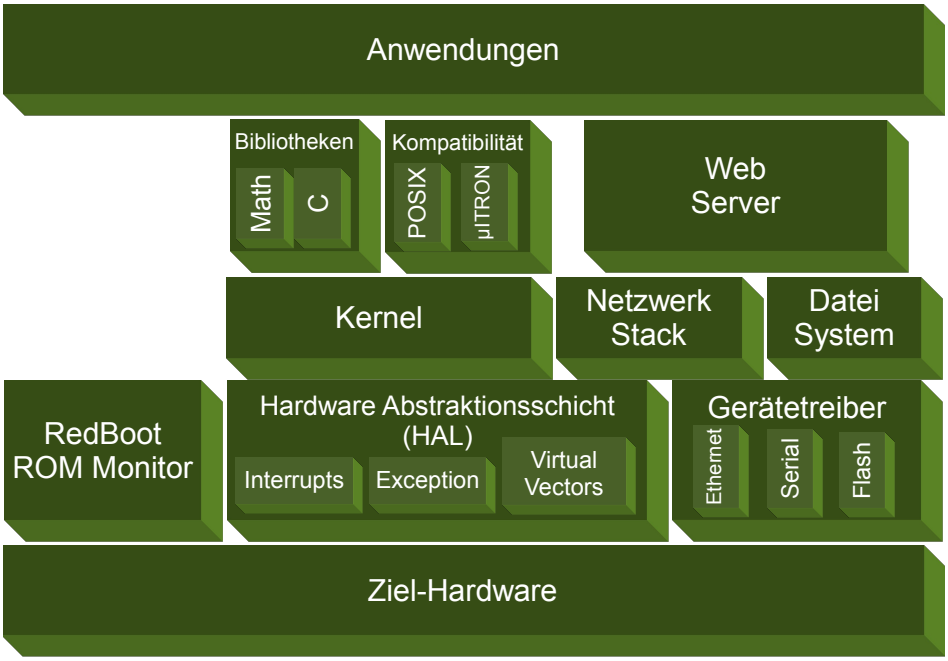


Abbildung 4.1: Die Kernkomponenten des eingebetteten Betriebsystems *eCos* aus [37].

Die Hardwareabstraktionsschicht (HAL[2]) stellt die Systemschicht für grundlegende Hardwarezugriffe dar und isoliert die architekturabhängigen Teile um eine einfache Portabilität des Betriebssystems zu ermöglichen.

Der Kernel von *eCos* stellt neben dem Task-Management und der Interruptbehandlung, auch Synchronisationsmechanismen wie Semaphoren und Mutexe zur Verfügung. Ebenso gehören Zeitdienste wie Alarme und Timer zu dessen Funktionsumfang.

Eine weitere Kernkomponente stellen Gerätetreiber, etwa für Ethernet oder die serielle Schnittstelle, dar, die einen großen Teil von *eCos* ausmachen (vgl. Abbildung 4.4(b)). Dabei kommunizieren diese über die Hardwareabstraktionsschicht mit der eigentlichen Hardware. Auf die Treiberschnittstelle und -architektur wird in Abschnitt 5, bei der exemplarischen Implementierung von Fehlertoleranzmechanismen, noch näher eingegangen.

Ebenfalls in *eCos* enthalten ist eine Implementierung der *ISO C* Bibliothek um häufig verwendete Funktionen und Datenstrukturen für eigene Anwendungen bereitzustellen, aber auch Betriebssystemfunktionen wie Speicherallokation bieten zu können.

Um Benutzern die Umstellung sowie die Portierung von vorhandenen Programmen zu erleichtern, besitzt *eCos* zudem eine Kompatibilitätsschicht, welche eine Schnittstellenimplementierung für *POSIX* und *µITRON* bietet.

[2] Hardware Abstraction Layer

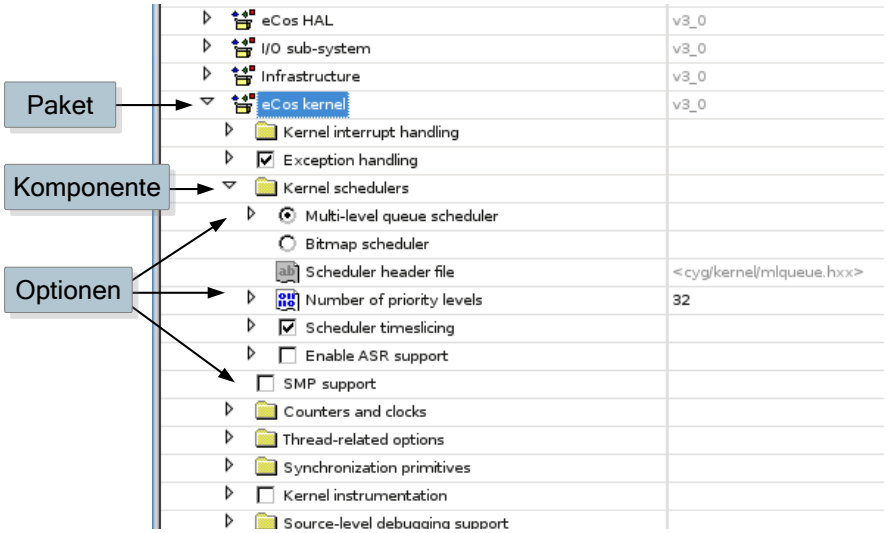


Abbildung 4.2: Das Konfigurationswerkzeug von *eCos* mit graphischer Oberfläche, zur Auswahl von Komponenten und deren Einstellungen.

4.2.2 Konfigurationsmöglichkeiten

Um bei der großen Menge an Optionen zur Konfiguration einen Überblick zu behalten, sowie dem Benutzer diese zu vereinfachen, bringt *eCos* ein Konfigurationswerkzeug inklusive graphischer Oberfläche mit.

Wie in Abbildung 4.2 zu sehen, sind die einzelnen Optionen in verschachtelten Paketstrukturen organisiert. Einzelne Komponenten können dabei je nach Bedarf aktiviert oder deaktiviert, sowie deren Optionen parametrisiert werden, wie die im Beispiel zu sehende Anzahl an Prioritäten für die vom Scheduler verwendete Multi-Level-Queue.

Die Pakete werden in der sog. *Component Description Language* (CDL) verfasst, die die enthaltenen Komponenten und Optionen festlegt, als auch mögliche Abhängigkeiten zu anderen Paketen beschreibt. Aus der gewählten Konfiguration werden schließlich entsprechende Makefiles generiert, die die benötigten Quellcodedateien bestimmen, als auch Headerdateien in denen mithilfe von Makros und `#define`-Anweisungen einzelne Optionen umgesetzt werden.

4.2.3 Nachteile

Gerade die Stärke von *eCos*, nämlich die Konfigurierbarkeit, wird durch die Umsetzung mittels Makros und bedingter Übersetzung, mithilfe von `#ifdef`-Anweisungen, zu einem Nachteil.

Wie am Beispiel der Implementierung der Scheduler-Threadklasse `Cyg_SchedThread`, in Abbildung 4.3 zu sehen, leidet durch die Verwendung zahlreicher Präprozessor-Anweisungen die Lesbarkeit deutlich. Auch die Umsetzung querschneidender Belange mittels Makros, die keine natürlichen Sprachelemente darstellen und so leicht zu Fehlerquellen werden können, schränkt sowohl die Wiederverwendbarkeit als auch die Erweiterbarkeit

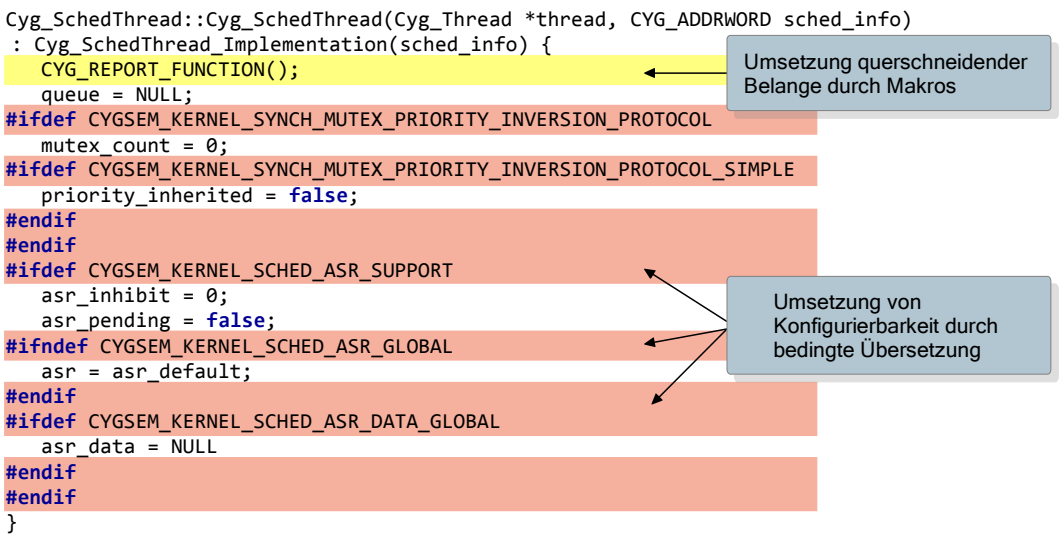


Abbildung 4.3: Umsetzung der Konfigurierbarkeit in *eCos* mittels `#ifdef` und Makros am Beispiel des Konstruktors von `Cyg_SchedThread`.

weiter ein.

Eine Möglichkeit Probleme dieser Art zu umgehen ohne die Performanz einzuschränken bietet bspw. die Verwendung von *AOP* (vgl. [4]). Auch aus diesem Grund wurde in dieser Arbeit, die Implementierung des querschneidenden Belangs der Fehlertoleranz sowie dessen Konfiguration, mittels Aspekten realisiert.

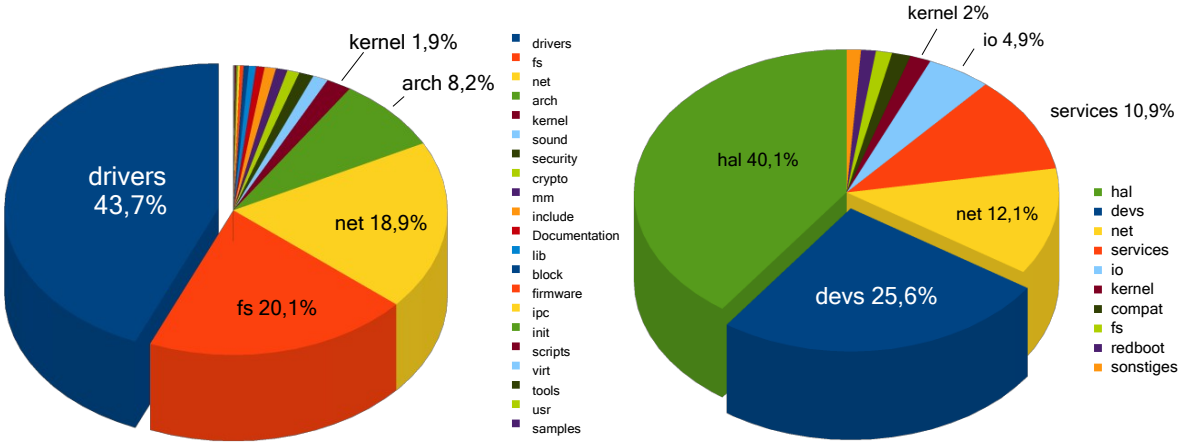
In Bezug auf die Implementierung von Fehlertoleranzmechanismen ist die Schwierigkeit in *eCos*, sowie generell in vielen eingebetteten Betriebssystemen, dass sie kaum über geeignete Isolationsmechanismen verfügen. So gibt es in *eCos* bspw. keine Trennung zwischen Benutzer- und Kernel-Modus oder Maßnahmen zum Speicherschutz. Dies spielt vor allem bei der Auswahl geeigneter Fehlertoleranzverfahren sowie der Möglichkeiten zum Schutz eines solchen eingebetteten Betriebssystems vor Softwarefehlern, eine Rolle.

4.3 Treiber

Gerätetreiber ermöglichen die Interaktion und Kommunikation eines Systems über Hardwarekomponenten mit der Umwelt, was gerade für eingebettete Systeme einen essenziellen Bestandteil darstellt.

Treiber machen üblicherweise einen Großteil des Betriebssystemcodes aus, wie in Abbildung 4.4(a) für das Mehrbenutzer-Betriebssystem *Linux*[3] zu sehen. In Abbildung 4.4(b) ist der Anteil des Gerätetreibercodes in *eCos* dargestellt, wobei hier zu beachten ist, dass die Hardwareabstraktionsschicht in *eCos* zwar den größeren Teil des Quelltexts ausmacht, dies aber der vor allem der Unterstützung von mehr als 16 Prozessortypen geschuldet ist, während die Anzahl an unterstützter Hardwaregeräte für eine dieser Platt-

[3]Kernel 2.6.32.59 mittels `du` ermittelt



(a) Anteil der Treiber in Linux

(b) Anteil der Treiber in eCos

Abbildung 4.4: Anteil des Treibercodes im Betriebsystem *Linux* (drivers), sowie des eingebetteten Betriebsystems *eCos* (devs, io), nach Anteil der genutzten Datenträgerkapazität.

formen zum Teil deutlich geringer ist.

4.3.1 Anfälligkeit von Gerätetreibern

Gerätetreiber sind häufig als Ursache für Abstürze von Applikationen oder sogar des gesamten Systems zu identifizieren. Untersuchungen zeigen, dass bspw. in *Windows XP* z. B. 85% der Betriebssystemabstürze durch Treiber verursacht werden, während etwa in *Linux* die Fehlerfrequenz in Gerätetreibern sieben Mal so hoch ist wie die im restlichen Code (vgl. [2], [39]). Grund für das hohe Fehleraufkommen sind zum Teil sicherlich die Quellen, aus denen die Treiber stammen, da viele Treiber oft von Drittanbietern erstellt werden, die evtl. weniger Erfahrung mit dem Betriebssystem sowie der dortigen Treiberimplementierung haben. Viele Treiber werden aber auch einfach weniger gesichtet und getestet, im Gegensatz zu Quelltext von Kernkomponenten wie dem Kernel.

Ursache dafür, dass bspw. Gerätefehler in vielen Fällen zu einem Treiberabsturz führen, ist häufig, dass viele Treiber ungenügend auf unvorhergesehenes Verhalten der Hardware vorbereitet sind. So können etwa durch transiente Hardwarefehler, an den Treiber kommunizierte Daten fehlerhaft sein, welche in bestimmten Fällen zu Überläufen bei Puffern oder durch Warten auf einen bestimmten Wert, zu endlosem Polling führen können.

Das Verhindern von Treiberfehlern hilft also, die Zuverlässigkeit des gesamten Systems zu erhöhen. Dabei kann grundsätzlich zwischen zwei Vorgehensweisen unterschieden werden.

- Durch **Treiberhärtung**, also das Eliminieren von Fehlern und Schwächen in der Treiberimplementierung zur Übersetzungszeit, können viele Treiberabstürze verhindert werden. Allerdings können nur vorhandene Treiber gehärtet werden, oder

dies müsste beim Hinzufügen neuer Treiber automatisch geschehen. Eine weitere Schwierigkeit bei der Treiberhärtung ist, dass Treiber schwer auf Hardwarefehler zu testen sind, da diese mitunter sehr komplex sein können und die jeweilige Hardware für den Test zur Verfügung stehen muss.

- Eine weitere Vorgehensweise um Gerätetreiberabstürze zu verhindern, ist die **Laufzeitfehlertoleranz von Treibern** zu erhöhen. Eine solche Laufzeitfehlertoleranz wird für *eCos* bei der Absicherung der Treiberschnittstelle in Abschnitt 5 behandelt.

4.3.2 Treiberresilienz

Zu den Maßnahmen bei der Implementierung, um Treiber resistenter gegen Hardwarefehler zu machen gehören (s. a. [5]),

- dass sämtlicher **Input vom Gerät überprüft** wird, um Fehler rechtzeitig zu erkennen und diese evtl. korrigieren zu können.
- Entdeckte Fehler sollten dazu dem **Betriebssystem gemeldet werden**, das seinerseits Maßnahmen ergreifen kann oder auftretende Fehler zumindest Erfassen und Auswerten kann.
- An kritische Stellen im Treibercode können **Timeouts** helfen, unendliches Warten auf eine Antwort der Hardware zu verhindern.
- Wenn ein Fehler erkannt wurde, sollte falls möglich eine **Wiederherstellung** des defekten Gerätes, z. B. durch Neustart, erfolgen.

Viele Lösungen um Treiberfehler zur Laufzeit zu erkennen und so eine Laufzeitfehlertoleranz von Gerätetreibern umzusetzen, verlassen sich vor allem auf Isolationsmechanismen mit Hardwareunterstützung wie Speicherschutz. In [40] z. B. werden Anwendungen vor Auswirkungen von Treiberfehlern bei einem Mikrokern-System geschützt, in dem eine Entkopplung der Treiber von den Anwendungen stattfindet. Bei Absturz eines Treibers kann dieser, durch transparente Wiederherstellungsmechanismen wie Microreboots, wieder in einen funktionstüchtigen Zustand gebracht werden.

Bei Nooks (siehe [39]) wird der Kernel vor Auswirkungen von Treiberfehlern geschützt, in dem kritische Treiber isoliert und in speziellen Schutzdomänen ausgeführt werden.

Da *eCos* wie bereits erwähnt über keine solchen Isolationsmechanismen verfügt, wurde hier ein anderer Ansatz gewählt und bei der Implementierung von Fehlertoleranzmechanismen im E/A-Subsystem angesetzt.

5 Absicherung der Treiberschnittstelle

Für die exemplarische Implementierung von Fehlertoleranzmechanismen wurde die allgemeine Treiberschnittstelle von *eCos* („*I/O Subsystem API*“ oder auch „*User API*“) ausgewählt.

Gründe dafür sind die einerseits in 4.3 vorgestellte Anfälligkeit von Betriebssystemen in Bezug auf Treiber, was besonders für eingebettete Betriebssysteme wie *eCos* gilt, die so gut wie keine Isolationsmechanismen oder Schutz gegen außer Kontrolle geratene Treiber bieten. Andererseits stellen Treiber einen Großteil des Betriebssystemquellcodes (vgl. 4.4(b)) dar und sind neben dem Kernel ein essenzieller Bestandteil. Gerade kleine eingebettete Systeme haben oft nur die Aufgabe Eingaben zu verarbeiten und in veränderter Form auszugeben, weswegen *eCos* bspw. so konfiguriert werden kann, dass es ohne Kernel auskommt und so wie eine Art Treiber-API genutzt werden kann.

Die Benutzerschnittstelle für Treiber in *eCos* wird allgemein von Applikationen verwendet, um über verschiedene Treiber mit Geräten zu kommunizieren. Da die Einbringung von Fehlertoleranzmechanismen, auch wenn sie hier auf ein bestimmtes Betriebssystem zugeschnitten ist, weitgehend generisch bleiben soll, d. h. auch evtl. nachträglich erstellte und hinzugefügte Treiber geschützt werden sollen, wird weitgehend auf Änderungen in den eigentlichen Treibern verzichtet.

Der folgende Abschnitt soll einen Überblick über den Aufbau und die Funktionsweise der Treiberschnittstelle geben, bevor in den anschließenden beiden Abschnitten auf die Implementierung der Fehlererkennung sowie der Fehlerbehandlung eingegangen wird.

5.1 Aufbau der Treiberschnittstelle

Das E/A-System in *eCos* kann in zwei grundlegende Einheiten aufgeteilt werden. Zum einen besteht es aus einer Menge von Treibern für bspw. Ethernet, Flash, Touchscreen, Tastatur, USB, Watchdog, Zeitgeber und serieller Schnittstelle, die nach Gerät sowie Zielarchitektur aufgeteilt sind. Dabei werden die Treiber über ein generisches Interface angesprochen, wobei die eigentliche Treiberimplementierung davon unabhängig ist. Zum anderen existiert die *User API*, die eine allgemeine Schnittstelle zu den unterschiedlichen Treibern darstellt, sowie zur Kommunikation mit selbst erstellten Treibern dient.

Hier soll auf die einzelnen Funktionen, die aus Sicht des Benutzers relevant sind, ebenso eingegangen werden wie auf den logischen Aufbau und Interaktion der Treiber untereinander, sowie einige technische Implementierungsdetails die für eine spätere Implementierung von Fehlertoleranzmechanismen in Abschnitt 5.2 von Interesse sind (vgl. [37], [38]).

5.1.1 Funktionen der Treiberschnittstelle

Wie bereits erwähnt, dient die Treiberschnittstelle als universeller Zugangspunkt für unterschiedliche Geräte. Dabei besitzt jedes Gerät einen eigenen Namen, mit dem sich dieses eindeutig identifizieren lässt.

Um die Funktionen der Benutzerschnittstelle verwenden zu können, wird ein Schlüssel („Handle“) vom Typ `cyg_io_handle` benötigt, welcher dem zu benutzendem Treiber zugeordnet ist. Dieses kann mithilfe der Funktion `cyg_io_lookup`, unter Angabe des Treibernamens, der stets mit „/dev/“ beginnt und bspw. für die serielle Schnittstelle „/dev/ser0“ lautet, aufgesucht und bereitgestellt werden.

Alle Funktionen der *User API* haben einen Rückgabewert vom Typ `Cyg_ErrNo`, der im Falle eines Fehlers, wie beispielsweise beim Nachschlagen eines nicht vorhandenen Treibernamens, negativ ist und einem Fehlercode aus `<cyg/error/codes.h>` entspricht. Folgende Funktionen werden durch die *User API* definiert (vgl. [38]):

- `Cyg_ErrNo cyg_io_lookup(const char *name, cyg_io_handle_t *handle)`: Diese Funktion überprüft, ob ein Gerätetreiber mit dem entsprechenden Namen existiert und gibt im positiven Fall ein Handle für den Eintrag zurück.
- `Cyg_ErrNo cyg_io_write(cyg_io_handle_t handle, const void *buf, cyg_uint32 *len)`: Schreibt Daten aus `buf` auf ein Gerät. Die Länge der zu schreibenden Daten wird durch `len` bestimmt, in der anschließend auch die Länge der tatsächlich geschriebenen Daten gespeichert wird.
- `Cyg_ErrNo cyg_io_read(cyg_io_handle_t handle, void *buf, cyg_uint32 *len)`: Liest Daten von einem Gerät, dass durch das entsprechende Handle identifiziert wird. `len` gibt hierbei die gewünschte Länge zu lesender Daten an, die sich von der tatsächlichen Länge gelesener Daten unterscheiden kann, die im Anschluss ebenfalls in `len` gespeichert wird.
- `Cyg_ErrNo cyg_io_bwrite(cyg_io_handle_t handle, const void *buf, cyg_uint32 *len, cyg_uint32 pos)`: Wie `cyg_io_write` aber für blockorientierte Geräte, wobei sich hier `pos` und `len` auf Blöcke beziehen und nicht auf Bytes.
- `Cyg_ErrNo cyg_io_bread(cyg_io_handle_t handle, void *buf, cyg_uint32 *len, cyg_uint32 pos )`: Wie `cyg_io_read` aber für Blockgeräte, wobei sich `pos` und `len` hier ebenfalls auf Blöcke beziehen.
- `Cyg_ErrNo cyg_io_get_config(cyg_io_handle_t handle, cyg_uint32 key, void *buf, cyg_uint32 *len)`: Mit dieser Funktion können unterschiedliche Konfigurationsdaten eines Gerätes abgefragt werden. Welche Daten genau in `buf` gespeichert werden, wird durch den Parameter `key` bestimmt. Der Wert für Parameter `len` muss dabei der Größe der abgefragten Daten entsprechen.
- `Cyg_ErrNo cyg_io_set_config(cyg_io_handle_t handle, cyg_uint32 key, const void *buf, cyg_uint32 *len)`: Mit Hilfe dieser Funktion kann die Konfiguration für ein bestimmtes Gerät geändert werden. Die zu ändernden Konfigurationsdaten, die durch `key`

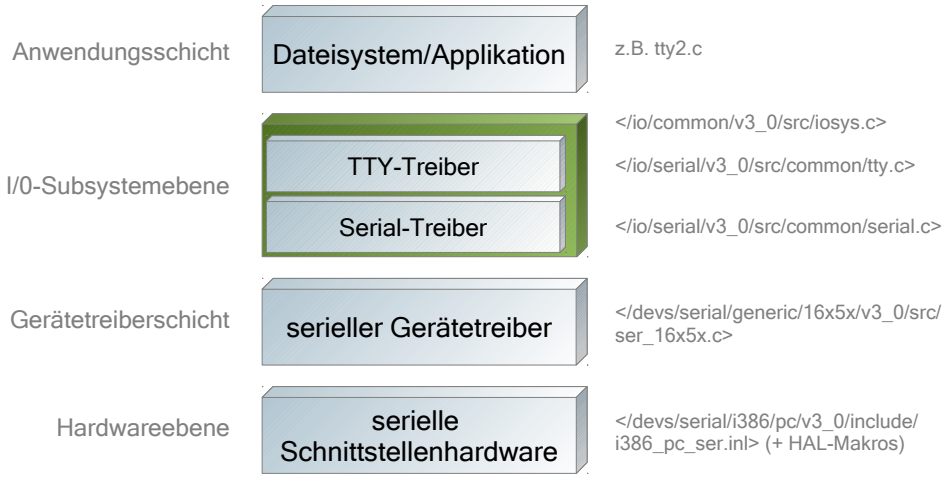


Abbildung 5.1: Aufbau der Schichtenarchitektur des Treibersystems in Anlehnung an [37], am Beispiel der TTY- und Serial-Treiberintegration mit zugehöriger *eCos*-Paketorganisation (rechts). Der geplante durch Fehlertoleranzmechanismen abzusichernde Bereich, der die Funktionen der *User API* betrifft, ist grün makiert.

bestimmt werden, sind für jeden Treiber unterschiedlich und können in `<cyg/io/config_keys.h>` nachgeschlagen werden.

- `cyg_bool cyg_io_select(cyg_io_handle_t handle, cyg_uint32 which, CYG_ADDRWORD info)`: Prüft ob Daten am Gerät bereitstehen oder das Gerät bereit zum Schreiben ist, je nachdem welchen Wert der Parameter `which` enthält.

Alle hier aufgeführten Funktionen kommunizieren je nach Handle mit den, im nächsten Abschnitt beschriebenen, treiberspezifischen Funktionen. Alle Treiber sowie die Schnittstelle selbst sind in *C* geschrieben, was den Einsatz von *AspectC++* erschwert.

5.1.2 Schichtenarchitektur für Treiber

Die besondere Architektur der Treiber in *eCos* unterstützt einen schichtweisen Aufbau. Dies erlaubt Treiber nach Grund- und Spezialfunktionalität aufzuteilen, sowie vorhandene Gerätetreiber zu erweitern. In Abbildung 5.1 ist der Aufbau der Schichten des Treibersystems, am Beispiel des TTY-Treibers („terminal like“), sowie des in der Schicht darunter liegenden Treibers für die serielle Schnittstelle, zu sehen. Dabei kennt jeder Treiber nur den Treiber der nächstunteren Schicht. Neben den oben beschriebenen Funktionen besitzt jeder Treiber eine `cyg_io_init`-Funktion, die bei der Treiberinitialisierung beim Start des Betriebssystems aufgerufen wird.

Während im Schema aus Abbildung 5.1 ein Programm der Anwendungsschicht über die *User API* mit dem TTY-Treiber kommuniziert, gibt dieser wiederum Eingaben und Ausgaben ebenfalls über die Treiberschnittstelle an den untergeordneten, seriellen

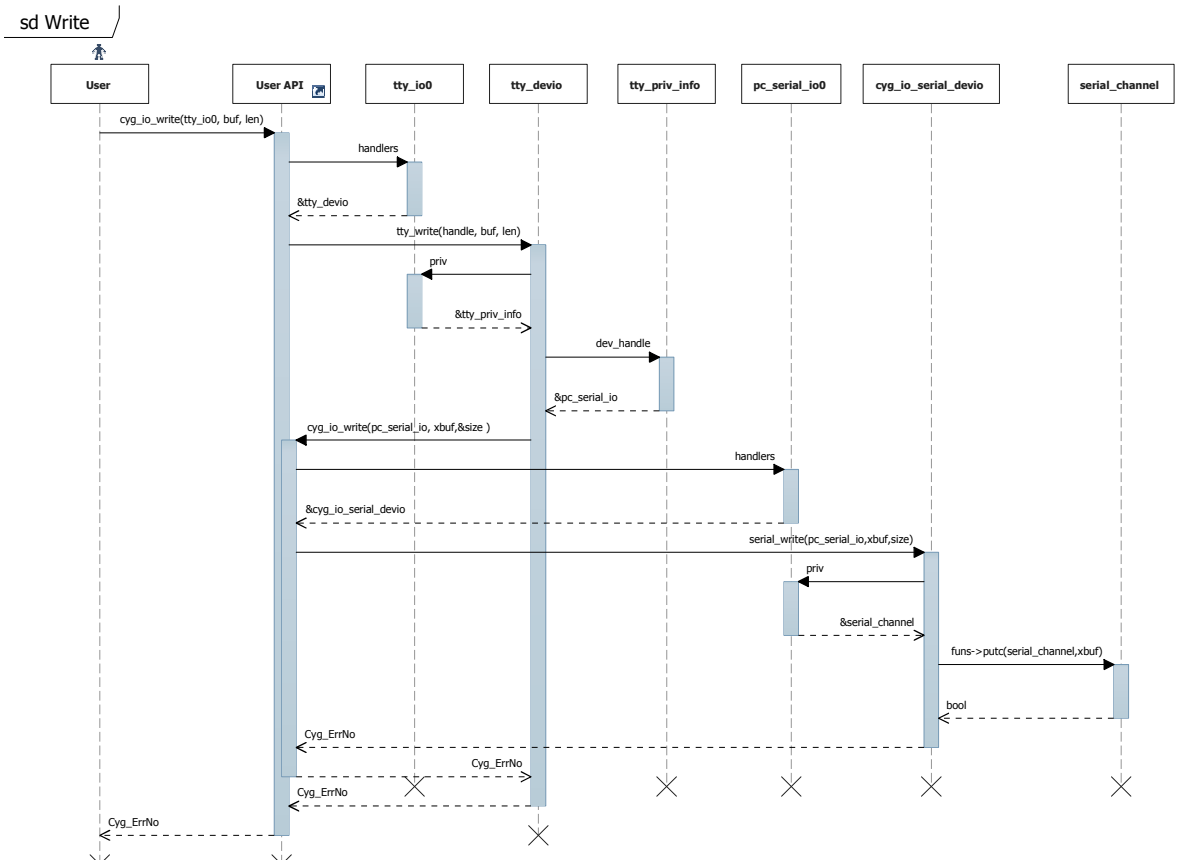


Abbildung 5.2: Sequenzdiagramm eines `cyg_io_write`-Aufrufs mit TTY- und Serialtreiber.

Treiber weiter. Der Terminal-Treiber bietet der Applikation dabei zusätzliche Funktionen, wie Bearbeitungsfunktionen und Zeilenpufferung, gegenüber dem einfachen seriellen Treiber.

Der serielle Treiber hingegen benutzt Funktionen des seriellen Gerätetreibers, der spezifisch auf ein Hardwaregerät festgelegt ist, sowie prozessorspezifische Funktionen der seriellen Schnittstellenhardware die über die HAL[1] von *eCos* angesprochen werden.

Während die eigentliche Treiberimplementierung also je nach System und dessen Hardwarekonfiguration unterschiedlich ist, bleibt die Kommunikation der Treiber mit der Anwendungsschicht sowie der Treiber untereinander durch die Treiberschnittstelle, in Abbildung 5.1 der grün markierte Bereich, stets identisch. Auch die Initialisierung der Treiber sowie die Konfiguration können über die Schichten hinweg durch die *User API* erfolgen.

Im Sequenzdiagram aus Abbildung 5.2 ist ein Beispielaufruf einer `cyg_io_write`-Funktion zu sehen. In diesem Beispiel sollen Daten formatiert und über die serielle Schnittstelle ausgegeben werden.

[1]Hardware Abstraction Layer - Hardwareabstraktionschicht von *eCos* zur Erleichterung der Portierbarkeit

Innerhalb der Treiberschnittstellenfunktion wird in einem ersten Schritt ein spezielles Objekt vom Typ `cyg_devio_table`, welches die E/A-Funktionen des vom Handle bestimmten Treibers beinhaltet, abgefragt. Mit Hilfe dieser E/A-Funktionstabelle wird schließlich die eigentliche `tty_write` Funktion des Terminaltreibers aufgerufen. Da der Terminaltreiber auf dem Treiber der seriellen Schnittstelle aufsetzt, folgt aus diesem heraus ein weiterer Aufruf von `cyg_io_write` über die *User API*, mit dem Handle der seriellen Schnittstelle. Der Vorgang aus dem ersten Aufruf innerhalb der Treiberschnittstelle wiederholt sich also, nur dass der serielle Treiber die unterste Treiberschicht darstellt und schließlich mit Hilfe von HAL-Funktionen eine Ausgabe über die Hardware erfolgt. Bei anschließendem Erfolg wird ein entsprechender Rückgabewert durch alle Schichten zurück zum Ursprung des ersten Aufrufs weitergegeben.

Pro Funktionsaufruf der *User API* durch die Anwendungsschicht können also abhängig von der Anzahl der Schichten, mehrere weitere Zugriffe auf die Treiberschnittstelle erfolgen, was die Bedeutung einer Absicherung an dieser Stelle erhöht.

5.1.3 Gerätetreiber- und Gerätefunktionseinträge im Speicher

Wie bereits im vorigen Abschnitt zu sehen, sind Funktionen und Eigenschaften der Treiber in speziellen Datenobjekten organisiert. Bei dem Handle, dass zum Ansprechen der Treiber über die *User API* als Parameter übergeben wird, handelt es sich lediglich um einen Zeiger auf einen Gerätetreibereintrag bzw. `cyg_devtab_entry`-Objekt, das Namen, Statusinformationen sowie weitere Zeiger auf Funktionen enthält.

Die wichtigsten Funktionen, die die Methoden der Treiberschnittstelle widerspiegeln, sind in einem gesonderten Objekt, dem `cyg_devio_table` organisiert. Diese Funktionstabelle ist ebenfalls über Zeiger aus dem Gerätetreibereintrag erreichbar und enthält wiederum Zeiger auf Funktionen.

In Abbildung 5.3 ist eine Übersicht der beiden Objekte mitsamt ihrer Attribute zu sehen. Hierbei erkennt man, dass alle Attribute des Gerätetreibereintrags sowie der Funktionstabelle `cyg_devio_table`, bis auf `status`, Zeiger auf Daten bzw. Funktionszeiger sind. Gerade diese exzessive Zeigerverwendung innerhalb der Treiberschnittstelle macht diese besonders anfällig für Fehler wie bspw. Bitkipper. Aus diesem Grund werden die im nächsten Abschnitt besprochenen Fehlertoleranzmechanismen an diesen Stellen ansetzen.

Während die Funktionstabellen der Treiber irgendwo in der Datensektion im Speicher angelegt werden, wobei diese stets vor den Gerätetreibereinträgen liegen, werden die Gerätetreibereinträge in einem ausgezeichneten Bereich der Datensektion angelegt, der durch die Label `__DEVTAB__[0]` bis `__DEVTAB_END__` begrenzt ist. Dies dient vor allem der `cyg_io_lookup`-Funktion, um den jeweiligen gesuchten Eintrag nach seinem Namen zu finden, in dem alle Gerätetreibereinträge sukzessiv durchlaufen werden, hilft aber auch den validen Bereich für eine spätere Plausibilitätsprüfung einzuschränken. Zusätzlich sind sämtliche Einträge von `cyg_devtab_entry` im Speicher ausgerichtet („Data Alignment“) und besitzen Speicheradressen, die ein Vielfaches, des von *eCos* festgelegten Wertes, `CYGARC_ALIGNMENT` darstellen, was ebenfalls für die Bereichsprüfung ausgenutzt werden kann.

Abbildung 5.3: Klassendiagramm der Gerätetreibereinträge `cyg_devtab_entry` und `cyg_devio_table` und ihre Verwendung in der *User API*.

5.2 Fehlererkennung in der Treiberschnittstelle

Hier soll die im vorigen Abschnitt beschriebene Treiberschnittstelle um Mechanismen zur Fehlererkennung ergänzt werden. Um fehlerhafte Objekt- oder Funktionsadressen, die etwa durch Bit-Flips im Speicher korrumpiert wurden zu erkennen, sind mehrere Lösungsansätze bekannt (vgl. 3.2.1.1).

Die hier verwendeten Verfahren beruhen auf dem Prinzip der Plausibilitätsprüfung, da diese Technik meist sehr effizient implementiert werden kann und zudem eine hohe Portabilität aufweist, d. h. in ähnlicher Form auch außerhalb dieser exemplarischen Implementierung in *eCos* verwendet werden kann, da keine Voraussetzungen an die Hardware oder das Betriebssystem gestellt werden.

Die Untersuchung der *User API* in Abschnitt 7.2.1 zeigt, dass der Großteil (bei allen verwendeten Testprogrammen mehr als 80%) der Fehler die durch injizierte Bit-Flips auftraten, durch ungültige Befehlszählerzustände (*IP*) und Speicherzugriffsverletzungen („*memory error*“) verursacht wurden. Die Mehrheit dieser Fehler lässt sich durch Bereichsprüfungen („*range check*“) einfach und zuverlässig erkennen. Eine Auflistung der für die Bereichsprüfung verwendeten, durch Label markierten Bereichsgrenzen und den zugehörigen Lokalitäten im Speicher ist in Tabelle 5.1 zu finden.

Im Folgenden werden einige der später evaluierten Methoden und Implementierungen zur Fehlerdetektion in der *User API* vorgestellt.

Label	Segment	Adresslokalität
`_DEVTAB_START`	data	Anfangsadresse der im Speicher hintereinander liegenden Einträge vom Typ `cyg_devtab_entry_t`. Äquivalent mit dem durch *eCos* bereitgestellten und bei der Initialisierung des I/O-Packets definierten Labels `__DEVTAB__`.
`_DEVTAB_END`	data	Markiert das Ende der `cyg_devtab_entry_t` Einträge im Speicher und entspricht dem von *eCos* definiertem Label `__DEVTAB_END__`.
`_DEVIO_START`	data	Früheste Speicheradresse bei der ein `cyg_devio_table_t` Eintrag erwartet wird. Wird hier auf das beim Memorylayout von *eCos* definiertem Label `__ram_data_start` gesetzt, welches den Anfang des Datensegments markiert.
`_DEVIO_END`	data	Untere Grenze von `cyg_devio_table_t` Einträgen. Entspricht dem Label `_DEVTAB_START`, da alle Instanzen von `cyg_devio_table_t` vor ihrem Eintrag in die Treibertabelle definiert werden (siehe Abschnitt 5.1.3).
`_FUNCTION_START`	text	Erwartete obere Adressgrenze für Funktionen im Codesegment. Für die Treiberfunktionen ist eine Einschränkung auf die Funktionsadresse von `cyg_start` ausreichend (siehe Abschnitt 5.1.3).
`_FUNCTION_END`	text	Untere Grenze für Funktionsadressen. Wurde für die Treiberfunktionen auf die Funktionsadresse von `cyg_io_init` beschränkt.
`_BSS_START`	bss	Segmentsanfangsadresse des BSS-Abschnitts, welche dem in *eCos* definiertem Label `__bss_start` entspricht.
`_BSS_END`	bss	Im Memorylayout von *eCos* als `__bss_end` definiertes Label, dass die Endadresse des BSS-Segments markiert.
`_INFO_START`	data	Datensegment-Startadresse `__ram_data_start` stellt die obere Grenze der treiberspezifischen Daten dar.
`_INFO_END`	bss	Da private Daten der Treiber sowohl in der Datensektion als auch der BSS-Sektion auftauchen können, ist deren untere Grenze dementsprechend zu berücksichtigen, entspricht also dem *eCos* Label `__bss_end`.

Tabelle 5.1: Verwendete Label zur Festlegung der Grenzen für die Bereichsprüfung zur Erkennung korrumpierter Zeigeradressen.

5.2.1 Dereferenzierungsfunktion

Ein notwendiger Zeitpunkt um die Gültigkeit einer Referenz bzw. einer Zeigeradresse auf eine Datenstruktur zu prüfen ist vor der Dereferenzierung, also bevor auf der referenzierten Datenstruktur Operationen ausgeführt werden. Da in der *User API* Zeiger auf Einträge vom Typ `cyg_devtab_entry_t` in Form des Handle als Übergabeparameter in jeder Funktion benutzt werden, wobei `cyg_devtab_entry_t` wiederum weitere Zeiger auf Methoden, Variablen und ein `cyg_devio_table_t`-Objekt besitzt, das seinerseits aus Funktionszeigern besteht, scheint eine Plausibilitätsprüfung jeweils vor der Dereferenzierung dieser Zeiger vielversprechend in Hinblick auf den Anteil detektierter Fehler.

```
1  inline cyg_devtab_entry_t& deref(cyg_devtab_entry_t* devtab) {
2    return *devtab;
3  }
4
5  inline cyg_devio_table_t& deref(cyg_devio_table_t* devio) {
6    return *devio;
7  }
```

Listing 5.1: Implementierung der Dereferenzierungsfunktion für `cyg_devtab_entry_t` und `cyg_devio_table_t`.

Da in der aktuellen *AspectC++*-Version allgemeine Dereferenzierungsoperationen keine Joinpoints besitzen, sind Änderungen am gegebenen Quellcode von *eCos* nötig. Die einfachste Methode Aspekten den Zugriff auf die Dereferenzierungsoperationen zu ermöglichen ist durch eine Alias-Methode, die die Dereferenzierung übernimmt. Wie in Listing 5.1 zu sehen, sind jeweils getrennte Dereferenzierungsfunktionen für die einzelnen Datenstrukturen angelegt worden, da einerseits verschiedene Überprüfungskriterien für jede Datenstruktur benutzt werden, andererseits die zurückgegebenen Referenzen in *C++* typisiert sein müssen.

```
1  Cyg_ErrNo cyg_io_write(cyg_io_handle_t handle, const void *buf, cyg_uint32 *len) {
2      cyg_devtab_entry_t* t = (cyg_devtab_entry_t *)handle;
3      if (!deref(deref(t).handlers).write) {
4          return -EDEVNOSUPP;
5      }
6      ... // restlicher Funktionsrumpf
7      return deref(deref(t).handlers).write(handle, buf, len);
8  }
```

Listing 5.2: Einbettung der Dereferenzierungsfunktion in den vorhandenen Quellcode der Gerätetreiberschnittstelle.

```
1  aspect PointerCheck {
2    pointcut derefCheck() = call("% deref(...)");
3
4    advice derefCheck() && args(devtab_entry) : around(cyg_devtab_entry_t* devtab_entry) {
5      if (((void*)devtab_entry < (void*)_DEVTAB_START)
6          || ((void*)devtab_entry >= (void*)_DEVTAB_END)
7          || ((unsigned long)devtab_entry % CYGARC_ALIGNMENT != 0)) {
8        errors_detected++;
9      }
10     *tjp->result() = devtab_entry;
11   }
```

```
12
13    advice derefCheck() && args(devio_tab) : around(cyg_devio_table_t* devio_tab) {
14      if (((void*)devio_tab < (void*)_DEVIO_START)
15          || ((void*)devio_tab >= (void*)_DEVIO_END) ) {
16          || (!cyclicFunctionCheck(devio_tab))) {
17        errors_detected++;
18      }
19      *tjp->result() = devio_tab;
20    }
21  };
```

Listing 5.3: Implementierung des Aspekts zur Überprüfung gültiger Speicheraddressen mit Hilfe der Dereferenzierungsfunktion.

In Listing 5.3 ist der zugehörige Aspekt mit Advices, die die Plausibilitätsprüfungen der jeweiligen Datenstrukturen implementieren, zu sehen. Die abgebildete Version führt ledigliche eine Fehlererkennung durch, in dem eine globale Variable hochgezählt wird (Zeile 10 und 17), kann aber später durch eine entsprechende Fehlerbehandlungsroutine (siehe Abschnitt 5.3) ersetzt werden, die bspw. die zurückgegebene Zeigeradresse manipuliert, wie in Zeile 10 und 19 bereits angedeutet.

Neben der Prüfung, ob sich eine Zeigeradresse zwischen den vorgesehenen Speicherbereichsgrenzen (siehe Tabelle 5.1) befindet, ist zudem eine Überprüfung der Speicherausrichtung (*„Data Alignment“*)) für `cyg_devtab_entry_t` (Zeile 7) und eine zyklische Funktionsprüfung für `cyg_devio_table_t` (Zeile 16) erstellt worden.

```
1  static bool cyclicFunctionCheck(cyg_devio_table_t* devio_tab) {
2    char** tablePtr = (char**) devio_tab;
3
4    for (unsigned i=0; i<DEVIO_FUNCTIONS; i++) {
5      void* functPtr = (void*)*tablePtr;
6      if ((functPtr < (void*)_FUNCTION_START) || (functPtr >= (void*)_FUNCTION_END)) return
           false;
7      tablePtr++;
8    }
9    return true;
10 }
```

Listing 5.4: Zyklische Prüfung der Funktionszeiger für eine gegebene Instanz von `cyg_devio_table_t`.

Eine zyklische Funktionsprüfung, die in Listing 5.4 dargestellt ist, wird zur Prüfung der Validität der Einträge einer `cyg_devio_table_t` Instanz benötigt, da die Dereferenzierungsfunktion das Ziel im Falle eines folgenden Funktionsaufrufes nicht feststellen kann. Der Nachteil dieser Prüfung liegt, neben dem eher kleinen Overhead gegenüber einer direkten Prüfung eines Funktionsaufrufes darin, dass zum Teil Fehler in nicht verwendetem Code gefunden werden und so evtl. eine mögliche Fehlerbehebungsroutine unnötigerweise ausgeführt oder eine Ausnahmebehandlung durchgeführt wird.

Ein allgemeiner Nachteil der Dereferenzierungsfunktion besteht in der begrenzten Anwendbarkeit. Ihr Einsatz beschränkt sich lediglich auf die die Datenstrukturen referenzierenden Zeigeradressen, während eine gezielte Überprüfung beim Zugriff auf einzelne Datenfelder nicht ohne Weiteres möglich ist. Zudem ist der durch den Einsatz der Dere-

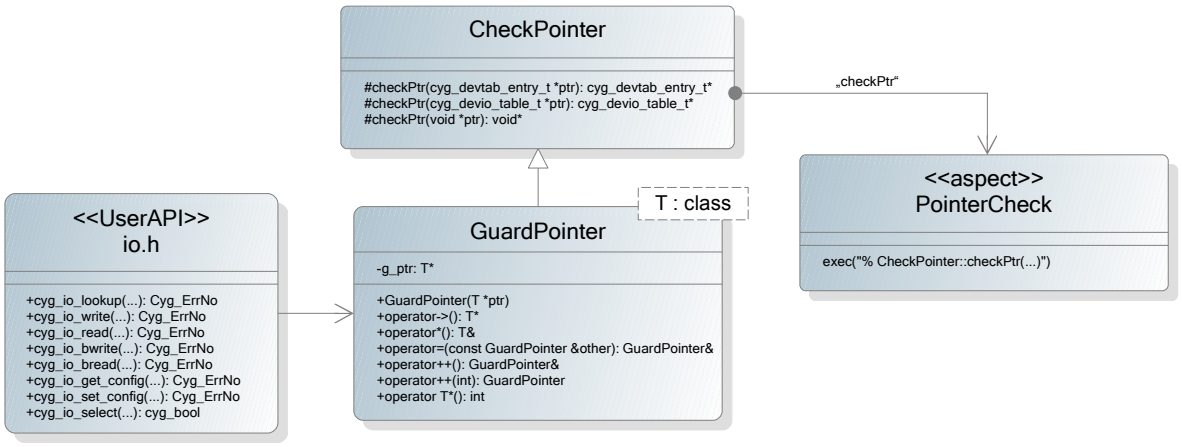


Abbildung 5.4: Klassendiagramm des *GuardPointers*, seiner Einbettung in das Umfeld der Treiberschnittstelle sowie die Verknüpfung zum zugehörigen Aspekt für die Zeigeradressenüberprüfung.

ferenzierungsfunktion modifizierter Quelltext intuitiv schlechter zu lesen, wie Listing 5.2 verdeutlicht.

5.2.2 GuardPointer

Die in *C++* als sog. *Smartpointer* (vgl. [41],[42]) bezeichnete abstrakte Datenstruktur, die sich ähnlich wie ein gewöhnlicher Zeiger verhält, dessen Funktionalität aber etwa um automatisches Löschen und Freigeben von Speicher (*„garbage collection“*) erweiterbar ist, soll hier in ähnlicher Form für die Fehlererkennung verwendet werden.

Die *GuardPointer* sollen innerhalb der *User API* als Wrapper für Zeiger auf kritische Strukturen bzw. Funktionen eingesetzt werden und fehlerhafte Zeigeradressen erkennen, bevor auf das referenzierte Objekt zugegriffen wird. Im Prinzip stellen sie eine erweiterte bzw. elegantere Lösung der Dereferenzierungsfunktion dar, da sie sich in ihrer Funktionsweise lediglich in ihrer Integration in den Quelltext unterscheiden.

Der Vorteil des *GuardPointers* liegt darin, dass er fast wie ein gewöhnlicher Zeiger verwendet werden kann und so nur minimale Änderungen am Quellcode notwendig sind, wodurch die Lesbarkeit erhalten bleibt. Zudem entsteht auch hier durch die Verwendung dieser speziellen Zeiger im nicht verwendeten Zustand, d. h. bei deaktiviertem Aspekt, wenn keine Überprüfung der Zeigeradresse stattfindet, kein zusätzlicher Overhead[2] (vgl. Abschnitt 7.2.3).

```
1  template <class T>
2  class GuardPointer : public CheckPointer {
3    T *g_ptr;   // der eigentliche Pointer
4  public:
5    GuardPointer(T *ptr = 0) : g_ptr(ptr){}
6
```

[2] g++ 4.4.5 mit Optimierungsoption -O2

```
7     T* operator->() { g_ptr=(T*)checkPtr(g_ptr); return g_ptr;}
8     T& operator*() { g_ptr=(T*)checkPtr(g_ptr); return *g_ptr; }
9
10    GuardPointer& operator=(const GuardPointer &other){ g_ptr = other.g_ptr; return *this; }
11    GuardPointer& operator++() {g_ptr++; return *this;}
12    GuardPointer operator++(int) {GuardPointer tmp(*this); operator++(); return tmp;}
13
14    operator T*() { return g_ptr; }
15  };
```

Listing 5.5: Implementierung des GuardPointers als Vorbereitung der Plausibilitätsprüfung der Zeigeradresse vor Dereferenzierungsoperationen.

In Listing 5.5 ist eine Implementierung eines allgemeinen *GuardPointers* zu sehen. Neben dem Überladen der Dereferenzierungsfunktionen sind weitere Operatoren modifiziert worden, um einem gewöhnlichen Zeiger zu imitieren (10-Zeile 14). Bei jeder Dereferenzierung wird nun die Methode `checkPointer(...)` aufgerufen die von der Klasse `CheckPointer` geerbt wurde.

```
1  class CheckPointer {
2  protected:
3    cyg_devtab_entry_t* checkPtr(cyg_devtab_entry_t *ptr) { return ptr; };
4    cyg_devio_table_t* checkPtr(cyg_devio_table_t *ptr) { return ptr; };
5    void* checkPtr(void *ptr) { return ptr; };
6  };
```

Listing 5.6: Die Klasse CheckPointer, deren Defaultfunktionen später durch Aspekte mit Plausibilitätsprüfungen ersetzt werden sollen.

In Listing 5.6 ist diese Klasse mit der prototypischen `checkPtr`-Funktion zu sehen, die dem späteren Aspekt als Joinpoint dient, um diese beim spezifischen *GuardPointer* durch die eigentliche Plausibilitätsprüfung zu ersetzen. Für die Objekte `cyg_devtab_entry_t` und `cyg_devio_table_t` wurden hier jeweils eigene Methoden zur Prüfung implementiert um diese später gesondert zu betrachten, sowie eine allgemeine Überprüfungsmethode vom Typ `void`. Der Grund dafür, dass hier nicht einfach mit generischen Typen gearbeitet wurde, liegt darin, dass in der benutzten *AspectC++*-Version das sog. Template Matching noch nicht vollständig implementiert war.

```
1  Cyg_ErrNo cyg_io_write(cyg_io_handle_t handle, const void *buf, cyg_uint32 *len) {
2      GuardPointer<cyg_devtab_entry_t> t = (cyg_devtab_entry_t *)handle;
3      GuardPointer<cyg_devio_table_t> io = t->handlers;
4      if (!io->write) {
5          return -EDEVNOSUPP;
6      }
7      ... // restlicher Funktionsrumpf
8      return io->write(handle, buf, len);
9  }
```

Listing 5.7: Exemplarische Verwendung der *GuardPointer* innerhalb der `cyg_io_write`-Funktion der *User API*.

Die Realisierung des verwendeten Aspekts, um die Fehlerdetektion für die *GuardPointer* umzusetzen, entspricht genau dem in Abschnitt 5.2.1 verwendeten Aspekt für die Dereferenzierungsfunktion, da sie auf demselben Prinzip beruhen. Der einzige bereits angesprochene Vorteil liegt in ihrer einfacheren Usability.

5.2.3 Akzessoren

Eine weitere vielversprechende Stelle Joinpoints explizit in den Code einzubringen, um dort Plausibilitätsprüfungen durchzuführen, ist bevor auf einen Member der beiden abstrakten Datenstrukturen zugegriffen wird. Hierzu wurde, wie in Listing 5.8 exemplarisch zu sehen, die Datenstrukturen `cyg_devtab_entry_t` und `cyg_devio_table_t` um Getter-Methoden für Datenfelder und Alias-Funktionsaufrufe für Funktionszeiger ergänzt.

```
1  typedef struct cyg_devtab_entry_t {
2    // urspruengliche Variablen und Funktionszeiger
3    const char        *name;
4    Cyg_ErrNo        (*lookup)(struct cyg_devtab_entry **tab, struct cyg_devtab_entry *
         sub_tab, const char *name);
5    ...   // weitere Variablen und Funktioszeiger
6    // getter und alias-Funktionsaufrufe
7    void* getPriv() { return priv; };
8    Cyg_ErrNo lookup_safe(cyg_devtab_entry **tab, cyg_devtab_entry *sub_tab, const char *
         name) { return lookup(tab, sub_tab, name); };
9  };
```

Listing 5.8: Exemplarische Abfragefunktion und Alias-Funktionsaufruf zum Einbringen von Joinpoints.

Auch der Quellcode der *User API* musste entsprechend der neuen Akzessoren modifiziert werden, damit diese auch benutzt werden. Listing 5.9 zeigt diese nötigen Quellcodeänderungen exemplarisch am Beispiel der `cyg_io_write`-Funktion. Zum einen wurde der direkte Zugriff auf die Funktions-Handler durch eine getter-Funktion in Zeile 3 ersetzt, zum anderen wurde der Aufruf der write-Funktion durch einen eigenen Funktionsaufruf in Zeile 9 ersetzt.

```
1  Cyg_ErrNo cyg_io_write(cyg_io_handle_t handle, const void *buf, cyg_uint32 *len) {
2      cyg_devtab_entry_t *t = (cyg_devtab_entry_t *)handle;
3      cyg_devio_table_t *io = t->getHandlers();
4      // Validate request
5      if (!io->write) {
6          return -EDEVNOSUPP;
7      }
8      ... // restlicher Funktionsrumpf
9      return io->write_safe(handle, buf, len);
10 }
```

Listing 5.9: Verwendung der Zugriffsfunktionen innerhalb der Methode `cyg_io_write` der *User API*.

Durch die nun vorhandenen Joinpoints können bei aktiviertem Aspekt, wie in Listing 5.10 zu sehen, verschiedene Validierungen durchgeführt werden:

- Für die Einträge von `name` und `dep_name` in `cyg_devtab_entry_t` wird eine Überprüfung des Treibernames durchgeführt (Zeile 16), die feststellt, ob der Name mit dem festgelegten Präfix „/dev/“ beginnt und ob der Treibername nur aus gültigen Zeichen besteht, wobei hier alle Zeichen zwischen ’/’ und ’z’ als gültig erachtet wurden.

- Der Eintrag `handlers` in `cyg_devtab_entry_t`, der einen Zeiger auf ein `cyg_devio_table_t` Objekt hält, wird auf Validität überprüft (Zeile 25) in dem geschaut wird, ob sich

die Zeigeradresse innerhalb der vorgesehenen, erwarteten Grenzen befindet (vgl. Tabelle 5.1).

- Für die treiberspezifischen Daten, die durch die Variable `priv` in `cyg_devtab_entry_t` referenziert werden, wird eine eigene Bereichsprüfung mit spezifischen Grenzen durchgeführt (Zeile 42).
- Für die Funktionszeiger `lookup` und `init` aus `cyg_devtab_entry_t` sowie alle Funktionszeiger aus `cyg_devio_table_t` wird geprüft, ob sich diese innerhalb des erwarteten Bereichs im Codesegment befinden (Zeile 32 ff.).

```
1  aspect GetterCheck {
2    pointcut nameCheck() =
3      call("% cyg_devtab_entry::getName()") || call("% cyg_devtab_entry::getDepName()");
4    pointcut handlerCheck() =
5      call("% cyg_devtab_entry::getHandlers()");
6
7    pointcut lookup_functionCheck() =
8      call("% cyg_devtab_entry::lookup_safe(...)");
9    pointcut write_functionCheck() =
10     call("% cyg_devio_table::write_safe(...)");
11   ... // weitere gleichartige Pointcuts fuer read, bwrite, bread, select, get_config und
         set_config
12
13   pointcut getPriv_functionCheck() =
14     call("% cyg_devtab_entry::getPriv()");
15
16   advice nameCheck() : after() {
17     const char* name = *tjp->result();
18     const char* dev_name;
19     if (!startsWith(name, "/dev/", &dev_name)
20         || !containsValidChars(dev_name)) {
21       errors_detected++;
22     }
23   }
24
25   advice handlerCheck() : after() {
26     void* handler = *tjp->result();
27     if (handler < (void*)_DEVIO_START || handler >= (void*)_DEVIO_END) {
28       errors_detected++;
29     }
30   }
31
32   advice lookup_functionCheck() : before() {
33     checkFunction((void*)((cyg_devtab_entry_t*)tjp->target())->lookup);
34   }
35
36   advice write_functionCheck() : before() {
37     checkFunction((void*)((cyg_devio_table_t*)tjp->target())->write);
38   }
39
40   ... // weitere Advices fuer read, bwrite, bread, select, get_config und set_config nach
         dem selben Muster
41
42     advice getPriv_functionCheck() : after() {
```

```
43      void* priv = *tjp->result();
44      if (priv < (void*)_INFO_START || priv >= (void*)_INFO_END) {
45        errors_detected++;
46      }
47    }
48  };
```

Listing 5.10: Aspekt zur Fehlererkennung mit Hilfe von Bereichsprüfungen vor dem Memberzugriff durch die implementierten Akzessoren der abstrakten Datentypen.

Der Vorteil der Akzessoren ist einerseits die Erhaltung der Lesbarkeit des Quellcodes ähnlich den *GuardPointern*, andererseits die Flexibilität der abgefragten Daten, d. h. die Menge eingebrachter Joinpoints gegenüber den vorigen Mechanismen, die sich auf Zeiger bzw. die Dereferenzierung beschränken. Ein Nachteil sind die größeren Modifikationen am Quellcode, die bspw. gegenüber den *GuardPointern* nötig sind, da neben den Änderungen innerhalb der Treiberschnittstellenmethoden auch direkte Änderungen an den beiden Datenstrukturen `cyg_devtab_entry_t` und `cyg_devio_table_t`, etwa durch Ergänzung der Alias-Methoden, durchgeführt werden mussten.

5.2.4 Kombinierte Lösungen

Neben den oben vorgestellten Fehlererkennungsmechanismen wurden auch Kombinationen dieser getestet (siehe Abschnitt 7.2.6). Da die *GuardPointer* der Dereferenzierungsfunktion entsprechen, wurden lediglich Kombinationen aus verschiedenen Varianten von Akzessoren und *GuardPointern* implementiert.

Neben einigen Überschneidungen, wie der zyklischen Funktionszeigerprüfung (vgl. Listing 5.4), die durch die besser geeigneten Zugriffsfunktionen (vgl. Listing 5.8) ersetzt wurde und der `getHandlers`-Funktion der Akzessoren, die durch den entsprechenden *GuardPointer* ersetzt wurde, ergänzen sich die beiden Verfahren gut. Durch Kombination der beiden Lösungen wurden die höchsten Fehlererkennungsraten ermittelt.

5.3 Fehlerbehandlung in der Treiberschnittstelle

Die bisher in Abschnitt 5.2 implementierte Fehlererkennung der Treiberschnittstelle soll hier um eine Fehlerkorrektur extendiert werden.

5.3.1 Anforderungen an die Fehlerbehebung

Die Anforderungen, die an die Wiederherstellungsfunktion gestellt werden, sind unter anderem:

- Vorgesehen ist eine möglichst einfache und kostengünstige Implementierung bzgl. Codegröße und Speicherbedarf.

- Die Wiederherstellungsroutine muss mit eingeschränktem Informationsbedarf auskommen, da innerhalb der Fehler erkennenden Aspekte, bis auf den Typ der Datenstruktur, nur wenig Informationen über Lokalität und Herkunft des Fehlers bekannt sind. Ebenfalls eingeschränkt ist der Handlungsspielraum, da im bisherigen Lösungskonzept im Prinzip nur die zurückgegebene Zeigeradresse manipuliert werden kann (siehe Listing 5.10 und Listing 5.3).

- Ein kompletter Neustart des Betriebssystems sollte möglichst vermieden werden, um Applikationsabstürze und evtl. Datenverluste zu verhindern.

- Ein Schutzmechanismus gegen Fehler in der Fehlerbehebungsroutine selbst sollte vorhanden sein.

5.3.2 Implementierung der Fehlerbehebung

Um den Anforderungen an die Fehlerbehebungsfunktion gerecht zu werden, wurde ein spezielles „Fehlergerät“ eingeführt, das beim Aufruf sämtlicher Funktionen einen definierten Fehlercode zurückliefert, um bspw. einer Anwendung, welche die Treiberschnittstelle benutzt die Möglichkeit bietet, den Fehler abzufangen und entsprechend zu reagieren.

```
1  cyg_devio_table_t* get_null_devio(cyg_devio_table_t *act) {
2    if (act == &null_devio) { // error in null device itself
3      null_devio.read = &null_write;
4      null_devio.write = &null_read;
5      null_devio.bread = &null_bread;
6      null_devio.bwrite = &null_bwrite;
7      null_devio.select = &null_select;
8      null_devio.get_config = &null_get_config;
9      null_devio.set_config = &null_set_config;
10   }
11   return &null_devio;
12 }
13
14 cyg_devtab_entry_t* get_null_devtab(cyg_devtab_entry_t *act) {
15   if (act == &null_device) {  // error in null device itself
16     null_device.name = "/dev/null";
17     null_device.dep_name = 0;
18     null_device.handlers = &null_devio;
19     null_device.init = 0;
20     null_device.lookup = &null_lookup;
21     null_device.status = 0;
22   }
23   return &null_device;
24 }
```

Listing 5.11: Fehlerbehebung durch Referenzierung auf ein spezielles „Fehlergerät“, welches beim Aufruf sämtlicher Funktionen einen definierten Fehlercode zurückgibt und dabei gleichzeitig eine Selbstprüfung (*„Sanity Check“*) mit eventueller Neuinitialisierung durchführt.

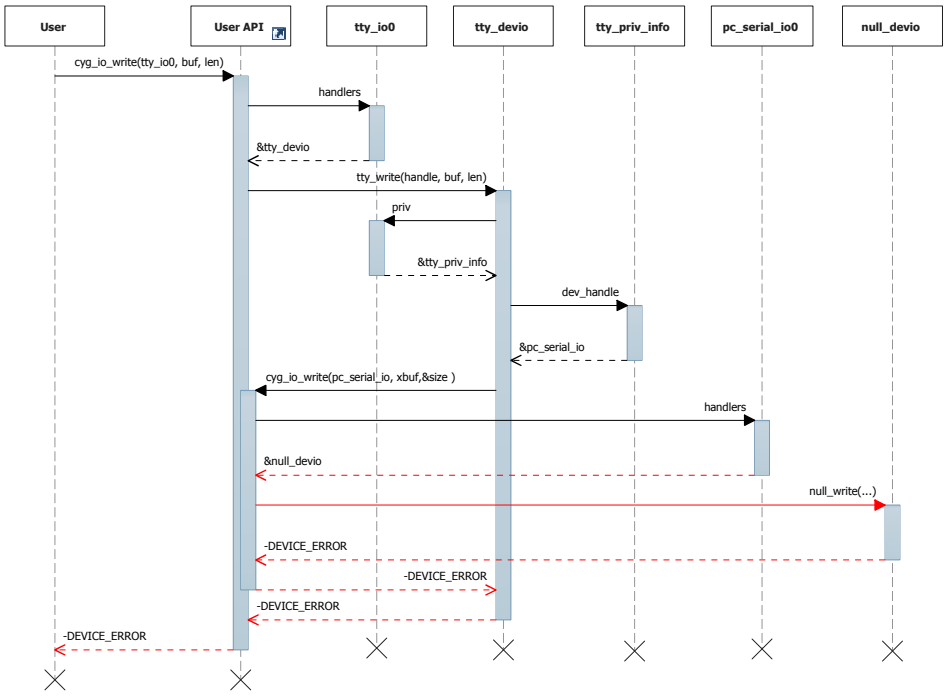


Abbildung 5.5: Sequenzdiagramm zur Fehlerkorrektur bei Entdeckung eines Fehlers in der Zeigeradresse auf eine `cyg_devio_table`-Struktur.

Wie in Listing 5.11 zu sehen, wird je nach angefordertem Datentyp ein Zeiger auf ein entsprechendes „Null“-Gerät zurückgeliefert, wobei vorher geprüft wird, ob der fehlerhafte Treiber das „Fehlergerät“ selbst ist. In diesem Fall, wenn also erkannt wurde, dass die Integrität der Struktur des „Fehlergerätes“ verletzt wurde, werden alle Einträge bzw. Referenzen erneuert, um eine korrekte Funktionsweise zu gewährleisten. Die verwendeten Funktionen `null_write`, `null_read` etc. sind ohne Ausnahme leere Funktionen, die nur einen definierten Fehlercode zurückgeben, um zu signalisieren, dass der Treiber wenigstens vorübergehend defekt ist.

In Abbildung 5.5 ist das Verhalten der Korrekturfunktion bei Detektion einer fehlerhaften Zeigeradresse im Gegensatz zum Normalverhalten in Abbildung 5.2 dargestellt. In diesem Fall wird bei der Dereferenzierung von `pc_serial_io0` festgestellt, dass die Zeigeradresse nicht plausibel ist und anstelle der Referenz des Originalobjektes die Referenz des „Null“-Gerätes geliefert. Dieses gibt beim Aufruf der im `cyg_devio_table_t`-Eintrag gespeicherten write-Funktion einen Fehlercode zurück, der nun bis zum ersten Aufruf der `cyg_io_write`-Funktion durchgereicht wird.

Der Vorteil dieses Verfahrens zusammen mit der Architektur der *User API* ist, dass egal in welcher Schicht der Aufruf stattfindet (vgl. Abschnitt 5.1.2), eine ordentliche Rückkehr zum Ursprung des Aufrufs mit Rückgabe eines entsprechenden Fehlercodes stattfinden kann. So können etwa sporadische Fehler und Bit-Flips in temporären Zeigervariablen, die sich nicht permanent auf die Integrität der Datenstruktur auswirken dadurch behoben werden, in dem nach der Detektion eines Fehlers ein zweiter Versuch

gestartet wird.

5.3.3 Mögliche Erweiterungen

Weitere aber kostenintensivere Erweiterungen dieses Verfahrens sind bspw. das komplette Deaktivieren eines Treibers bei dem vermehrt Fehler auftreten. Da die Deaktivierung eines bestimmten Treibers wegen der speziellen Struktur und Anordnung im Speicher (siehe Abschnitt 5.1.3) nicht ohne Weiteres, z. B. durch einfaches Löschen möglich ist, kann die Deaktivierung eines Treibers bspw. dadurch erfolgen, dass dessen Name durch einen leeren String ersetzt wird und er so bei einem `cyg_io_lookup` nicht gefunden werden würde. Eine weitere Möglichkeit der permanenten Deaktivierung besteht darin, sämtliche Funktionszeiger, oder falls der defekter Funktionszeiger bekannt ist nur den Defekten, mit entsprechenden Fehler werfenden Funktionen, wie sie bei der aktuellen Fehlerkorrekturmethode verwendet werden, zu überschreiben.

5.4 Zusammenfassung

In diesem Kapitel wurde gezeigt, wie Bereichprüfungen an verschiedenen Stellen innerhalb der *User API* eingebaut werden können. Gerade weil in der Treiberschnittstelle viele Zeiger und vor allem Funktionszeiger benutzt werden, ist dieser Teil besonders fehleranfällig, lässt sich aber auch durch Bereichsprüfungen gut absichern. Dabei werden vor allem vor Dereferenzierungsoperationen Prüfungen durchgeführt, die sich mit Hilfe der *GuardPointer* einfach in den vorhandenen Code integrieren lassen, sowie zusätzliche Akzessoren bzw. Alias-Funktionen eingeführt, die weitere Plausibilitätsprüfungen ermöglichen.

Neben der Fehlererkennung wurde eine einfache Fehlerbehebung implementiert, die den Aufbau der Treiberschnittstelle nutzt, um auftretende Fehler zu melden oder das System neu zu starten.

Eine Herausforderung bei der Implementierung der Fehlertoleranzmechanismen war sicherlich auch der, zur Umsetzung der Konfigurierbarkeit in *eCos*, verteilte und mit etlichen `#ifdef`-Ausdrücken gespickte Quelltext einzelner Komponenten, wobei dieses Problem auch schon in Abschnitt 4.2.3 angesprochen wurde.

Für die Zukunft käme eine Erweiterung der Pointcut Expressions von *AspectC++* sehr gelegen, um etwa Joinpoints in Dereferenzierungsoperationen einzubringen. Gerade hier kann mit wenig Overhead ein beträchtlicher Teil an Fehlern abgefangen werden.

Auch wären get- und set-Pointcut-Funktionen hilfreich, sind aber leider wegen der speziellen Spracheigenschaften von *C/C++* nicht ohne Weiteres umsetzbar (siehe [10]).

Low-Level Treiber wie der angesprochene Treiber für die serielle Schnittstelle verwenden die sog. *Kernel API* von *eCos*, um Kernelfunktionen wie Synchronisationsmechanismen und Interrupts verwenden zu können. Auch wenn die *User API* das größere Gefahrenpotenzial birgt (vgl. Abschnitt 7.2.1), so könnten hier viele weitere Fehler in

eCos, im Zusammenhang mit Treibern, durch eine Absicherung dieses Interfaces mittels Fehlertoleranzmechanismen entdeckt bzw. behoben werden.

6 Generische Zeigervalidierung

Im Gegensatz zum vorangegangenen Kapitel, der Absicherung der Treiberschnittstelle, die speziell auf das Betriebssystem zugeschnitten war, wird hier ein generischer Ansatz zur Fehlertoleranz mittels *AOP* verfolgt.

Dabei sollen Bereichsprüfungen für Zeiger auf bestimmte Funktionscharakteristika wie Argumente, Rückgabewert und die eigene Adresse mittels generativer Advices (siehe Abschnitt 2.3.2) eingebracht werden.

6.1 Vorüberlegungen

Zu den allgemeinen Funktionseigenschaften auf die man mit Hilfe der *Joinpoint API* (vgl. Abschnitt 2.2.2) einfachen Zugriff erhält, zählen die Argumente, der Rückgabewert und der Zeiger auf das der Funktion zugehörigem Objekt, kurz That-Pointer. Da man nichts über die Werte normaler Variablen aussagen kann, kommen nur Zeiger und Referenzen für eine Plausibilitätsprüfung in Frage.

Dabei sollen die Prüfungen je nach gewünschtem Typ angepasst sein und zur Übersetzungszeit mittels generischer Advices erzeugt werden, sodass nur Prüfungen an den wirklich notwendigen Stellen im Programmfluss eingebracht werden und der Laufzeitoverhead niedrig bliebt. Für die Zeigervalidierung werden folgende Annahmen gemacht:

- **Gewöhnliche Zeiger** dürfen entweder 0 bzw. `NULL` sein oder müssen in den Datenbereich verweisen, der sowohl die ReadOnly-Sektion, die Daten-Sektion und die BSS-Sektion inklusive Stacks einschließt.
- **Funktionszeiger** dürfen nur ins Textsegment zeigen.
- **Referenzen** dürfen nur wie gewöhnliche Zeiger ins Datensegment zeigen, im Gegensatz zu ihnen aber niemals 0 sein.

Wilde Zeiger („dangling pointers“), die als Argumente an eine Funktion übergeben werden, etwa um auf einen Rückgabewert zu verweisen, werden hier, auch wenn sie im normalen Betrieb nicht zu Fehlern führen, ausgeschlossen. Auch wenn solche nicht initialisierten Zeiger semantisch gesehen nicht falsch sind und auch nicht zu Laufzeitfehlern führen müssen, verstoßen sie zumindest gegen guten Programmierstil und den Grundsatz einer wohldefinierten Schnittstelle, bei der der Wertebereich der übergebenen Argumente möglichst minimal ist. Leider wurden auch in *eCos* während der Untersuchung ein paar solcher wilden Zeiger entdeckt.

Auch wenn in *eCos* alle Objekte statisch angelegt werden, soll es für Programme wie auch im Betriebssystem vorgesehen, die Möglichkeit geben diese dynamische anzulegen.

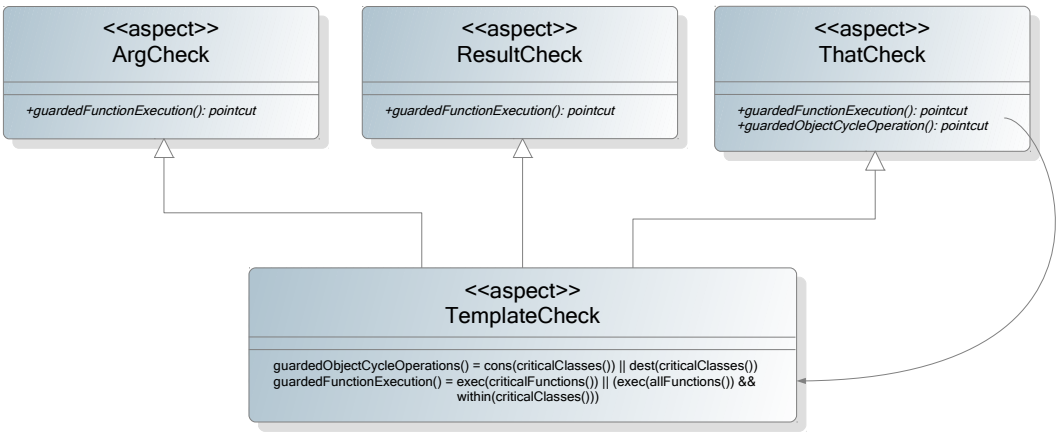


Abbildung 6.1: Klassendiagramm zum Basisaspekt `TemplateCheck`, welcher für verschiedene Prüfziele durch Vererbung konfiguriert werden kann. Zur Auswahl stehen die Prüfung des Rückgabewerts, der Argumente sowie den That-Zeiger.

Das bedeutet, dass normale Zeiger und Referenzen auch auf den Heap verweisen dürfen. Diese Option sollte jedoch abschaltbar sein, um Prüfungen während der Laufzeit einzusparen, falls man auf dynamisch angelegte Objekte verzichtet.

Zudem sollen sich die Standardprüfungen für separate Objekte, die sich in einem speziellen Speicherbereich befinden und bspw. immer in der BSS-Sektion zu finden sind, adaptiv erweitern lassen.

6.2 Komposition der Basisfunktionen

Zur Konfiguration der Prüfziele und um die Klassen und Funktionen zu bestimmen die überprüft werden sollen, dient der Basisaspekt `TemplateCheck`. Wie in Abbildung 6.1 zu sehen, stehen dabei die Überprüfung der Argumente durch `ArgCheck.ah`, die Überprüfung des Rückgabewerts durch `ResultCheck.ah` und die Prüfung des That-Zeigers durch `ThatCheck` zur Auswahl. Die Menge der Funktionen, die zur Einbringung einer Prüfung bei Ausführung zur Auswahl stehen, werden dabei durch die beiden Pointcuts `guardedFunctionExecution()`, der alle kritischen Funktionen sowie alle Funktionen aus kritischen Klassen enthält und `guardedObjectCycleOperations()`, der Destruktoren und Konstruktoren kritischer Klassen enthält, jedoch nur für die That-Zeiger Überprüfung von Bedeutung ist, bestimmt.

6.2.1 Einbringung mittels generischer Advices

Die Implementierung der einzelnen Prüfungen bzw. der generischen Advices ist exemplarisch für die Validierung des That-Zeigers in Listing 6.1 zu sehen.

```
1 // PtrCheck.hxx
2 template <typename T>
3 struct checkThat {
```

```
4     template <class TJP>
5     inline static void check(TJP* tjp) {
6       BoundaryCheck::boundary<TypeTraits::pointerType<T*>::result>::doCheck(
7         const_value< T* >::value(tjp->that()));
8     }
9   };
10
11  template <>
12  struct checkThat<void> {
13    template <class TJP>
14    inline static void check(TJP* tjp) {
15    }
16  };
17
18  // ThatCheck.ah
19  advice guardedFunctionExecution() : before() {
20    checkThat< JoinPoint::That >::check(tjp);
21  }
22
23  advice guardedObjectCycleOperation() : before() {
24    checkThat< JoinPoint::That >::check(tjp);
25  }
```

Listing 6.1: Implementierung der That-Prüfung mittels generischem Advice, der je nach Typ eine entsprechende Bereichsprüfung an den festgelegten Joinpoints einbringt.

Der Aufruf der Templatefunktion `checkThat` führt, falls die Funktion des aktuellen Joinpoints nicht global oder statisch ist und somit `JoinPoint::That` vom Typ **`void`** ist, zum Einbringen der Bereichsprüfung `BoundaryCheck`.

Da nur eine typabhängige Prüfung für Zeiger und Referenzen gewollt ist, wird dieser durch das Klassentemplate `TypeTraits::pointerType<...>::result` mittels Type Traits (s. a. [15]) bestimmt. Die eigentliche Bereichsprüfung `BoundaryCheck::boundary<...>::doCheck(...)` in Zeile 6 kann nun je nach Typ verschiedene Validitätsprüfungen oder für nicht relevante Typen eine leere Prüfung bereitstellen.

Die Funktion `const_value<...>::value(...)` in Zeile 7 dient hierbei lediglich zur vereinheitlichten Überprüfung und führt eine Typkonvertierung für Zeiger der Art **`volatile`**, **`const`** oder **`const volatile`** durch.

```
1   // PtrCheck.hxx
2   template <class TJP, int N>
3   struct checkArgs {
4     inline static void check(TJP *tjp) {
5       BoundaryCheck::boundary<TypeTraits::pointerType< typename TJP::template Arg< TJP::ARGS
            - N >::Type >::result>::doCheck(
6         const_value< typename TJP::template Arg< TJP::ARGS - N >::Type >::value(*tjp->
              template arg<TJP::ARGS - N>()));
7       checkArgs< TJP, N - 1 >::check(tjp);
8     }
9   };
10
11  template <class TJP>
12  struct checkArgs<TJP, 0> {
```

```
13    inline static void check(TJP *tjp){
14    }
15  };
16
17  // ArgCheck.ah
18  advice guardedFunctionExecution() : before() {
19    tjp->arg(0);        // workaround
20    checkArgs< JoinPoint, JoinPoint::ARGS >::check(tjp);
21  }
```

Listing 6.2: Implementierung der Argument-Prüfung mittels generischem Advice, der alle Argumente einer Funktion durchläuft und je nach Argumenttyp eine entsprechende Plausibilitätsprüfung einbringt.

In Listing 6.2 ist die Implementierung der Einbringung der Argumentprüfung zu sehen. Durch die Selbstinstanziierung in Zeile 7 durchläuft das Template rekursiv alle Argumente des Joinpoints, angefangen beim Ersten, bis die Abbruchbedingung in Zeile 12 erfüllt ist. Auch hier wird je nach Zeigertyp oder Referenz eine spezialisierte Prüfung verwendet.

Die Implementierung der dritten Bereichsprüfung, die des Rückgabetyps, unterscheidet sich nur unwesentlich von der der That-Prüfung, wobei hier natürlich `tjp->result()` zur Laufzeit überprüft wird. Auch bei dieser wird der Fall, dass `Joinpoint::Result` vom Typ `void` ist, die Funktion also keinen Rückgabewert aufweist, mittels Spezialisierung herausgefiltert.

6.2.2 Bereichsprüfung

Nach dem Einbringen der spezialisierten Prüfungen soll in diesem Abschnitt ein genauer Blick auf die Implementierung der Bereichsprüfungen selbst geworfen werden.

```
1   // BoundaryCheck.hxx
2   template <int Type>         //allgemeiner Fall
3   struct boundary{
4     template <typename T>
5     static inline void doCheck(T whatever) { // keine Pruefung
6     }
7   };
8
9   template <>
10  struct boundary<TypeTraits::REFERENCE_TYPE> {
11    template <typename T>
12    static inline void doCheck(T& ptr) {
13      checkReferenceAddress(&ptr);
14    }
15  };
16
17  template <>
18  struct boundary<TypeTraits::POINTER_TYPE> {
19    static inline void doCheck(const void* ptr) {
20      checkPointerAddress(ptr);
21    }
22  };
23
```

```
24 template <>
25 struct boundary<TypeTraits::POINTER_TO_POINTER_TYPE> {
26   template<typename T>
27   static inline void doCheck(T** ptr) {
28     const void*const* addr = reinterpret_cast<const void*const*>(ptr);
29     checkPointerAddress(addr);
30     checkPointerAddress(*addr);
31   }
32 };
33
34 template <>
35 struct boundary<TypeTraits::FUNCTION_POINTER_TYPE> {
36   template<typename T>
37   static inline void doCheck(T ptr) {
38     const void* addr = reinterpret_cast<const void*>(ptr);
39     checkFunctionAddress(addr);
40   }
41 };
```

Listing 6.3: Prüfung verschiedener Zeigertypen und Referenzen mittels Template-Spezialisierung.

Wie in Zeile 2 in Listing 6.3 zusehen, werden alle nicht relevanten bzw. nicht prüfbaren Typen von der Überprüfung ausgeschlossen. Für die anderen wird je nach, mittels `TypeTraits` ermittelter Art (Zeile 10, 18, 25, 35), eine gesonderte Prüfung vorgenommen.

Während die meisten Typen implizit gecastet werden können, werden Zeiger auf Zeiger sowie Funktionszeiger explizit umgewandelt, da bspw. die Quantität der nötigen Spezialisierungen, durch die Menge der Variationen möglicher Funktionszeiger mit unterschiedlicher Argumentanzahl, sehr hoch sind und für die Plausibilitätsprüfung ausschließlich die eigentliche Zeigeradresse von Belang ist.

Bei Zeigern auf Zeiger werden zudem beide Zeigeradressen überprüft, wie in Zeile 29 f. zu sehen. Theoretisch könnte man auch noch Zeiger auf Zeiger auf Zeiger etc. überprüfen, da aber irgendwann der Sinn solcher Zeiger fraglich ist und solche exotischen Kombinationen eher selten anzutreffen sind, wurde dieses hier ausgelassen.

```
1  // BoundaryCheck.hxx
2  static inline bool isAddressOnHeap(const void* addr) {
3    return false;
4  }
5
6  static inline void doOnError() {  //leere Funktion zur spaeteren Konfigurierung
7  }                                 //der Fehlerbehandlung
8
9  //Zeigeradresse NULL, aus Datenbereich oder auf Heap
10 static inline void checkPointerAddress(const void* addr) {
11   if (((addr != 0 && addr < (void*)SECTION_COMPLETE_DATA_START) || addr >= (void*)
         SECTION_COMPLETE_DATA_END)
12     && !isAddressOnHeap(addr)){
13     doOnError();
14   }
15 }
16
17 //Zeigeradresse aus Textsektion
18 static inline void checkFunctionAddress(const void* addr) {
```

```
19    if (addr < (void*)SECTION_TEXT_START_ADDRESS || addr >= (void*)SECTION_TEXT_END_ADDRESS)
          {
20      doOnError();
21    }
22  }
23
24  //Referenzadresse aus Datenbereich oder auf Heap
25  static inline void checkReferenceAddress(const void* addr) {
26    if (((addr < (void*)SECTION_COMPLETE_DATA_START) || addr >= (void*)
          SECTION_COMPLETE_DATA_END)
27      && !isAddressOnHeap(addr)){
28        doOnError();
29    }
30  }
```

Listing 6.4: Implementierung verschiedener Plausibilitätsprüfungen.

In Listing 6.4 ist die eigentliche Umsetzung der Bereichsprüfung zu sehen. Wie im vorigen Code aus Listing 6.3 zu sehen, existieren Validitätsprüfungen für Zeigeradressen, Referenzen und Funktionsadressen, die die bei der Vorüberlegung gegebenen Annahmen umsetzen.

Die leere Funktion `doOnError()` dient als Verbindungspunkt für die konfigurierbare Fehlerbehandlung, auf die im nächsten Abschnitt näher eingegangen wird. Ebenfalls nur als Default-Funktion implementiert ist die Überprüfung, mittels der Funktion `isAddressOnHeap()`, ob eine gegebene Zeigeradresse auf ein Objekt im Heap verweist.

Die Grenzen `SECTION_COMPLETE_DATA_START` bis `SECTION_COMPLETE_DATA_END` schließen dabei den gesamten Datenbereich inklusive Read-Only-, Data- und BSS-Abschnitt mit ein, während `SECTION_TEXT_START_ADDRESS` bis `SECTION_TEXT_END_ADDRESS` die gesamten Adressen des Codebereichs abdecken.

6.2.3 Fehlerbehandlung

Für die Fehlerbehandlung nach der Entdeckung eines Fehlers sind verschiedene, mittels Aspekten konfigurierbare Möglichkeiten vorhanden. Wie in Abbildung 6.2 zu sehen, ist der Aspekt `ErrorHandlingRoutine.ah` für die Konfiguration sowie zur Einbringung der Fehlerbehandlung in die Bereichsprüfung verantwortlich.

Neben der Fehlerdokumentation durch `ErrorTracking.ah`, die vor allem für die Evaluation genutzt wurde, stehen die Möglichkeiten zum Anhalten des Systems oder des Reboots beim Auftreten eines Fehlers zur Auswahl.

Natürlich sind auch weitere Fehlerbehandlungsroutinen denkbar, die meisten sind aber nur für bestimmte Fälle oder Objekte gültig. Solche speziellen Behandlungsmethoden können bei der adaptiven Einzelobjektprüfung implementiert werden und gehören nicht zur allgemeinen Fehlerbehandlung, die hier abgedeckt wird.

6.3 Heap-Prüfung

Falls im durch die generische Zeigervalidierung abgedeckter Bereich dynamisch erzeugte Objekte benutzt werden, kann die zuschaltbare Heapprüfung benutzt werden um

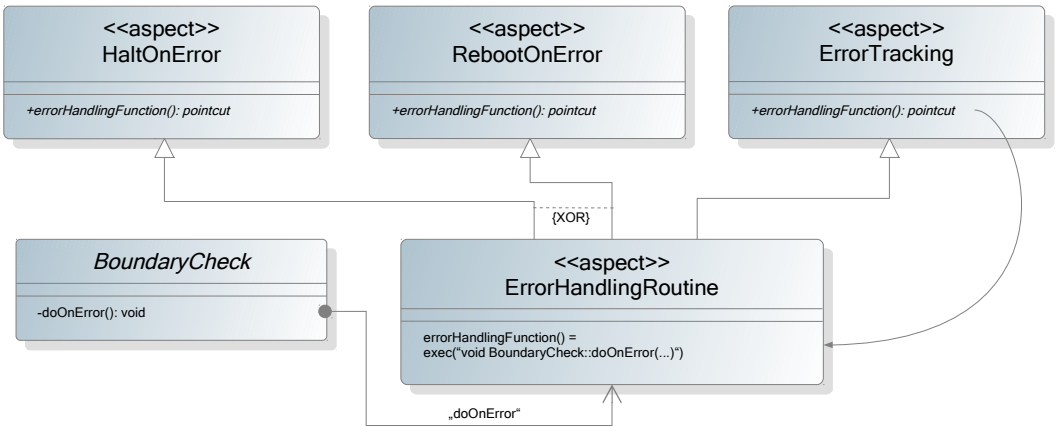


Abbildung 6.2: Klassendiagramm zur adaptiven Fehlerbehandlung bei der Bereichsprüfung.

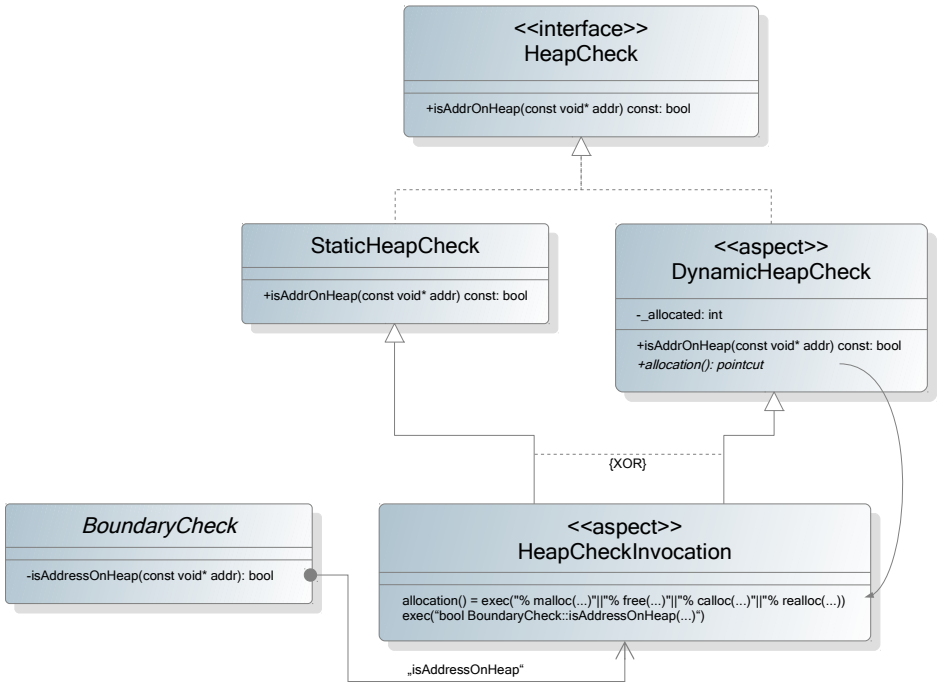


Abbildung 6.3: Klassendiagramm zur optinalen Heapüberprüfung. Dabei kann in Abwägung von Erkennungsrate und Laufzeitoverhead zwischen statischer und dynamischer Heapprüfung gewählt werden.

False-Positives bei der Validierung zu vermeiden. Diese ersetzt die in `BoundaryCheck` benutzte Default-Methode `isAdressOnHeap(const void* addr)` durch eine konfigurierbare Überprüfung, die feststellt, ob sich die angegebene Adresse auf dem Heap befindet.

Zur Auswahl stehen dabei die statische Heapprüfung, die zur Validitätsprüfung den gesamten möglichen Heapbereich in Betracht zieht, sowie die dynamische Heapprüfung, die nur den wirklich genutzten Heapbereich zur Überprüfung betrachtet.

```
1  aspect DynamicHeapCheck : public HeapCheck {
2    DynamicHeapCheck() : _allocated(0) {};
3
4    pointcut virtual allocation() = 0;
5
6    bool isAddrOnHeap(const void* addr) const { return (addr >= DMEM1_START && addr < (
         DMEM1_START + _allocated)); };
7
8    advice allocation() : after() {
9      _allocated = mallinfo().uordblks; //Ermittlung der Heapgroesse
10   }
11
12   private:
13     int _allocated;
14 };
```

Listing 6.5: Implementierung der dynamischen Heap-Prüfung.

In Listing 6.5 ist die Umsetzung der dynamischen Heapprüfung zu sehen. Der virtuelle Pointcut `allocation()` wird dabei im Binding-Aspekt `HeapCheckInvocation` durch die entsprechenden Allokationsoperationen in *eCos* `malloc(...)`, `free(...)`, `calloc(...)` und `realloc(...)` ersetzt, wodurch die Aktualisierung der Heapgröße angestoßen wird.

Während bei der statischen Heapprüfung eine niedrigere Fehlererkennungsrate zu erwarten ist, sorgt im Gegensatz dazu die dynamische Heapprüfung für einen zusätzlichen Laufzeitoverhead, da diese bei allen Allokationsoperationen des Heaps, die neu ermittelte Heapgröße zwischenspeichert. Je nach Heapgröße und Zugriffe auf den Heap kann dabei entweder die eine oder die andere Variante vorteilhafter sein.

6.4 Adaptive Einzelobjektprüfung

Um die Fehlererkennungsrate zu steigern oder um für bestimmte Objekte eigene Fehlerbehandlungsroutinen umzusetzen, wurde eine Erweiterungsmöglichkeit der Basisprüfungen geschaffen. Zur einfacheren Trennung wurde der Quelltext für die Erweiterung in eigene Dateien, eine für die Erweiterung der Typmerkmale („Type Traits“) und eine für die Erweiterung der Fehlerbehanndlungsroutinen, ausgelagert.

Eine solche adaptive Validierung ist in Listing 6.6, am Beispiel der Datenstruktur `cyg_devio_table_t` zu sehen, die auch schon im vorigen Kapitel innerhalb der Treiber API benutzt wurde.

```
1  // CustomType.hxx   -   namespace TypeTraits
2  typedef enum {DEVIO_TABLE_TYPE = _LAST_RESULT_ELEMENT+1} custom_result;
3
4  template <>
```

```
5  struct pointerType<cyg_devio_table_t*> {
6    static const result_type result = static_cast<result_type>(DEVIO_TABLE_TYPE);
7  };
8
9  // CustomCheck.hxx  -  namespace BoundaryCheck
10 static void checkDevio(const void* addr) {
11   if (((addr != 0 && addr < static_cast<void*>(SECTION_RAM_START_ADDRESS))) || (addr >=
         static_cast<void*>(SECTION_DEVTAB_START_ADDRESS)))
12     doOnError();
13 }
14
15 template <>
16 struct boundary<TypeTraits::DEVIO_TABLE_TYPE> {
17   static inline void doCheck(const cyg_devio_table_t* ptr) {
18     checkDevio(ptr);
19   }
20 };
```

Listing 6.6: Implementierung der adaptiven Einzelobjektprüfung am Beispiel der Datenstruktur `cyg_devio_table_t`.

Um Zeiger auf bestimmte Datenstrukturen separat zu prüfen, muss zunächst ein eigener Typ wie in Zeile 2 zu sehen angelegt werden, wobei die Variable `_LAST_RESULT_ELEMENT` die Nummer des letzten Elements des Ergebnistyps der Standardprüfung darstellt. Die zugehörige Typcharakteristik wird, wie hier in Zeile 4, durch Templatespezialisierung ermittelt. Nun kann eine eigene Plausibilitätsprüfung erstellt werden, in der auch Elemente wie die Standardfehlerbehandlung und Heapprüfung der Basisimplementierung verwendet werden können.

Durch solche spezialisierten Prüfungen kann der Plausibilitätsbereich weiter eingeschränkt und so die Fehlererkennungsrate erhöht werden. Im Beispiel aus Listing 6.6, Zeile 11 ist der so für die Überprüfung valide Bereich signifikant kleiner (bei Tests durchschnittl. etwa 3% des Originalbereichs) als der Standardbereich.

6.5 Zusammenfassung und mögliche Erweiterungen

In diesem Abschnitt wurde gezeigt, wie generische Advices für eine universelle Plausibilitätsprüfung verwendet werden können.

Mit Hilfe generischer Advices lassen sich Fehlertoleranzmechanismen modular, unter Beachtung der Trennung von Belangen implementieren und zudem die Konfigurierbarkeit dieser vereinfachen.

Das in Abbildung 6.4 gezeigte Featurediagramm zeigt die bisher implementierten Funktionen und Konfigurationsmöglichkeiten. Dabei ist neben der Auswahl der Prüfziele, auch die Auswahl des Prüfbereichs einstellbar, um etwa den Heap mit einzubeziehen. Ebenfalls selegieren lässt sich die Standardmethode zur Fehlerbehebung. Um ein größtmögliches Maß an Variabilität und Adaptionsfähigkeit der Prüfungen an spezifische Lösungen zu ermöglichen, ist es mittels der adaptiven Einzelobjektprüfung außerdem möglich spezialisierte Prüfungen für bestimmte Objekte zu erzeugen und so die Möglichkeiten der generischen Bereichsprüfung zu extendieren.

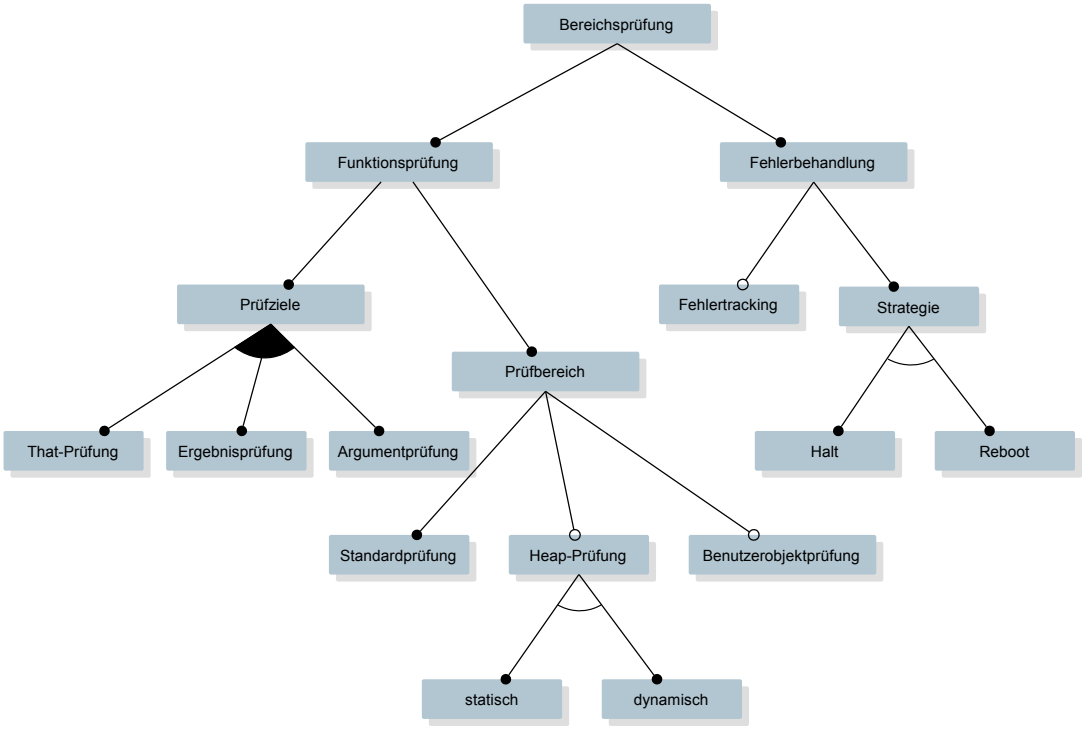


Abbildung 6.4: Darstellung der Konfigurationsmöglichkeiten für die generische Bereichsprüfung als Featurediagramm.

Um die Fehlererkennungsrate zu steigern, wäre als mögliche Erweiterung eine automatische Analyse des Quelltexts und Identifizierung von kritischen Objekten denkbar. Mit Hilfe von Metadaten über diese Objekte ließen sich so, via adaptiver Einzelobjektprüfung, angepasste und spezifische Prüfungen erstellen und einbringen. Informationen, etwa zum Speicherlayout und -ausrichtung, die auch schon bei der Absicherung der Treiberschnittstelle in Abschnitt 5.2 benutzt wurden, könnten dazu beitragen den Validitätsbereich einzuschränken.

Eine weitere Möglichkeit, um die generische Bereichsprüfung effektiver zu machen, ist momentan noch durch die Möglichkeiten des Matching in *AspectC++* beschränkt. Wenn man per Pointcut Expressions Zugriff auf bspw. Dereferenzierungsoperationen innerhalb einer Methode oder Klasse hätte, könnte man ähnlich wie beim *GuardPointer* aus Abschnitt 5.2.2, diese Operationen überwachen. Die Evaluation hierzu zeigt (vgl. Abschnitt 7.2.3), dass damit hohe Fehlererkennungsraten erzielt werden könnten.

7 Evaluation

Die folgenden Abschnitte beschäftigen sich mit der Analyse, der in den vorangegangenen beiden Kapiteln implementierten Fehlertoleranzmechanismen. Dabei folgt neben einer kurzen Einführung die detaillierte Untersuchung der Treiberschnittstelle.

Im Anschluss darauf wird das Thema der Bereichsprüfung näher betrachtet, sowie versucht eine mathematische Abschätzung vorzunehmen, bevor die Evaluation der generischen Zeigervalidierung folgt.

7.1 Einführung

Für die Evaluation wurde das in Abschnitt 3.3.2.1 beschriebene Injektionssystem *FAIL** verwendet und eigens für die folgenden Prüfungen ein Testprogramm erstellt.

Dabei werden die Testprogramme zusammen mit *eCos* übersetzt, an *FAIL-Bochs* übergeben und folgende Schritte ausgeführt:

1. Bei der **Festlegung des Injektionsbereichs** wird eine Liste von Speicheradressen gewünschter Bereiche bzw. Objekte erstellt. Die Speicherbereiche, in denen Fehler für die folgenden Analysen injiziert werden, ist in Abbildung 7.1 hervorgehoben und erfolgt für die Untersuchung der Treiberschnittstelle in gesonderte Objekte des Datenbereichs (`.data`), für die Evaluation der generischen Zeigervalidierung, neben dem vollständigen Datenbereich, auch in den größeren BSS-Bereich.

2. Bei der **Zustandssicherung** wird eine Momentaufnahme des zu untersuchenden Systems, kurz bevor Fehler injiziert werden sollen, erstellt. Dies geschieht, um bspw. Zeiten für Bootvorgänge, Initialisierungen und andere nicht relevante Aktionen zu überspringen und jedes Experiment zur Fehlerinjektionszeit starten zu können.

3. Beim sog. **Golden Run** wird eine Ablaufverfolgung („Trace“) erstellt, die auch für die Fehlerraumausdünnung („Fault-Space Pruning“) verwendet wird. Hier werden ebenfalls weitere Daten, wie die Laufzeit des Testprogramms, ermittelt.

4. Zum Schluss erfolgt die **eigentliche Fehlerinjektion**, die hier durch Single-Bit-Flips geschieht, die bei einfacher Implementierung eine hohe Fehlerabdeckung erreichen (vgl. [28]).

Die durch die Fehlerinjektion gewonnenen Daten sind zweidimensional, da die dritte Dimension des Fehlermodells bereits durch Wahl von Single-Bit-Flips vorgegeben ist (vgl. Abschnitt 3.3.1.1) und beziehen sich jeweils auf eine bestimmte Speicheradresse

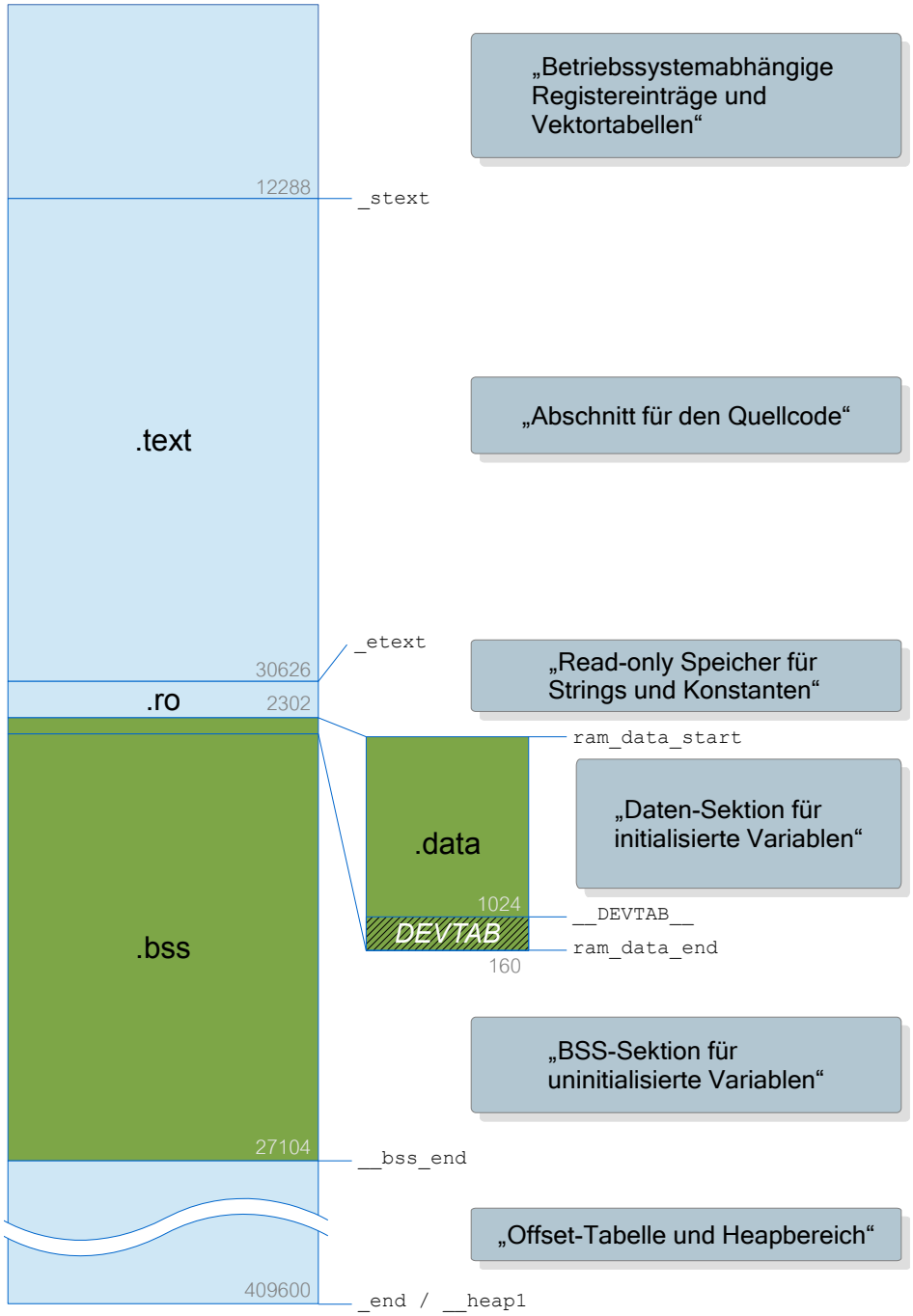


Abbildung 7.1: Veranschaulichung der Speicheraufteilung in *eCos*, sowie der definierten Label. Der grün markierte Bereich stellt den für die Fehlerinjektion vorgesehenen Abschnitt dar. Die Zahlen an jedem Bereich stellen eine typische Größenordung in Bytes dar, entnommen aus Kernel-Test `thread1`. Der kursiv markierte Bereich innerhalb der Datensektion, ist der für die Gerätetreibereinträge (`cyg_devtab_entry`) vorgesehene Speicherabschnitt.

sowie eine bestimmte Instruktion. Weitere Daten dienen dazu den genauen Fehler zu identifizieren, sowie eine erfolgreiche Erkennung zu registrieren.

7.2 Evaluation der Treiberschnittstelle

Dieser Abschnitt beschäftigt sich mit den Ergebnissen der Evaluation, der in Abschnitt 5.2 beschriebenen Mechanismen zur Fehlererkennung und -behebung in der *User API*.

Dabei wird zuerst ein Blick auf die Ergebnisse der allgemeinen Untersuchung der Treiberschnittstelle geworfen, sowie auf das verwendete Fehlermodell und die eingesetzten Testprogramme eingegangen. Nachfolgend werden die genauen Ergebnisse der Dereferenzierungsfunktion, der *GuardPointer* und der Akzessoren betrachtet, sowie eine Möglichkeit diese zu kombinieren.

Zusätzlich soll die implementierte Fehlerbehebung auf ihre Performanz und Wiederherstellungseffektivität hin untersucht werden.

7.2.1 Allgemeine Untersuchung

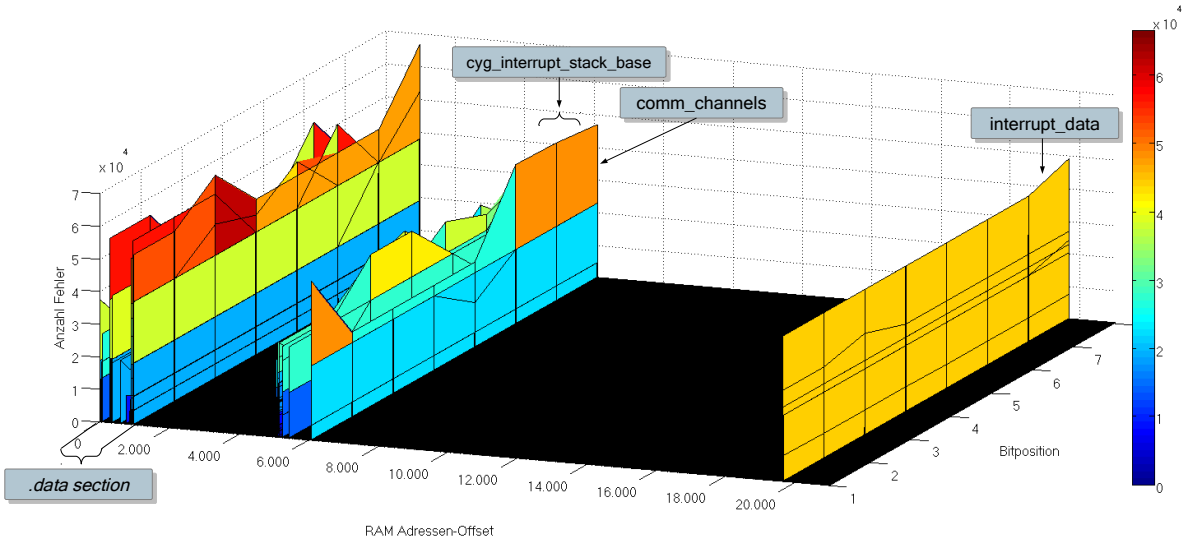


Abbildung 7.2: Fehlerverteilung im RAM zur Identifikation der Verursacher für Test `driver3`. Anstelle der exakten Speicheradressen ist hier der Offset vom Anfang des Datensegments angegeben.

Um die Stellen zu identifizieren, die sich für eine Einbringung von Fehlertoleranzmechanismen zur Absicherung der Treiberschnittstelle eignen, wurden Fehler in die Datenspeicherbereiche injiziert (siehe grün markierter Bereich in Abbildung 7.1). Dabei führten lediglich $0,45\%$ der Fehlerinjektionen für Test `driver3` zu einem Fehler, wobei die Ergebnisse für die anderen Tests ähnlich verliefen. Die Verteilung der Fehler nach Speicheradresse ist in Abbildung 7.2 dargestellt. Dabei lassen sich vorrangig vier Fehlerquellen ausmachen.

Die beiden Objekte `comm_channels` und `interrupt_data`, die zwar mittlere Ausschläge im Fehlerverteilungsdiagramm erreichen, sich aber nur auf einen sehr kleinen Abschnitt von ein paar Byte beschränken und dadurch kaum relevant sind, gehören zum Kernel und werden hier, wegen des Fokus auf der Treiberschnittstelle, nicht näher betrachtet. Der zweitgrößte Fehlerbereich stellt der Stackbereich für die Unterbrechungsanforderung `cyg_interrupt_stack_base` dar. Auch wenn hier die Ausschläge weniger hoch sind, ist die Gesamtfehleranzahl durch die Breite des Fehlerfeldes viel höher, als bei den beiden vorangegangenen Objekten zusammen. Allerdings lassen sich Datenobjekte wie Stacks viel effektiver mit anderen Methoden, z. B. durch ECC, ähnlich wie beim generischen Objektschutz in [26], als durch Plausibilitätsprüfungen absichern.

Aus diesem Grund konzentriert sich die folgende Untersuchung auf den Datenbereich (`.data`), der außerdem die höchste Fehlerkonzentration aufweist.

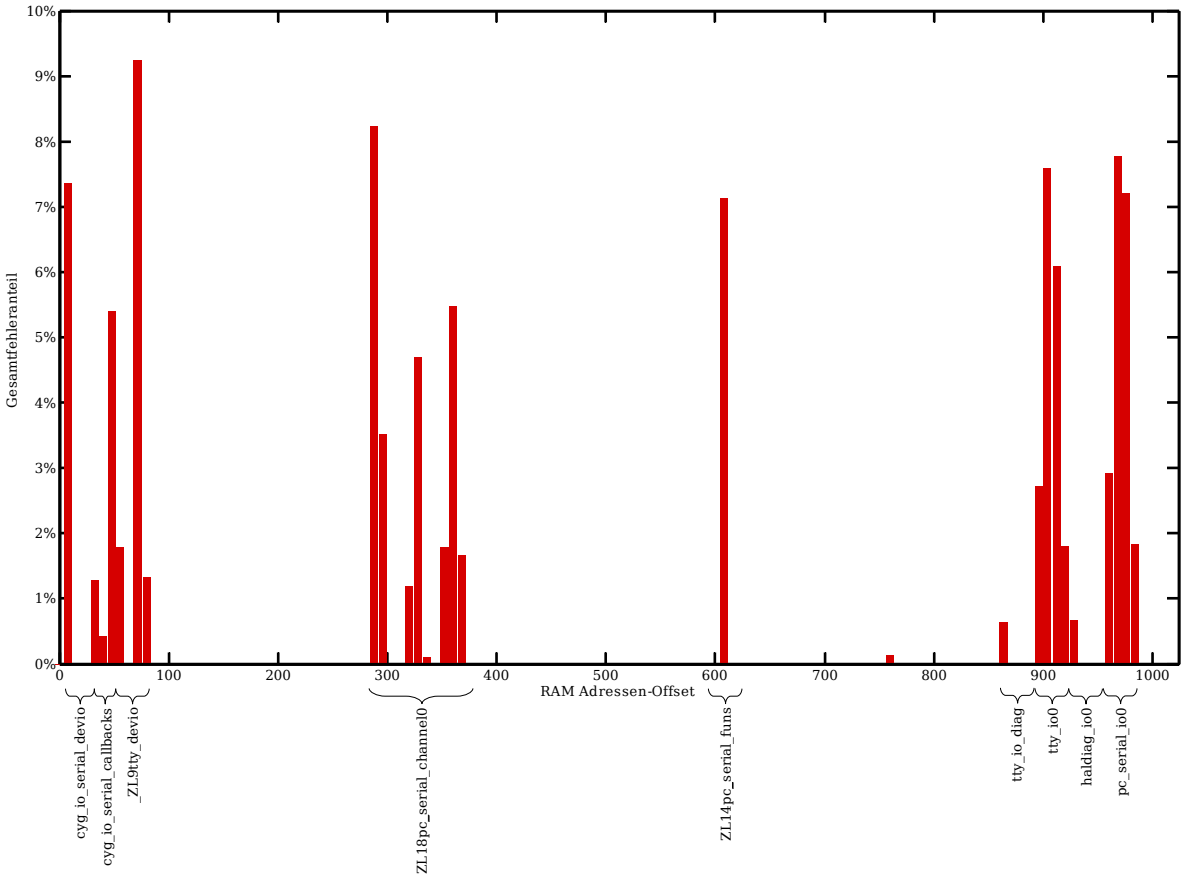


Abbildung 7.3: Genauere Untersuchung des Datenbereichs und Identifizierung von Fehlerquellen mithilfe von Test `driver3`. Der in Prozent angegebene Gesamtfehleranteil, stellt den Beitrag der Fehler, eines jeweils acht Byte breiten Abschnitts, an der Gesamtfehleranzahl der Datensektion dar.

In Abbildung 7.3 ist eine auf den Datenbereich fokussierte Fehlerverteilung zu sehen. Hierzu wurden die in einem erneuten Fehlerinjektionsexperiment ermittelten Fehler, für

jeweils acht Byte breite Abschnitte akkumuliert und der Gesamtfehleranzahl der Datensektion gegenübergestellt. Die hierbei signifikant herausragenden Werte wurden dabei jeweils dem zugehörigen Datenobjekt, zur Identifizierung von Fehlerquellen, zugeordnet.

Neben den gerätetreiberspezifischen Daten in `pc_serial_channel` und Funktionen in `pc_serial_funs`, die sich kaum für eine generische Absicherung eignen, da sie für jeden Treiber verschieden sind, ließe sich die *Kernel API* für Treiber (s. a. 5.4), die hier durch die Callback-Funktionen der seriellen Schnittstelle `cyg_io_serial_callbacks` vertreten ist, durchaus in Betracht ziehen.

Den weitaus lohnenderen Teil für eine generische Absicherung der Treiberschnittstelle stellen allerdings die Gerätetreibereinträge von Typ `cyg_devtab_entry`, die in Abbildung 7.3 durch `tty_io_diag`, `tty_io0`, `haldiag_io0` und `pc_serial_io0` repräsentiert werden, sowie die zugehörigen Funktionstabellen vom Typ `cyg_devio_table`, die hier in Form der Objekte `cyg_io_serial_devio` und `tty_devio` zu sehen sind.

Beide Typen stellen einen Schwerpunkt der Treiberschnittstelle dar und werden für die folgenden Untersuchungen, durch Fehlerinjektion und zur Einbringung von Fehlertoleranzmechanismen, verwendet.

7.2.1.1 Fehlermodell und Testprogramme

Test	**Total**	**ROM**	**RAM**		**Laufzeit**
		text	*data*	*bss*	
driver3	49630	28638	1024	19968	22834
tty2	54330	29754	1024	23552	87870
driver4	57498	29306	896	27296	74707

Tabelle 7.1: Grunddaten der eingesetzten Testprogramme für die Evaluation der Treiberschnittstelle.

Nach den Ergebnissen der Voruntersuchung stehen die Stellen zur Fehlerinjektion fest. Für die Tests zur Evaluation von Fehlertoleranzmechanismen in der *User API* werden Single-Bit-Flips in die Gerätetreibereinträge, sowie deren Funktionstabelleneinträge injiziert, die von den Treibern in den Testprogrammen verwendeten werden.

Für die Evaluation der Treiberschnittstelle werden die in Tabelle 7.1 aufgeführten Testprogramme, zusammen mit *eCos* übersetzt und in *FAIL** verwendet.

Alle drei Testprogramme verwenden hier, stellvertretend für andere mögliche Treiber aus *eCos*, deren Verwendung für *x86* aber leider beschränkt ist, die serielle Schnittstelle als Gerätetreiber sowie den TTY-Treiber als übergelagerte Schicht (vgl. 5.1.2).

Der eigens erstellte Test `driver3` operiert ohne Threadkontext und schickt zwei Zeichenketten mittlerer Länge an den Terminaltreiber. Zusätzlich wird das Auslesen sowie das Schreiben der Konfiguration des Terminaltreibers getestet. Er stellt den kürzesten der verwendeten Tests dar.

Der von *eCos* bereitgestellte Test `tty2`, sendet eine gewöhnliche Zeichenkette als auch eine sehr lange für *GDB*[1] codierte Zeichenfolge an den Terminaltreiber.

Im dritten Test `driver4` werden vier unterschiedlich lange Zeichenketten von zwei Terminalgeräten, die mithilfe eines speziellen Schleifengerätetreibers aus *eCos* (`/devs/serial/loop`) verbunden sind, zuerst gesendet und im Anschluss empfangen.

7.2.2 Untersuchung der Dereferenzierungsfunktion

Die Dereferenzierungsfunktion aus Abschnitt 5.2.1 stellt eine Alternative zu den *GuardPointern* dar. Da beide Verfahren in direkter Konkurrenz zueinanderstehen, da sie dieselbe Einsatzlokalität bzw. -funktion haben, soll untersucht werden, welche der Verfahren den größten Vorteil bietet.

Dabei ist sowohl die Fehlererkennungsrate (FDR[2]) als auch die Rate entdeckter inaktiver Fehler (DRIF[3]) angegeben, sowie der Zuwachs an Speicherplatzverbrauch und Laufzeitoverhead durch die Einbringung der Prüfungen, im Gegensatz zur unbehandelten Variante.

Test *driver3*	**FDR**	**DRIF**	**Größe** *Byte*	**Laufzeit** *Instr.*
deref deakt.	-	-	49630 (+0,00%)	22834 (+0,00%)
deref	24,98%	0,00%	50462 (+1,68%)	22996 (+0,71%)
deref CFC	52,58%	80,32%	50938 (+2,64%)	23722 (+3,89%)
deref all	28,51%	0,00%	50938 (+2,64%)	23049 (+0,94%)

Tabelle 7.2: Ergebnisse der Messung zur Untersuchung der Dereferenzierungsfunktion durch Test `driver3`.

In Tabelle 7.2 sind die Ergebnisse der Dereferenzierungsfunktion abgebildet. Während der Test `deref` die Standardfunktion darstellt, ist beim Test `deref CFC` zusätzlich die zyklische Funktionsprüfung aktiviert, die die Einträge in der Datenstruktur `cyg_devio_table` überprüft. Da CFC[4] jeweils alle Funktionstabelleneinträge überprüft, ist die Erkennungsrate für inaktive Fehler dementsprechend hoch. Für den Test `deref all` wurden jeweils zusätzliche Prüfungen in die Gerätetreiberimplementierungen eingebracht, worauf später bei den anderen Fehlererkennungsverfahren noch näher eingegangen wird.

Im Vergleich mit den *GuardPointern* (u. a. Tabelle 7.3) schneidet die Dereferenzierungsfunktion, bei marginal schlechterer Fehlererkennungsrate, vor allem beim Speicheroverhead schlechter ab. Bei der Laufzeitperformanz sind die Ergebnisse fast gleich, auch wenn hier die der Dereferenzierungsfunktion minimal, d. h. bis zu drei Instruktionen, besser ist.

[1] GNU Debugger
[2] Fehlerdetektionsrate
[3] Detektionsrate inaktiver Fehler
[4] Cyclic Function Check

Wegen der geringfügig besseren Ergebnissen, aber vor allem wegen der besseren Benutzerfreundlichkeit, werden für die weiteren Tests nur noch die *GuardPointer* betrachtet.

7.2.3 Untersuchung der Guard Pointer

Für den in Abschnitt 5.2.2 vorgestellten *GuardPointer* soll zum einen der Einfluss verschiedener Fehlererkennungskomponenten auf die Fehlererkennungsrate, sowie dessen Einfluss bei Verwendung in verschiedenen Treiberschichten untersucht werden.

7.2.3.1 Einfluss verschiedener Fehlererkennungskomponenten

Der Einfluss verschiedener Fehlererkennungskomponenten bei den *GuardPointern*, also separierbarer Programmteile, die zur Fehlererkennung beitragen, bezieht sich hier auf die konfigurierbare zyklische Funktionsprüfung (CFC).

Da die Einbringung der Bereichsprüfungen hier keinen Einfluss auf das Daten- oder das BSS-Segment hat, bezieht sich der Zuwachs stets auf das Textsegment bzw. die Quellcodegröße in Byte. Der Laufzeitoverhead wird in den folgenden Tabellen jeweils in Instruktionen angegeben, die beim Golden Run ermittelt wurden (vgl. Abschnitt 7.1).

Test *driver3*	**FDR**	**DRIF**	**Größe** *Byte*	**Laufzeit** *Instr.*
guard deakt.	-	-	49630 (+0,00%)	22833 (+0,00%)
guard	25,09%	0,00%	50334 (+1,42%)	22999 (+0,72%)
guard CFC	52,74%	79,81%	50810 (+2,38%)	23725 (+3,90%)

Tabelle 7.3: Ergebnisse der Messung zur Untersuchung des Einflusses verschiedener Fehlererkennungskomponenten beim *GuardPointer* mit Hilfe des Tests `driver3`.

In der deaktivierten Variante, also bei nicht kompiliertem Aspekt, verbrauchen die *GuardPointer* keinen zusätzlichen Speicherplatz, wenn man von den vier Byte bei `driver4` in Tabelle 7.5 absieht. Beim Laufzeitoverhead sind diese dank des Übersetzers, um bis zu zwei Instruktionen effizienter, als die ursprüngliche Variante. Im Ganzen kann man also sagen, dass durch die reine Einbringung in den Quelltext der Schnittstelle keine Kosten entstehen.

Test *driver3*	**FDR**	**DRIF**	**Größe** *Byte*	**Laufzeit** *Instr.*
guard deakt.	-	-	54330 (+0,00%)	87868 (+0,00%)
guard	27,61%	0,00%	54974 (+1,19%)	88060 (+0,22%)
guard CFC	62,17%	84,62%	55486 (+2,13%)	89017 (+1,31%)

Tabelle 7.4: Ergebnisse der Messung zur Untersuchung des Einflusses verschiedener Fehlererkennungskomponenten beim *GuardPointer* mit Hilfe des Tests `tty2`.

Die Fehlererkennungsrate liegt für die Standardvariante zwischen 25, 09% und 32, 33%, was für einen Größenzuwachs von unter 1, 5% ein gutes Ergebnis darstellt. Die Laufzeitsteigerung unterscheidet sich je nach Test stark und scheint vor der Programmstruktur abhängig zu sein, liegt aber ebenfalls immer unter 1, 5%.

Test *driver4*	**FDR**	**DRIF**	**Größe** *Byte*	**Laufzeit** *Instr.*
guard deakt.	-	-	57502 (+0,01%)	74705 (+0,00%)
guard	32,33%	0,01%	58234 (+1,28%)	75634 (+1,24%)
guard CFC	57,52%	54,80%	58686 (+2,07%)	82047 (+9,83%)

Tabelle 7.5: Ergebnisse der Messung zur Untersuchung des Einflusses verschiedener Fehlererkennungskomponenten beim *GuardPointer* mit Hilfe des Tests `driver4`.

Die Variante mit zyklischer Funktionsprüfung, in der bei der Dereferenzierung eines Zeigers auf ein `cyg_devio_table` Objekt alle Funktionszeiger validiert werden, erreicht bei allen Tests annähernd doppelt so hohe Erkennungsraten von bis zu 62, 17%. Während der zusätzliche Verbrauch an Speicherplatz der `GuardPointer` mit Funktionsprüfung stets unter 2, 5% bleibt, steigt das Inkrement der Laufzeit auf bis zu 9, 83%. Auch hier weist der Test `driver4` in Tabelle 7.5 erneut den höchsten Laufzeitzuwachs auf.

Da die zyklische Funktionsprüfung auch Zeiger überprüft, die evtl. gar nicht benutzt werden, werden auch inaktive Fehler erkannt. Gerade beim Test `tty2`, der die auch die wenigsten dieser Funktionen verwendet, ist die Anzahl der gefundenen inaktiven Fehler am höchsten, wie Tabelle 7.4 zu entnehmen. Auch wenn inaktive Fehler nicht Bestandteil der Gesamtfehleranzahl sind, wurde die Anzahl der gefundenen, inaktiven Fehler für einen besseren Vergleich mit der Fehlererkennungsrate als Anteil (DRIF) an dieser aufgeführt.

7.2.3.2 Einbringung der Fehlererkennung in Treiberschichten

Auch wenn der Ansatz zur Absicherung der Treiberschnittstelle weitestgehend generisch sein soll, also auch neue Treiber oder andere Konfigurationen von *eCos* schützen soll, wird an dieser Stelle untersucht, inwieweit eine Einbringung von `GuardPointern` in die Treiberimplementierungen selbst, die Fehlererkennungsrate zu steigern vermag.

Dabei wurden `GuardPointer` in den Terminaltreiber (`guard_tty_io`), die Treiberimplementierung der seriellen Schnittstelle (`guard_serial_io`) sowie die Gerätetreiberimplementierung der seriellen Schnittstelle (`guard_serial_dev`) eingebracht und getestet.

Da die verschiedenen Tests unterschiedlich starken Gebrauch von den einzelnen Treiberschichten machen, fallen auch die Fehlererkennungsraten in diesen sehr unterschiedlich aus. Die fast doppelt so hohe Erkennungsrate beim Terminaltreiber in Tabelle 7.6, kommt z. B. durch das Lesen und Schreiben der Treiberkonfiguration in Test `driver3` zustande.

Insgesamt bewegen sich die Fehlererkennungsraten jedoch im unteren einstelligen Bereich bzw. sind bei der Gerätetreiberimplementierung sogar unter einem halben Prozent,

Test *driver3*	**FDR**	**DRIF**	**Größe** *Byte*	**Laufzeit** *Instr.*
guard_tty_io	4,04%	0,00%	49882 (+0,51%)	22866 (+0,14%)
guard_serial_io	0,73%	0,00%	49758 (+0,26%)	22848 (+0,06%)
guard_serial_dev	0,17%	0,00%	49690 (+0,12%)	22841 (+0,03%)
guard_all	28,38%	0,00%	50810 (+2,38%)	23052 (+0,95%)

Tabelle 7.6: Untersuchung der Möglichkeit den *GuardPointer* in verschiedenen Treiberschichten einzubringen, wobei jeweils Fehlererkennungsrate (FDR), Speicher- und Instruktionsoverhead dargestellt sind.

Test *tty2*	**FDR**	**DRIF**	**Größe** *Byte*	**Laufzeit** *Instr.*
guard_tty_io	2,34%	0,00%	54590 (+0,48%)	87900 (+0,03%)
guard_serial_io	2,32%	0,00%	54458 (+0,24%)	87896 (+0,03%)
guard_serial_dev	0,01%	0,00%	54398 (+0,13%)	87877 (+0,01%)
guard_all	32,84%	0,00%	55518 (+2,19%)	88125 (+0,29%)

Tabelle 7.7: Untersuchung der Möglichkeit den *GuardPointer* in verschiedenen Treiberschichten einzubringen, wobei jeweils Fehlererkennungsrate (FDR), Speicher- und Instruktionsoverhead dargestellt sind.

da hier so gut wie keine Einbringung der Zeigerprüfung möglich war. Auch wenn die Erkennungsraten sehr niedrig ausfallen, so entstehen durch die zusätzliche Einbringung jedoch auch kaum Kosten in Bezug auf Performanz und Speicheroverhead, skalieren also mit der Erkennungsrate. Zudem arbeitet die Fehlererkennung in den Treiberimplementierungen überschneidungsfrei, d. h., dass die Fehlererkennungsraten, die sich aus allen Einzelprüfungen in den Treibern zusammen mit der Standardprüfung der `GuardPointer` aus dem vorigen Abschnitt ergeben, summieren, wie beim Test `guard_all` zu sehen ist.

Test *driver4*	**FDR**	**DRIF**	**Größe** *Byte*	**Laufzeit** *Instr.*
guard_tty_io	1,57%	0,00%	57754 (+0,45%)	74767 (+0,02%)
guard_serial_io	1,52%	0,00%	57626 (+0,22%)	75022 (+0,42%)
guard_loop	0,27%	0,00%	57566 (+0,12%)	74721 (+0,02%)
guard_all	37,69%	0,01%	58718 (+2,12%)	76024 (+1,76%)

Tabelle 7.8: Untersuchung der Möglichkeit den *GuardPointer* in verschiedenen Treiberschichten einzubringen, wobei jeweils Fehlererkennungsrate (FDR), Speicher- und Instruktionsoverhead dargestellt sind.

7.2.4 Untersuchung der Akzessoren

Neben den *GuardPointern* gehören die Akzessoren aus Abschnitt 5.2.3 zu den Mechanismen, die zur Fehlererkennung in der Treiberschnittstelle verwendet werden. Wie bei der Untersuchung der *GuardPointer* soll hier ebenfalls der Einfluss verschiedener Fehlererkennungskomponenten auf die Fehlererkennungsrate, außerdem die Einbringung in verschiedene Treiberschichten untersucht werden.

7.2.4.1 Einfluss verschiedener Fehlererkennungskomponenten

Im Gegensatz zu den *GuardPointern* lassen sich die Fehlererkennungsmechanismen der Zugriffsfunktionen in vielerlei Hinsicht konfigurieren. Insgesamt wurden hier die Einflüsse von `handlerCheck()` (`hc`), `nameCheck()` (`nc`), `lookup_functionCheck()` (`lc`) und die verschiedenen Funktionsüberprüfungen (`fc`) untersucht.

Test *driver3*	**FDR**	**DRIF**	**Größe** *Byte*	**Laufzeit** *Instr.*
getter deakt.	-	-	49626 (-0,01%)	22834 (+0,00%)
getter hc	17,37%	0,00%	49754 (+0,25%)	22858 (+0,11%)
getter hc,nc	22,86%	0,16%	49886 (+0,52%)	23535 (+3,07%)
getter hc,nc,lc	26,62%	0,16%	49918 (+0,58%)	23545 (+3,11%)
getter hc,nc,lc,fc	53,87%	0,16%	50078 (+0,90%)	23569 (+3,22%)
getter hc,lc,fc	47,71%	0,00%	49914 (+0,57%)	22892 (+0,25%)

Tabelle 7.9: Ergebnisse der Messung zur Untersuchung des Einflusses verschiedener Fehlererkennungskomponenten der Akzessoren mit Hilfe des Tests `driver3`.

Die maximale Fehlererkennungsrate der Akzessoren bei allen aktivierten Prüfungstypen, erreicht dabei Test `driver3` mit $53,87\%$, zu sehen in Tabelle 7.9. Auch wenn dieses Ergebnis unter dem Maximalwert der *GuardPointer* liegt, so werden hier einerseits kaum inaktive Fehler detektiert, was unnötige Fehlerbehandlungen vermeiden kann, andererseits schneiden die Akzessoren in Bezug auf den Speicherplatzverbrauch besser ab.

Den größten Zuwachs der Erkennungsrate bei den einzelnen Konfigurationen erhält die Zugriffsfunktion durch die Funktionsprüfung (`fc`), ähnlich wie beim `Guard Pointer` , nur das hier nur die aufrufende Funktion geprüft wird, was sich vor allem in der Laufzeitperformanz bemerkbar macht.

Im letzen Test `getter hc,lc,fc`, wurde eine Variante ohne Treibernamenüberprüfung getestet, da diese den durchschnittlich geringsten Anteil an der Erkennungsrate, dafür aber die höchsten Kosten, vor allem in Bezug auf die Laufzeit, verursacht. Als Alternative zur momentanen Implementierung der Namensprüfung, würde sich anbieten diesen etwa durch eine Prüfsumme zu schützen.

Insgesamt sind die Akzessoren, vor allem was Speicherplatzverbrauch angeht, performanter als die *GuardPointer* und können bei Verzicht oder Optimierung der Treibernamenprüfung auch im Bereich der Laufzeitperformanz besser abschneiden. Lediglich

Test *tty2*	**FDR**	**DRIF**	**Größe** *Byte*	**Laufzeit** *Instr.*
getter deakt.	-	-	54302 (-0,05%)	87870 (+0,00%)
getter hc	19,92%	0,00%	54462 (+0,24%)	87902 (+0,04%)
getter hc,nc	20,50%	0,10%	54586 (+0,47%)	88552 (+0,78%)
getter hc,nc,lc	22,35%	0,10%	54618 (+0,53%)	88562 (+0,79%)
getter hc,nc,lc,fc	53,22%	0,10%	54778 (+0,82%)	88594 (+0,82%)
getter hc,lc,fc	52,96%	0,00%	54622 (+0,54%)	87944 (+0,08%)

Tabelle 7.10: Ergebnisse der Messung zur Untersuchung des Einflusses verschiedener Fehlererkennungskomponenten der Akzessoren mit Hilfe des Tests `tty2`.

Test *driver4*	**FDR**	**DRIF**	**Größe** *Byte*	**Laufzeit** *Instr.*
getter deakt.	-	-	57498 (+0,00%)	74707 (+0,00%)
getter hc	23,46%	0,01%	57626 (+0,22%)	74919 (+0,28%)
getter hc,nc	27,78%	0,76%	57754 (+0,45%)	76629 (+2,57%)
getter hc,nc,lc	31,05%	0,76%	57786 (+0,50%)	76649 (+2,60%)
getter hc,nc,lc,fc	51,44%	0,76%	57946 (+0,78%)	76861 (+2,88%)
getter hc,lc,fc	50,26%	0,01%	57818 (+0,56%)	75151 (+0,59%)

Tabelle 7.11: Ergebnisse der Messung zur Untersuchung des Einflusses verschiedener Fehlererkennungskomponenten der Akzessoren mit Hilfe des Tests `driver4`.

bei der maximalen Fehlererkennungsrate und bei den nötigen Strukturänderungen zur Einbringung schneiden diese schlechter ab.

7.2.4.2 Einbringung der Fehlererkennung in Treiberschichten

Wie bei den *GuardPointern* soll hier eine Einbringung der Fehlererkennung in die Treiberschichten bzw. die Gerätetreiberimplementierung untersucht werden.

Test *driver3*	**FDR**	**DRIF**	**Größe** *Byte*	**Laufzeit** *Instr.*
getter_tty_io	6,80%	0,60%	49822 (+0,39%)	22856 (+0,10%)
getter_serial_io	5,22%	0,00%	49754 (+0,25%)	22844 (+0,04%)
getter_serial_dev	1,51%	0,00%	49662 (+0,06%)	22839 (+0,02%)
getter_all	67,65%	0,82%	50462 (+1,68%)	23606 (+3,38%)

Tabelle 7.12: Untersuchung der Fehlererkennungsrate bei Einbringung der Akzessoren in verschiedene Treiberschichten für Test `driver3`.

Test *tty2*	**FDR**	**DRIF**	**Größe** *Byte*	**Laufzeit** *Instr.*
getter_tty_io	7,75%	0,06%	54522 (+0,35%)	87892 (+0,03%)
getter_serial_io	7,54%	0,09%	54430 (+0,18%)	87888 (+0,02%)
getter_serial_dev	0,21%	0,00%	54362 (+0,06%)	87875 (+0,01%)
getter_all	69,48%	0,29%	55162 (+1,53%)	88639 (+0,88%)

Tabelle 7.13: Untersuchung der Fehlererkennungsrate bei Einbringung der Akzessoren in verschiedene Treiberschichten für Test `tty2`.

Bei den Zugriffsfunktionen können, durch die Einbringung in die Treiberschichten, bedeutend höhere Fehlerdetektionsraten erzielt werden als bei den *GuardPointern*. Sowohl in der Terminaltreiberimplementierung, als auch der Treiberimplementierung der seriellen Schnittstelle sind die Zuwächse bei der Erkennungsrate mehr als doppelt so hoch und erreichen bis zu neun Prozent.

Test *driver4*	**FDR**	**DRIF**	**Größe** *Byte*	**Laufzeit** *Instr.*
getter_tty_io	9,06%	0,17%	57694 (+0,34%)	74750 (+0,06%)
getter_serial_io	8,78%	0,00%	57630 (+0,23%)	74932 (+0,30%)
getter_serial_dev	0,88%	0,00%	57534 (+0,06%)	74717 (+0,01%)
getter_all	71,77%	0,97%	58366 (+1,51%)	77139 (+3,26%)

Tabelle 7.14: Untersuchung der Fehlererkennungsrate bei Einbringung der Akzessoren in verschiedene Treiberschichten für Test `driver4`.

Der Grund für die viel höheren Detektionsraten bei den Akzessoren, im Gegensatz zu den *GuardPointern*, liegt vor allem im Schutz der treiberspezifischen Daten durch `getPriv()`.

Zu bedenken ist dabei, dass die Treiber eigentlich nicht mehr zur Schnittstelle gehören und dort eingebrachte Fehlertoleranzmechanismen, neu hinzugefügte Treiber nicht schützen würden. Die Einbringung kann daher eher als Treiberhärtung, der bereits existierenden Treiber, angesehen werden.

7.2.5 Evaluation der Fehlererbehebung

Neben der Fehlererkennung wurde überdies eine Fehlerbehandlung in Abschnitt 5.3 implementiert. Die Ergebnisse, vor allem die Rate erfolgreicher Wiederherstellungsversuche, sowohl für *GuardPointer* als auch Akzessoren, sollen hier näher betrachtet werden.

Als Testszenarien wurde hier, die in Abschnitt 7.2.3.1 bereits im Zuge der Fehlererkennung untersuchte Standardprüfung des *GuardPointers*, sowie die Zugriffsfunktion, mit allen aktivierten Prüfungstypen, sowie die Varianten in denen Prüfungen zusätzlich in die Treiberimplementierungen eingebracht wurden, genommen.

Test *driver4*	**FDR**	**WR**	**Reboot**	**Größe** *Byte*	**Laufzeit** *Instr.*
guard	25,37%	93,38%	0,00%	52570 (+ 5,92%)	23006 (+0,76%)
guard_all	30,20%	76,66%	0,00%	54718 (+10,25%)	23065 (+1,02%)
getter_ac	53,52%	78,17%	0,00%	50458 (+ 1,68%)	23593 (+3,32%)
getter_all	67,11%	82,92%	20,96%	50906 (+ 2,58%)	23647 (+3,56%)

Tabelle 7.15: Auswertung der Ergebnisse zur Fehlerbehebung hinsichtlich Fehlererkennungsrate (FDR), erfolgreicher Wiederherstellungsrate (WR) sowie dem Anteil dieser durch Neustart, mittels Testprogramm `driver3`.

Die Variante des *GuardPointers* mit aktivierter zyklischer Funktionsprüfung ist hier nicht dargestellt, da diese durch die hohe Anzahl an detektierter, inaktiver Fehler eine vermeintlich hohe Fehlererkennungsrate aufweist. Hierbei kommt der Nachteil, beim Aufspüren inaktiver Fehler zum Vorschein, da diese bei der Fehlerbehebung, wie gewöhnliche Fehler behandelt werden und so bspw. einen Neustart des Systems verursachen können, obwohl dieser inaktive Fehler nie aktiviert werden würde.

Test *tty2*	**FDR**	**WR**	**Reboot**	**Größe** *Byte*	**Laufzeit** *Instr.*
guard	28,46%	96,03%	0,00%	57278 (+5,43%)	88069 (+0,23%)
guard_all	32,52%	80,02%	0,00%	59418 (+9,36%)	88140 (+0,31%)
getter_ac	54,25%	95,91%	0,00%	55166 (+1,59%)	88626 (+0,86%)
getter_all	68,46%	97,31%	23,20%	55614 (+2,42%)	88687 (+0,93%)

Tabelle 7.16: Auswertung der Ergebnisse zur Fehlerbehebung hinsichtlich Fehlererkennungsrate (FDR), erfolgreicher Wiederherstellungsrate (WR) sowie dem Anteil dieser durch Neustart, mittels Testprogramm `tty2`.

Für die Wiederherstellungsrate (WR) gilt, dass die Wiederherstellung dann als erfolgreich betrachtet wird, wenn ein Aufruf einer Funktion der *User API* aus dem Testprogramm heraus mit einem speziellen, für diesen Test definierten Fehlercode zurückkehrt (vgl. dazu Abbildung 5.5).

Da die hier aufgeführten Tests kaum inaktive Fehler aufspüren, entspricht die Fehlerdetektionsrate dem der Ergebnisse zur jeweiligen Fehlererkennung aus den vorigen Abschnitten. Die Wiederherstellungsrate liegt bei der Standardversion der *GuardPointer* in allen Tests stets über 93%, während diese mit Zunahme der Treiberschichteneinbringung (`guard_all`) vor allem in Tabelle 7.15 bis auf 76% einbricht. Grund hierfür ist vor allem der Zugriff auf treiberspezifische Datenstrukturen, die etwa in `cyg_devtab_entry` im `void`-Zeiger `priv` angelegt werden. Da diese Daten nicht durch das Fehlergerät ersetzt oder nachgeahmt werden können, da keine Informationen über die Struktur vorhanden sind, eine Rückgabe eines Wertes aber zwingend erfolgen muss, können Fehler an

Test *driver4*	**FDR**	**WR**	**Reboot**	**Größe** *Byte*	**Laufzeit** *Instr.*
guard	32,38%	93,19%	0,00%	58942 (+2,50%)	75988 (+1,72%)
guard_all	35,68%	84,93%	0,00%	59678 (+3,78%)	76484 (+2,38%)
getter_ac	52,87%	84,64%	0,00%	58334 (+1,45%)	77073 (+3,17%)
getter_all	71,48%	88,13%	18,46%	58814 (+2,29%)	77364 (+3,56%)

Tabelle 7.17: Auswertung der Ergebnisse zur Fehlerbehebung hinsichtlich Fehlererkennungsrate (FDR), erfolgreicher Wiederherstellungsrate (WR) sowie dem Anteil dieser durch Neustart, mittels Testprogramm `driver4`.

dieser Stelle nicht behoben werden. Dieses Problem wird bei den Zugriffsfunktionen in `getPriv_functionCheck()` umgangen, in dem ein Neustart des Betriebssystems, wegen eines nicht korrigierbaren Fehlers, ausgeführt wird.

Insgesamt schneiden die Akzessoren, sowohl bei der Höhe der Fehlererkennungsrate in den Treiberimplementierungen, die bis zu $97,31\%$ in Tabelle 7.16 erreicht, als auch bei der Wiederherstellung nach auftretenden Fehlern in diesen, besser ab.

Deutlich zu sehen ist, dass der Speicherplatzverbrauch, sowie der Laufzeitoverhead gegenüber den Varianten ohne Fehlerbehandlung deutlich zugenommen hat. Dabei sind 64 Byte des Größenzuwaches im Datensegment durch das Fehlergerät (vgl. 5.3.2) bedingt, während der Rest des gesteigerten Speicherplatzverbrauchs, wie bei den anderen Tests, den Codebereich betrifft. Auch in diesem Fall spielt wieder die Zusammensetzung der Testprogramme eine Rolle.

7.2.6 Kombinierte Lösungen

Zum Schluss sollen Varianten, die sowohl Akzessoren als auch *GuardPointer* enthalten getestet werden, um wenn möglich die Vorteile beider Verfahren zu nutzen und eine maximale Fehlererkennung zu ermöglichen. Dabei arbeiten beide Varianten nicht überschneidungsfrei, da sie bspw. bei der Funktionsüberprüfung in der Gerätetreiberfunktionstabelle, denselben Ansatz verfolgen.

Bei den beiden Tests `Kombination1` und `Kombination2` sind jeweils die Akzessoren mit allen eingeschalteten Prüfungen sowie die *GuardPointer* mit ausgeschalteter zyklischer Funktionsprüfung (CFC) aktiv. Die Abschaltung der CFC hat dabei keinen Einfluss auf die Erkennungsrate bei den Kombinationen, sondern lediglich auf das Aufspüren inaktiver Fehler. In `Kombination2` sind im Gegensatz zum Test `Kombination1` zudem die Erkennungsmechanismen in die Gerätetreiberimplementierungen, wie schon bei den Einzeltests in Abschnitt 7.2.3.2 und Abschnitt 7.2.4.2, eingebracht worden.

Wie man in Tabelle 7.18 sieht, können die Fehlererkennungsraten gegenüber den Einzelimplementierungen noch einmal gesteigert werden, auch wenn trotz abgeschalteter zyklischer Funktionsprüfung etwa 20% bis 30% Überschneidungen auftreten, was natürlich zu Verlust bei der Performanz bzw. einem höheren Overhead führt. Weitergehende

Test	FDR	DRIF	Größe *Byte*	Laufzeit *Instr.*
driver3 Kombination1	59,51%	0,17%	50650 (+2,06%)	23706 (+3,82%)
driver3 Kombination2	75,59%	0,84%	51482 (+3,73%)	23791 (+4,19%)
tty2 Kombination1	59,93%	0,10%	55358 (+1,89%)	88749 (+1,00%)
tty2 Kombination2	78,08%	0,29%	56190 (+3,42%)	88852 (+1,12%)
driver4 Kombination1	60,92%	0,77%	58558 (+1,84%)	77524 (+3,77%)
driver4 Kombination1	79,89%	1,10%	59358 (+3,23%)	78145 (+4,60%)

Tabelle 7.18: Ergebnisse der Untersuchung von der Kombination von Akzessoren und *GuardPointern*.

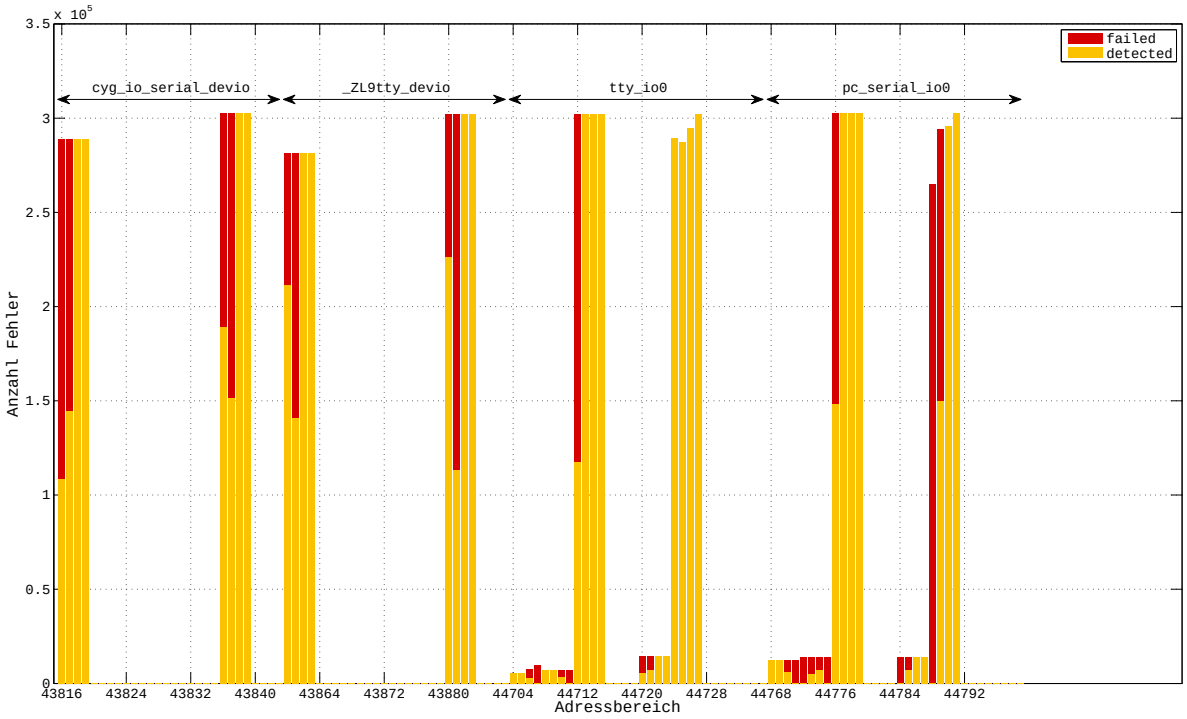


Abbildung 7.4: Ortsverteilung von Fehler und Detektion von Test `tty2 Kombination2`.

Optimierungen, um diese Überschneidungen zu eleminieren, wären aber durchaus möglich. Hier soll lediglich gezeigt werden, welche Fehlererkennungsraten möglich sind.

Zum Abschluss ist noch einmal die Ortsverteilung der Fehler in Abbildung 7.4, wie sind in der Einführungsuntersuchung gegeben war, aber nun mit den detektierten Fehlern von `tty2 Kombination2` dargestellt, sowie die erkannten Fehler nach Fehlerart in Abbildung 7.5 noch einmal aufgeführt.

Zu sehen ist, dass so gut wie alle Instruktionszeigerfehler zudem ein Großteil der aufgetretenen Speicherzugriffsverletzungen, durch die Bereichsprüfungen erkannt wurden.

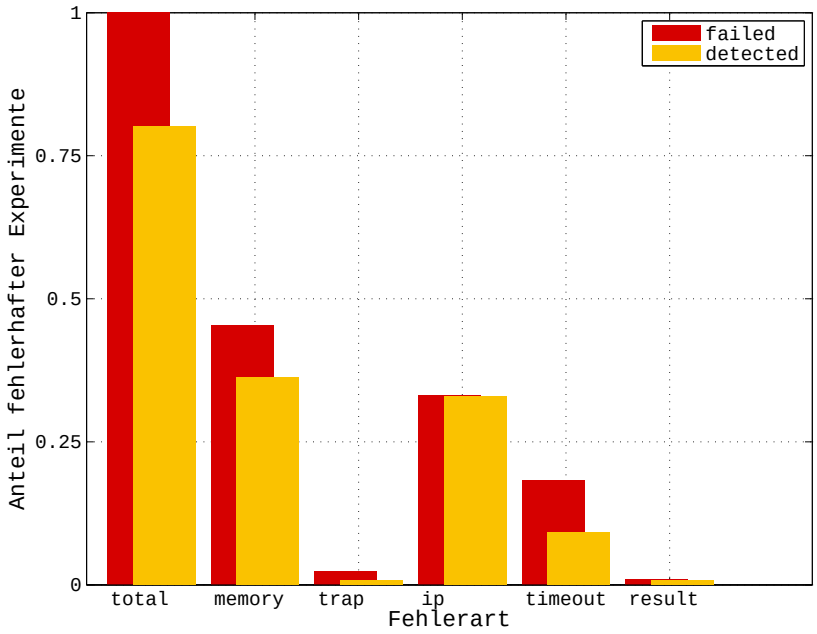


Abbildung 7.5: Verteilung von Fehler und Detektion nach Fehlertyp von Test `tty2 Kombination2`.

Bei der Ortsverteilung der Fehler, bei dem jeder Balken ein Byte darstellt, ist zu sehen, dass die Fehlererkennung bei den Bytes für einen Zeiger mit den niederwertigeren Bits, also in Abbildung 7.4 von rechts nach links[5], abnimmt. Dies entspricht dem typischen Bild einer Bereichsprüfung, auf das später näher in Abschnitt 7.3 eingegangen wird.

7.2.7 Zusammenfassung

Die hier zum Test verwendeten „gestapelten“ Treiber stellen einen durchaus typischen Anwendungsfall dar. Um die Ergebnisse zu verfeinern, wären weitere Tests mit anderen Treibertypen in Erwägung zu ziehen.

Insgesamt lassen sich die Akzessoren sowie die *GuardPointer*, für die Absicherung der Treiberschnittstelle erfolgreich einsetzen. Die hohen Fehlererkennungsraten, die bei Kombination beider Verfahren annähernd 80% erreichen, bei nur geringem Overhead von Speicherplatz und Laufzeit von wenigen Prozent sind natürlich auch der Speichereinteilung von *eCos*, sowie der Implementierung der Schnittstelle zu verdanken.

Die Bereichsprüfung kann hier deshalb so effizient arbeiten, weil die Datenstrukturen bekannt sind und sich deren Lokalität im Speicher stark, für `cyg_devtan_entry` sogar auf etwas mehr als hundert Byte, eingrenzen lässt. Trotzdem lassen sich die *GuardPointer* auch in anderen Anwendungen einfach und effizient einsetzen und erweitern, oder mit anderen Prüfverfahren bestücken. Gerade die Prüfung vor der kritischen Dereferenzierung kann helfen, einige schwere Folgen durch Bitfehler zu vermeiden.

[5]Little-Endian System

7.3 Abschätzung der Fehlererkennungsrate bei der Bereichsprüfung

Neben der experimentellen Bestimmung der Fehlererkennungsrate gibt es zusätzlich die Möglichkeit diese durch ein mathematisches Modell abzuschätzen. Auch wenn solche Modelle, bei steigender Komplexität des Problems, in der Praxis oft nur schwer zu erstellen und handhabbar sind, soll das an dieser Stelle für eine Abschätzung der in den Experimenten erwarteten Fehlerdetektionsrate versucht werden.

Dabei interessiert vor allem, dass in Abschnitt 3.2.1.1 beschriebene und in dieser Arbeit hauptsächlich eingesetzte Fehlererkennungsverfahren der Bereichsprüfung, bei dem die Fehlerdetektionsrate für eine zu überwachende Variable fester Bitlänge, die nur Werte aus einem bestimmten, vorgegebenen Bereich annehmen darf, zu bestimmen ist. Die gesuchte Abschätzung soll dabei unabhängig vom genauen Wert der Variable sein, sondern nur von der Größe des Gültigkeitsbereichs abhängen.

Beispiel für eine solche zu überwachende Variable wäre ein Funktionszeiger, der lediglich in den Text-Bereich des Speichers zeigen darf und für den die Fehlererkennungsrate den Anteil ungültiger Zeiger, nach Durchlaufen aller möglichen Bit-Flips darstellt, d. h. derjenigen, die in einen Bereich außerhalb des Textsegments zeigen. Dabei ist z. B. die Größe des zu überwachenden Bereichs bekannt, nicht jedoch seine genaue endgültige Lage im Speicher.

Insgesamt liegen folgende Gegebenheiten und Vereinfachungen vor:

- Gegeben ist ein Bit-String X fester Länge n mit $X = x_{n-1}...x_0$ und $x \in \{0, 1\}$, wobei X den Wert $value(X) = \sum_{i=0}^{n-1} 2^i x_i$ repräsentiert.
- Des Weiteren ist die Größe des Gültigkeitsbereichs $r = b - a + 1$ gegeben, mit den oberen und unteren Grenzen $a, b \in \mathbb{N}_0$ mit $a, b \in [0, 2^n - 1]$ und $a < b$, wobei die spätere Formel nur von r abhängig sein soll.
- Wir beschränken uns hier nur auf Single-Bit-Flips, d. h. $X_{BF} = x_{n-1}...x_{i-1}1x_{i+1}...x$ falls $x_i = 0$ ist bzw. $X_{BF} = x_{n-1}...x_{i-1}0x_{i+1}...x$ falls $x_i = 1$ mit $i \in [0, n - 1]$ entspricht dem Bit-String nach dem Bitkipper. Die Wahrscheinlichkeit eines solchen Bit-Flips soll dabei gleichverteilt und für jede Bitposition $\frac{1}{2}$ betragen.
- Gesucht ist die Fehlererkennungsrate FDR, die für alle möglichen Single-Bitkipper im Bit-String X den Anteil an Bit-Flips ermittelt die dazu führt, dass $value(X_{BF}) < a$ oder $value(X_{BF}) > b$ ist, wobei inital $a \leq value(X_{BF}) \leq b$ ist.

Im Beispiel in Abbildung 7.6 sind zwei Möglichkeiten eines Intervalls für eine Gültigkeitsbereichsgröße von $r = 4$ gegeben. Die bei der Bit-Stringrepräsentation mit rot markierten Bits, würden bei einem Bit-Flip an dieser Position einen Unter- oder Überlauf verursachen. Dabei ist zu sehen, dass die Lage des Intervalls bei der Fehlererkennungsrate eine entscheidende Rolle spielen kann. Bei größeren Intervallen z. B. dem Textsegment bei *eCos*, das selbst bei kleinen Programmen im zweistelligen Kilobereich liegt, kann also die Lage wie später noch zu sehen eine Rolle spielen.

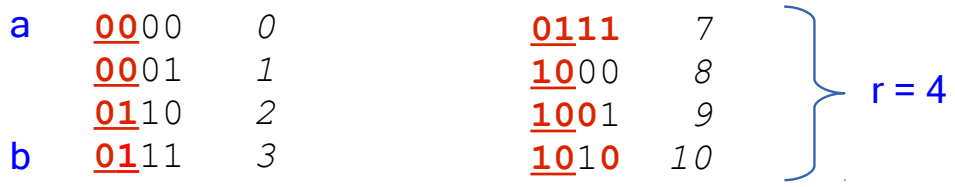


Abbildung 7.6: Beispiel zur Abschätzung der Fehlererkennungsrate bei der Bereichsprüfung für unterschiedliche Ausprägungen der selben Bereichsgröße. Während die unterstrichenden Bits die Worst-Case-Erkennungsrate repräsentieren, würde ein Bit-Flip bei allen rot markierten Bits einen Unter- bzw. Überlauf verursachen und so erkannt werden.

7.3.1 Abschätzung der Worst-Case Fehlererkennungsrate

Die Worst-Case Fehlererkennungsrate liegt dann vor, wenn $a = 0$ bzw. das erste Element eine Zweierpotenz ist und r eine Zweierpotenz ist. In diesem Fall würden Bitkipper in allen nicht verwendeten Bits zu einer Bereichsüber- oder Unterschreitung führen. Dann entspricht die Fehlererkennungsrate dem Anteil, der nicht durch r belegten Bits und so genau Formel 7.1.

$$FDR_{wc} = 1 - \frac{\lceil \log_2 (b - a + 1) \rceil}{n} = 1 - \frac{\lceil \log_2 (r) \rceil}{n} \tag{7.1}$$

In Abbildung 7.6 ist beim linken Intervall ein solcher Worst-Case-Fall exemplarisch dargestellt. Während die beiden LSB[6]s für jede dieser Zahlen aus dem zu schützenden Intervall, einzeln beliebig änderbar sind, ohne einen Unter- oder Überlauf zu erzeugen, erzeugt bspw. die Bitrepräsentation der Zahl 7 im rechten Intervall für jeden Bit-Flip einen Fehler.

7.3.2 Abschätzung der durchschnittlichen Fehlererkennungsrate

Um die durchschnittliche Erkennungsrate abzuschätzen, muss für jede Bitposition die Wahrscheinlichkeit das ein Single-Bit-Flip einen Unter- oder Überlauf verursacht ermittelt werden. Dies entspricht der in Formel 7.2 dargestellten Wahrscheinlichkeit, wobei hier je nach Wert von x_i zwei Möglichkeiten unterschieden werden.

$$FDR = \frac{1}{n} \sum_{i=0}^{n-1} P(value(X) + 2^i > b) \cdot P(x_i = 0) + P(value(X) - 2^i < a) \cdot P(x_i = 0) \tag{7.2}$$

Falls $x_i = 0$ und so ein Bit-Flip von 0 nach 1 den Wert von X um 2^i erhöhen würde, ist die Wahrscheinlichkeit dafür das X_{BF} größer als die obere Schranke b wird ausschlaggebend, während beim umgekehrten Fall die Wahrscheinlichkeit für einen Unterlauf betrachtet wird.

[6]Least Significant Bit

Ausformuliert ergibt sich die geschlossene Formel 7.3.

$$\begin{aligned} FDR = \frac{1}{n}\sum_{i=1}^{n}\Bigg(p_{of}\cdot\sum_{k=0}^{2^{i-1}-1}\left(\left\lfloor\frac{b+1+k}{2^i}\right\rfloor-\left\lfloor\frac{a+k}{2^i}\right\rfloor\right) \\ + p_{uf}\cdot\left(1-\sum_{k=0}^{2^{i-1}-1}\left(\left\lfloor\frac{b+1+k}{2^i}\right\rfloor-\left\lfloor\frac{a+k}{2^i}\right\rfloor\right)\right)\Bigg) \end{aligned} \tag{7.3}$$

$$\text{mit } p_{of} = \begin{cases} \frac{2^{i-1}}{b-a+1} & \text{für } 2^{i-1}+a \le b \\ 1 & \text{sonst} \end{cases}, \quad p_{uf} = \begin{cases} \frac{b-2^{i-1}-a-1}{b-a+1} & \text{für } b-2^{i-1} \ge a \\ 0 & \text{sonst} \end{cases}$$

Da die vollständige Formel 7.3 durch die exponentiell langen Summenterme zur Bestimmung der Wahrscheinlichkeit ob $x_i = 1$ bzw. $x_i = 0$, schwer zu handhaben ist und dies in der Konsequenz, für größere Werte von i die Berechnungsdauer strapazieren würde, wird im Folgenden eine vereinfachte Abschätzung der Fehlererkennungsrate gegeben, deren Ergebnisse sich nur marginal und in vernachlässigbarer Höhe unterscheiden.

$$\begin{aligned} FDR &\ge \frac{1}{n}\sum_{i=1}^{n}\Bigg(p_{of}\cdot 2^{i-1}\left(\left\lfloor\frac{b+1+k}{2^i}\right\rfloor-\left\lfloor\frac{a+k}{2^i}\right\rfloor\right) \\ &\quad + p_{uf}\cdot\left(1-2^{i-1}\left(\left\lfloor\frac{b+1+k}{2^i}\right\rfloor-\left\lfloor\frac{a+k}{2^i}\right\rfloor\right)\right)\Bigg) \\ &\ge \frac{1}{n}\sum_{i=1}^{n}\Bigg(p_{of}\cdot 2^{i-1}\left(\left\lfloor\frac{b-a+1}{2^i}\right\rfloor-1\right) \\ &\quad + p_{uf}\cdot\left(1-2^{i-1}\left(\left\lfloor\frac{b-a+1}{2^i}\right\rfloor-1\right)\right)\Bigg) \\ &= \frac{1}{n}\sum_{i=1}^{n}\Bigg(\frac{2^{i-1}}{r}\cdot\frac{2^{i-1}}{r}\left(\left\lfloor\frac{r}{2^i}\right\rfloor-1\right) \\ &\quad + \left(1-\frac{r-2^{i-1}-2}{r}\right)\cdot\left(1-\frac{2^{i-1}}{r}\left(\left\lfloor\frac{r}{2^i}\right\rfloor-1\right)\right)\Bigg) \end{aligned} \tag{7.4}$$

Eine exemplarische Veranschaulichung der Fehlerdetektionsrate in Abhängigkeit der Intervallgröße, sowie ein Vergleich mit der Worst-Case Fehlererkennungsrate ist in Abbildung 7.7 zu sehen.

7.3.3 Bewertung der Abschätzung

Um die Güte der Abschätzung von Formel 7.4 zu ermitteln, wurden eine Reihe zufällig gewählter, experimentell bestimmter Fehlerdetektionsraten, von denen einige in Abbildung 7.8 eingezeichnet sind, mit denen der Abschätzung verglichen. Auch wenn eine Vielzahl der errechneten Werte von Formel 7.4 sehr nah an die realen Werte heranreichen, können in Extremfällen einige Ergebnisse sehr stark abweichen. Diese Abweichung kann, wie bei den eingekreisten Werten mehr als 10% betragen. Auch wenn die Lage des Intervalls für größere Werte im Mittel keinen hohen Einfluss auf die Erkennungsrate hat, kann es bei ungünstigen Werten trotzdem zum Worst-Case kommen.

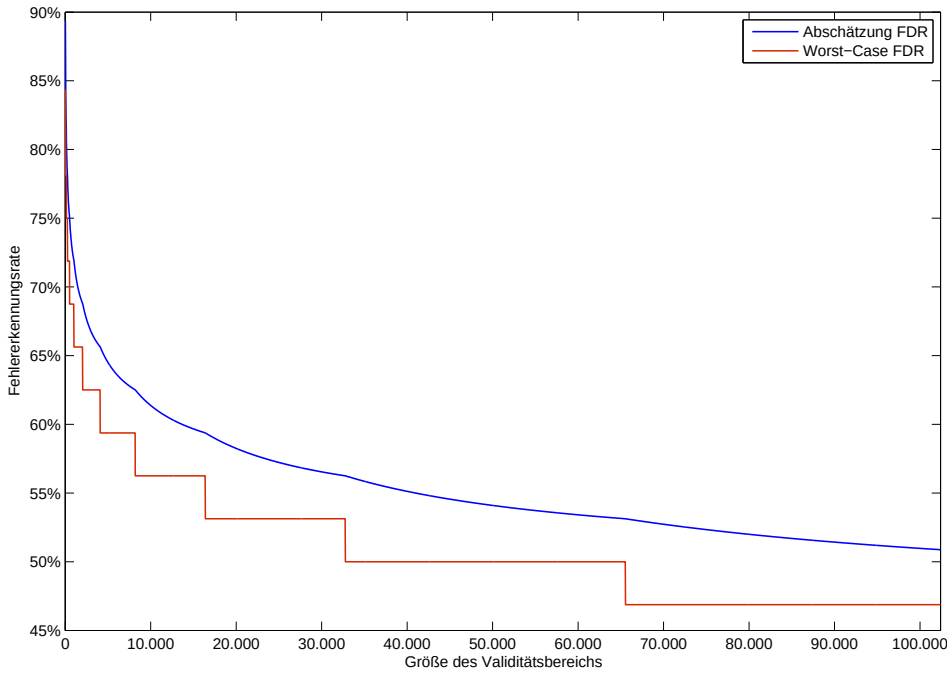


Abbildung 7.7: Abschätzung der Bereichsprüfung und Worst-Case Erkennungsrate in Abhängigkeit von der Intervallgröße für $n = 32$.

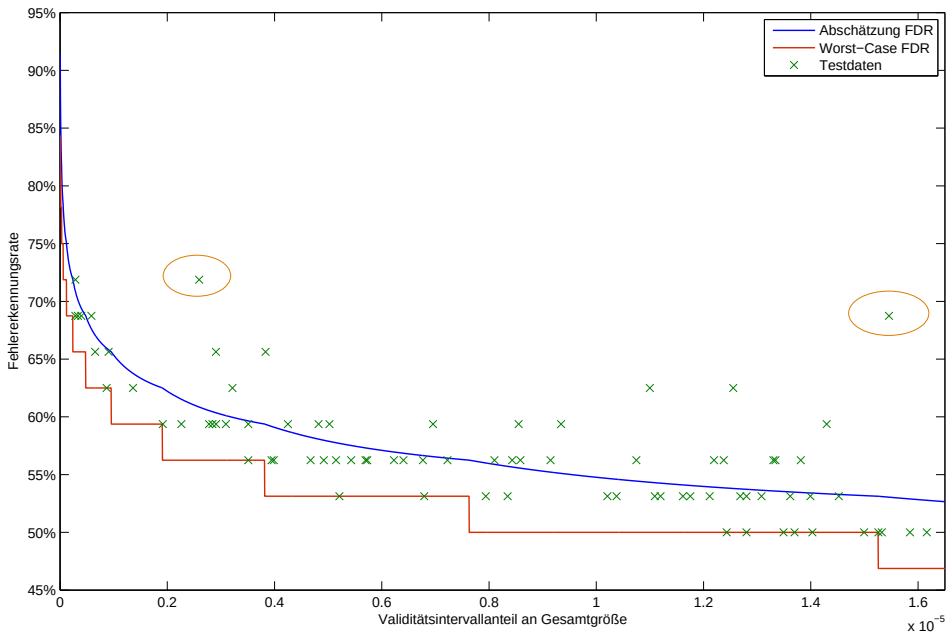


Abbildung 7.8: Vergleich von Abschätzung und experimentell bestimmten Werten in Bezug zum Anteil der Gültigkeitsintervall an der Gesamtgröße für $n = 32$ zur Bestimmung der Güte.

FDR_{WC}	**Validitätsintervallgröße**	**Anteil an Gesamtgröße**
$\geq 80\%$	≤ 64	$1,4901 \cdot 10^{-8}$
$\geq 70\%$	≤ 512	$1,1921 \cdot 10^{-7}$
$\geq 60\%$	≤ 4096	$9,5367 \cdot 10^{-7}$
$\geq 50\%$	≤ 65536	$1,5259 \cdot 10^{-5}$
$\geq 40\%$	≤ 524288	$1,2207 \cdot 10^{-4}$

Tabelle 7.19: Worst-Case Detektionsraten und zugehörige maximale Validitätsintervallgrößen zur Veranschaulichung praktikabler Werte für $n = 32$.

Während für eine einzelne Variable diese Abschätzung ausreichen kann, ist die kumulative Fehlererkennungsratenabweichung bei Kombination mehrerer aufeinanderfolgender Bereichsprüfungen, wie etwa bei der generischen Bereichsprüfung in Abschnitt 6 oder der Absicherung der Treiberschnittstelle in Abschnitt 5 nicht mehr akzeptabel. Ein weiterer Faktor, der die Praxistauglichkeit einer solchen mathematischen Abschätzung infrage stellt, ist die Komplexität des untersuchten Programms und damit der Schwierigkeit der Identifizierung von Stellen, an denen eine solche Validitätsprüfung stattfindet.

Zum Schluss soll mit der in Tabelle 7.19 gegebenen Nebeneinanderstellung von anvisierter bzw. minimal geforderter Fehlerdetektionsrate zur erforderlichen Validitätsintervallgröße veranschaulicht werden, dass Bereichsprüfungen keinesfalls für jegliche Fehlererkennungsaufgaben sinnvoll eingesetzt werden können. Nur wenn der gültige Bereich im Gegensatz zum Gesamtbereich sehr klein ist, sind passable Fehlererkennungsraten zu erwarten. Dabei ist zu beachten, dass die Worst-Case-Erkennungsraten von der tatsächlichen Fehlererkennungsrate stark abweichen kann und so um bis zu etwa 20% bessere Raten erzielt werden können.

Interessant ist auch zu sehen, dass eine Verschiebung des Validitätsbereichs im Einzelfall einen großen Einfluss auf die Fehlererkennungsrate haben kann. Eine solche Verschiebung bspw. des Data-Segments durch Programmänderungen und folglicher Vergrößerung des Textsegments kann unter Umständen dazu führen, dass trotz gleich großer Gültigkeitsbereiche eine schlechtere Fehlererkennungsrate erreicht wird.

7.4 Evaluation der generischen Zeigervalidierung

An dieser Stelle soll die in Abschnitt 6 beschriebene generische Zeigervalidierung empirisch untersucht werden. Dabei soll zuerst kurz auf die verwendeten Testprogramme eingegangen werden, gefolgt von den Untersuchungsergebnissen zu den Fehlererkennungsraten, sowie Ergebnissen zur Performanz der Prüfungen. Im Anschluss soll der Einfluss der einzelnen Prüfungstypen auf das Gesamtergebnis gezeigt werden.

Um einen Vergleich von einer maßgeschneiderten Prüfung mit der generischen Zeigervalidierung zu bekommen, werden außerdem die ermittelten Daten zur Erkennungsrate sowie Performanz offeriert, die bei der Einbringung der Prüfung in die im vorigen Abschnitt untersuchte Treiberschnittstelle, erhoben wurden.

7.4.1 Versuchsprogramme

Bei den Versuchsprogrammen, die hier zum Testen der generischen Zeigervalidierung benutzt werden, handelt es sich um Testapplikationen aus *eCos*, die zur Überprüfung der Kernelfunktionen dienen.

Da einige Tests von der Laufzeit her einfach zu lang sind und kein Überschuss an sehr kurzen Tests entstehen soll, in denen lediglich der Konstruktor einer Klasse getestet wird, werden letztendlich die in Tabelle 7.20 aufgeführten Kernel-Tests für die Evaluation verwendet.

Die Fehlerinjektion erfolgte dabei durch Bitflips, sowohl in der Datensektion als auch der BSS-Sektion (vgl. Abbildung 7.1). Die Fehlerrate, durch gefundene aktive Fehler, lag zwischen $0,23\%$ und $0,79\%$.

```
1  pointcut criticalFunctions() = ("");
2  pointcut criticalClasses() = derived("Cyg_SchedThread_Implementation")
3        || derived("Cyg_HardwareThread") || derived("Cyg_Scheduler_SchedLock")
4        || derived("Cyg_Alarm") || derived("Cyg_DNode_T") || derived("Cyg_Mbox%")
5        || derived("Cyg_%_Semaphore%") || derived("Cyg_Counter") || derived("Cyg_Mqueue")
6        || derived("Cyg_Condition_Variable") || derived("Cyg_RealTimeClock")
7        || derived("Cyg_CList_T") || derived("Cyg_Check_Structure_Sizes")
8        || derived("Cyg_Llistt") || derived("Cyg_Flag") || derived("Cyg_SpinLock")
9        || derived("Cyg_Exception_Control")  || derived("Cyg_Mutex");
```

Listing 7.1: Pointcut Expression für die generische Bereichsprüfung der Kernel-Tests.

Die Auswahl der Klassen aus dem *eCos*-Kernel Paket, in die an geeigneter Stelle eine generische Bereichsprüfung eingebracht wird, ist durch Listing 7.1 festgelegt und sollte den Großteil, der im Kernel-Paket deklarierten Klassen ausmachen. Die Pointcut Expressions spezifizieren hierbei die in `TemplateCheck.ah` definierten Pointcuts `guardedObjectCycleOperations()` und `guardedFunctionExecution()` (vgl. Abbildung 6.1).

7.4.2 Fehlererkennungsrate und Performanz

Für eine erste Untersuchung wird die generische Zeigervalidierung in ihrer Grundfunktionalität getestet, d. h., da die Kernel-Funktionen keine dynamische Allokation benut-

Test	**Größe** *Bytes*	**Laufzeit** *Instruktionen*	**Beschreibung**
bin_sem1	56186	6941	Tests zur Basisfunktionalität binärischer Semaphoren mittels drei Semaphoren vom Typ `Cyg_Binary_Semaphore` und zwei Threads.
bin_sem3	56574	10862	Tests zur Timeoutfunktionalität binärischer Semaphoren mittels drei Semaphoren und zwei Threads.
clock0	51386	7578	Basistests zur Uhrenfunktionalität von `Cyg_Clock`, `Cyg_Counter` und `Cyg_Alarm`.
cnt_sem0	47258	4805	Kurzer Test der Basisfunktionen dreier zählender Semaphoren (`Cyg_Counting_Semaphore`).
cnt_sem1	56218	7536	Längerer Test dreier zählender Semaphoren mit zwei Threads.
flag0	47294	3373	Konstruktor- und Destruktortest von `Cyg_Flag`.
flag1	62458	33223	Test der Basisfunktionalitäten sowie Synchronisation dreier Flags durch drei Threads.
mqueue1	61274	126261	Test einer Nachrichtenwarteschlange (`Cyg_Mqueue`) zwischen zwei Threads, sowie zwei Semaphoren zur Synchronisation.
mutex1	59162	8608	Grundlagentest für zwei Mutexe (`Cyg_Mutex`), sowie drei Monitor Objekten (`Cyg_Condition_Variable`) in drei Threads.
mutex2	64250	64694	Ausgiebiger Test der Release-Funktionalität von Monitoren und Mutexen.
release	55418	6372	Test der Release-Fähigkeit von Threads (`Cyg_Thread`).
sched1	54686	4154	Basistests des Schedulers (`Cyg_Scheduler`) mit zwei Threads.
thread0	50458	4113	Konstruktor/Destruktor Tests zweier Threads.
thread1	54910	5556	Tests zur Basisfunktionalität zweier Threads.
thread2	59930	8216	Prioritättests dreier Threads, sowie Semaphoren.

Tabelle 7.20: Zur Evaluation verwendete Testprogramme, aus den Kernel-Tests von *eCos*.

zen, mit deaktivierter Heapprüfung und ohne gesonderte Einzelobjektprüfung. Die konfigurierbaren Prüfziele, bestehend aus Argumentprüfung, That-Zeiger-Prüfung und Ergebnisprüfung, sind hierbei alle aktiviert. In Tabelle 7.21 sind die Ergebnisse der Un-

Test	**FDR**	**Total**	**ROM**	**RAM**		**Laufzeit**
			text	*data*	*bss*	
bin_sem1	11,07%	63774 (+13,51%)	35614	1024	27136	8494 (+22,37%)
bin_sem3	14,23%	64890 (+14,70%)	36730	1024	27136	13702 (+26,15%)
clock0	5,61%	59738 (+16,25%)	38746	1024	19968	8377 (+10,54%)
cnt_sem0	0,56%	54174 (+14,63%)	33150	1024	20000	4883 (+ 1,62%)
cnt_sem1	8,86%	63610 (+13,15%)	35450	1024	27136	9475 (+25,73%)
flag0	1,35%	54238 (+14,68%)	33214	1024	20000	3425 (+ 1,54%)
flag1	17,26%	71966 (+15,22%)	40222	1024	30720	39187 (+17,95%)
mqueue1	73,14%	71230 (+16,25%)	42558	1024	27648	130123 (+ 3,06%)
mutex1	19,46%	66270 (+12,01%)	34494	1024	30752	11813 (+37,23%)
mutex2	13,56%	71614 (+11,46%)	36286	1024	34304	68690 (+ 6,18%)
release	15,15%	63226 (+14,09%)	35066	1024	27136	8901 (+39,69%)
sched1	4,73%	61726 (+12,87%)	33598	1024	27104	4652 (+11,99%)
thread0	1,79%	57274 (+13,51%)	32858	1024	23392	4697 (+14,20%)
thread1	8,72%	62394 (+13,63%)	34266	1024	27104	7025 (+26,44%)
thread2	11,39%	67610 (+12,81%)	35866	1024	30720	12194 (+48,42%)

Tabelle 7.21: Untersuchung der Fehlererkennungsrate (FDR), sowie des Speicher- und Laufzeitoverheads bei der generischen Zeigervalidierung. Die Laufzeit bezeichnet dabei die Instruktionen eines kompletten Testprogrammdurchlaufs, die Größe ist in Byte angegeben.

tersuchung aufgelistet. Dabei ist sowohl die Fehlererkennungsrate (FDR) als auch der Speicher- und Laufzeitoverhead aufgeführt. Die Laufzeit wird dabei im Golden Run, also ohne die Injektion eines Fehlers, durch die Anzahl der Instruktionen bestimmt, die Größe ist in Bytes angegeben, wobei sich die Größenänderungen hier nur auf das Textsegment beziehen, da für die Bereichsprüfung keine Daten gespeichert werden müssen.

Auffällig ist, dass der Speicheroverhead, unabhängig von der Fehlererkennungsrate, für alle Tests ähnlich groß ist und zwischen $11,46\%$ und $16,25\%$ liegt, wohingegen der Instruktionsoverhead weitestgehend, mit steigender Fehlererkennungsrate, ebenfalls größer zu werden scheint.

Nicht überraschend dagegen ist, dass die Kernel-Tests cnt_sem0, flag0 und thread0 eine sehr niedrige Fehlererkennungsrate von unter 2% aufweisen, da bei diesen Tests lediglich der Konstruktor bzw. Destruktor getestet wird.

In Abbildung 7.9 sind die Fehlererkennungsraten, zum besseren visuellen Vergleich, für jeden Test graphisch dargestellt. Der ebenfalls dargestellte Mittelwert liegt bei $9,83\%$. Zusätzlich ist darunter jeweils die Erkennungsrate von inaktiven Fehlern, d. h. Fehler, die während der Laufzeit keinen Einfluss auf den Programmfluss oder das Ergebnis haben,

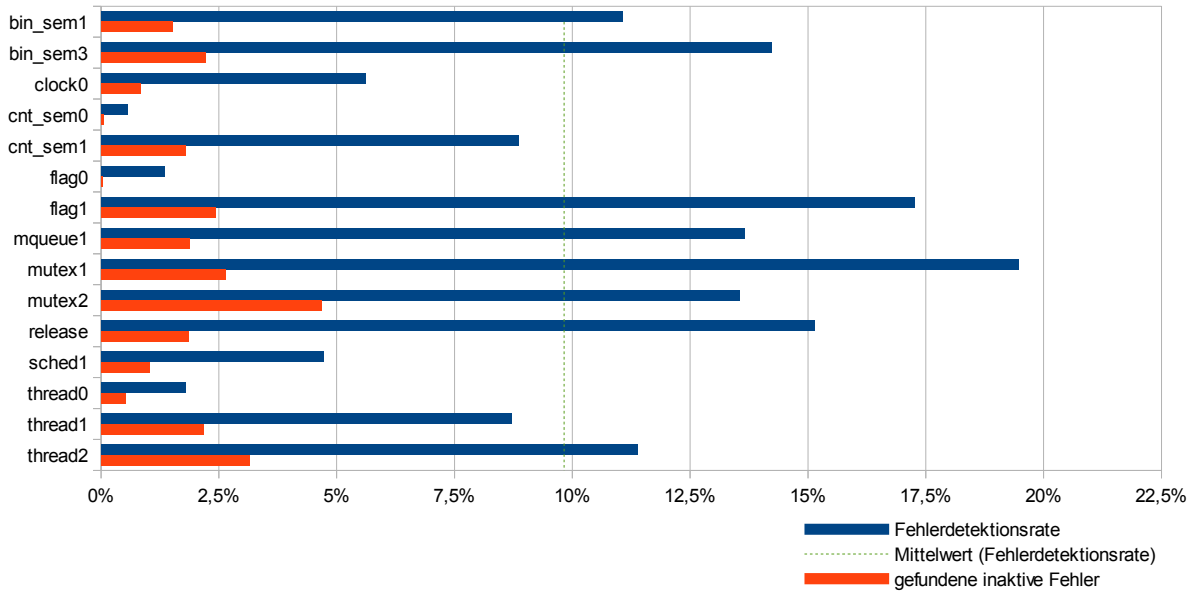


Abbildung 7.9: Fehlerdetektionsraten der generischen Zeigervalidierung für die unterschiedlichen Kernel-Tests. Zusätzlich dazu ist der Anteil der detektierten inaktiven Fehler, sowie der Mittelwert der Fehlererkennungsrate angegeben.

dargestellt.

Um die Effizienz, d. h. das Verhältnis von Ressourceneinsatz und Fehlererkennungsrate, bei den einzelnen Tests besser beurteilen zu können, wird hier sowohl die Laufzeiteffizienz in Formel 7.5, sowie die Speichereffizienz in Formel 7.6 festgelegt.

$$Laufzeiteffizienz = \left(\frac{\#Instruktionen_{\text{Zeigervalidierung}}}{\#Instruktionen_{\text{Original}}} - 1 \right) \cdot \frac{1}{FDR} \tag{7.5}$$

$$Speichereffizienz = \left(\frac{\#Bytes_{\text{Zeigervalidierung}}}{\#Bytes_{\text{Original}}} - 1 \right) \cdot \frac{1}{FDR} \tag{7.6}$$

Die Kennzahl der Laufzeiteffizienz gibt dabei den prozentualen Zuwachs an Instruktionen an, die benötigt werden, um ein pro Prozent Fehlererkennungsrate zu ermöglichen. Bei der Speichereffizienz wird dementsprechend der prozentuale Zuwachs in Bytes, der durch die Einbringung der generischen Bereichsprüfungen entsteht, der Erkennungsrate gegenübergestellt. Dabei gilt, je kleiner die Effizienzkennzahl ist, desto besser ist diese bzw. weniger Ressourcen werden für die Fehlererkennung benötigt.

In Abbildung 7.10 ist eine Veranschaulichung zum Vergleich der Effizienz der Zeigervalidierung bei den einzelnen Kernel-Tests gegeben. Dabei stellen die Radien der einzelnen Kreise, die die Tests repräsentieren, die Fehlererkennungsrate dar. Tests mit Erkennungsraten unter zwei Prozent wurden ausgelassen, da deren Effizienzwerte zum Teil erheblich, von den der anderen abweichen und wenig Aussagekraft besitzen.

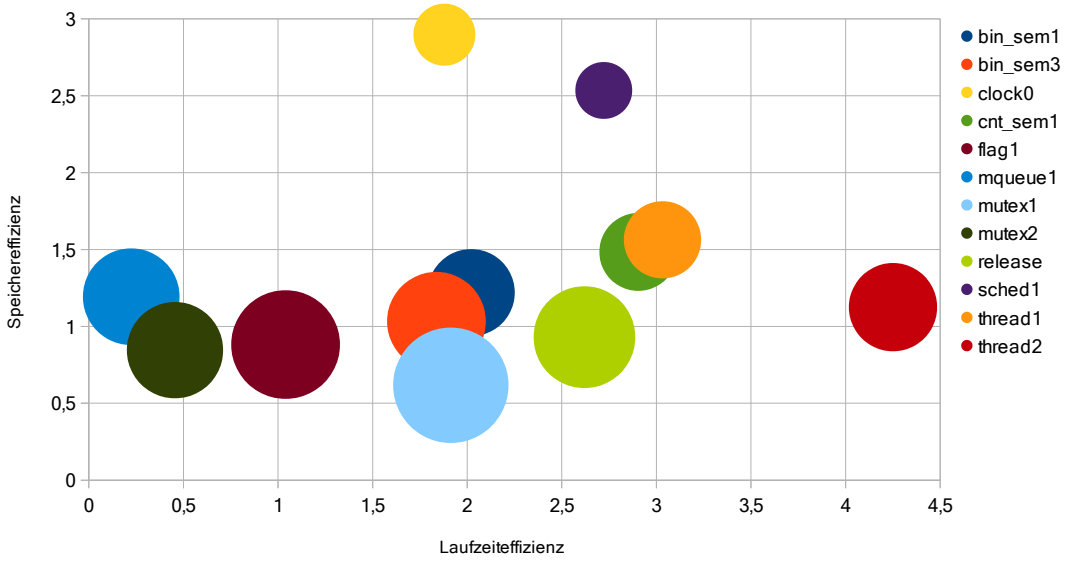


Abbildung 7.10: Veranschaulichung der Effizienz der Zeigervalidierung anhand der verschiedenen Kernel-Tests, wobei ein kleinerer Wert besser ist. Die Höhe der Fehlererkennungsrate ist durch den Radius dargestellt.

Bei der Speichereffizienz residieren die meisten Tests grob um eins herum. Die beiden Ausreißer stellen clock0 und sched1 dar, die aber auch die kleinsten Tests, in Bezug auf die Programmgröße in Byte, sind.

Bei der Laufzeiteffizienz sehen die Unterschiede größer aus, wobei sich hier die drei von der Instruktionsanzahl größten Tests, mqueue1, mutex2 und flag1 alle am linken Rand des Feldes befinden, was die Vermutung nahe legt, dass bei längeren Programmen der Laufzeitzuwachs, durch Einbringung der Zeigervalidierung, weniger ins Gewicht fällt. Der Test, der dieser Vermutung bei erster Betrachtung zu widersprechen scheint und abseits des Hauptfeldes der Laufzeiteffizienz mit einem Wert von größer vier liegt, ist thread2. Bei Betrachtung der Implementierung erkennt man jedoch, dass dieser schlechte Wert von allem den wiederholten Aufrufen von Zugriffsmethoden geschuldet ist, die diesen Test dominieren. Sowohl die Zugriffe durch `set_priority` und `get_priority` auf mehrere `Cyg_Thread` -Objekte, sowie Zugriffe durch `wait` und `post` auf mehrere Semaphoren, die bis auf eine Zuweisungsfunktion bzw. Rückgabefunktion keinerlei Instruktionen besitzen, jedes Mal jedoch der That-Zeiger durch die Bereichsprüfung validiert wird, sorgen dafür, dass im Quellcode auf wenige Instruktionen stets eine Prüfung desselben That-Zeigers folgt und so ein hoher Instruktionsoverhead entsteht, ohne jedoch die Fehlererkennungsrate deutlich zu steigern.

7.4.3 Bewertung der einzelnen Prüfungstypen

Um den Einfluss der Prüfungstypen auf das Gesamtergebnis zu ermitteln, wurden diese jeweils einzeln getestet. In Abbildung 7.11 ist der Anteil der Erkennungsrate der einzelnen Prüfungstypen an der Gesamterkennungsrate graphisch in einem Netzdiagramm

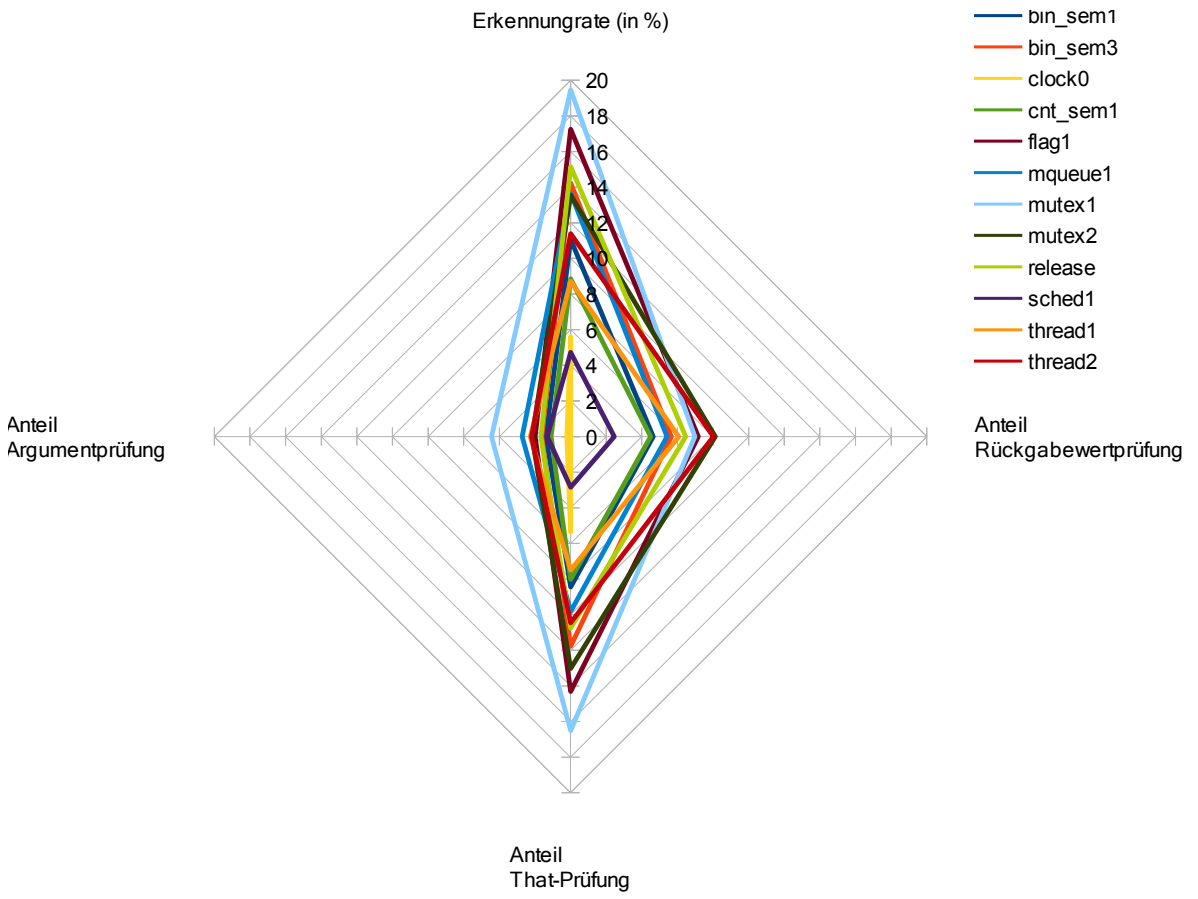


Abbildung 7.11: Veranschaulichung des Anteils des jeweiligen Prüfungstyps an der Gesamterkennungsrate. Wobei zur besseren Visualisierung nur Tests mit Erkennungsraten von mehr als zwei Prozent berücksichtigt werden.

dargestellt.

Dabei ist zu beachten, dass auch hier Tests mit Erkennungsraten von unter zwei Prozent nicht in das Diagramm eingeflossen sind, da diese lediglich einen Punkt darstellen würden. Abbildung 7.11 zeigt, dass beim Anteil an der Gesamterkennungsrate vor allem die That-Prüfung, gefolgt von der Rückgabetypprüfung, eine entscheidende Rolle spielt. Die Argumentprüfung hingegen scheint nur einen geringen Teil an der gesamten Fehlererkennungsrate zu haben, wobei dieses natürlich auch mit der Natur der hier verwendeten Tests zusammenhängt.

Ebenfalls zu erwähnen ist, dass die einzelnen Prüfungstypen nicht überschneidungsfrei arbeiten, d. h., dass Fehler, die mit einem Prüfungstyp gefunden werden, evtl. ebenfalls von einer anderen Prüfung detektiert werden, weswegen die Summe der Fehlererkennungsraten der einzelnen Prüfungen größer als die Gesamterkennungsrate ist. Insgesamt liegt der Anteil der an Überschneidungen bei 43%, was bedeutet, dass durchschnittlich $4,25\%$ des Fehlererkennungsmittelwerts von $9,83\%$, von mehr als einer Prüfung lokalisiert wurden.

7.4.3.1 Argumentprüfung

Test	**FDR**	**Total**	**ROM**	**RAM**		**Laufzeit**
			text	*data*	*bss*	
bin_sem1	1,38%	56926 (+1,32%)	28766	1024	27136	7130 (+2,72%)
bin_sem3	1,62%	57338 (+1,35%)	29178	1024	27136	11247 (+3,54%)
clock0	0,21%	52186 (+1,56%)	31194	1024	19968	7729 (+1,99%)
cnt_sem0	0,01%	47934 (+1,43%)	26910	1024	20000	4805 (+0,00%)
cnt_sem1	1,10%	56926 (+1,26%)	28766	1024	27136	7679 (+1,90%)
flag0	0,00%	47994 (+1,48%)	26970	1024	20000	3373 (+0,00%)
flag1	1,99%	63166 (+1,13%)	31422	1024	30720	33808 (+1,76%)
mqueue1	2,72%	62270 (+1,63%)	33598	1024	27648	126787 (+0,42%)
mutex1	4,44%	59870 (+1,20%)	28094	1024	30752	8902 (+3,42%)
mutex2	2,17%	64990 (+1,15%)	29662	1024	34304	65116 (+0,65%)
release	1,66%	56158 (+1,34%)	27998	1024	27136	6550 (+2,79%)
sched1	1,31%	55386 (+1,28%)	27258	1024	27104	4226 (+1,73%)
thread0	0,75%	51134 (+1,34%)	26718	1024	23392	4171 (+1,41%)
thread1	2,26%	55642 (+1,33%)	27514	1024	27104	5752 (+3,53%)
thread2	2,19%	60670 (+1,23%)	28926	1024	30720	8677 (+5,61%)

Tabelle 7.22: Untersuchung der Fehlererkennungsrate (FDR), sowie des Speicher- und Laufzeitoverheads bei der Argumentprüfung. Die Laufzeit bezeichnet dabei die Instruktionen eines kompletten Testprogrammdurchlaufs, die Größe ist in Byte angegeben.

Die Argumentprüfung alleine erreicht Fehlererkennungsraten von bis zu $4,44\%$, wie den Ergebnissen aus Tabelle 7.22 zu entnehmen ist. Während der Speicherplatzoverhead bei allen Tests fast konstant ist und gerade einmal um $0,5\%$ schwankt, steigt der Laufzeitoverhead in den meisten Fällen mit der Erkennungsrate, allerdings auch nur bis zum Maximalwert von gerade einmal $5,61\%$.

Die meisten Tests benutzen kaum Methoden, in denen Zeiger in der Argumentliste verwendet werden. Die häufigsten Argumentzeigerprüfungen finden beim Scheduling statt, bei dem Methoden wie `Cyg_HardwareThread::thread_entry(Cyg_Thread *thread)`, `Cyg_Scheduler_Implementation::rem_thread(Cyg_Thread *thread)` oder `Cyg_Scheduler::add_thread(Cyg_Thread*)` den Hauptteil der eingebrachten Bereichsprüfungen bilden.

Bei ungeschickten Kombinationen, wie bei Test `thread2`, in der durch hintereinanderfolgende Prioritätenwechsel viele solcher Aufrufe folgen, leidet die Effizienz dementsprechend, während der Test `mqueue1` ziemlich gut abschneidet, da dieser Test einer der wenigen ist, in dem viel Zeiger und Referenzen auf eigene Typen in Argumentlisten weitergegeben werden.

Was die Effizienz der Argumentprüfung angeht, ist diese wie bereits gesagt im Bezug zum Speicherplatzverbrauch fast konstant, was sich auch im Diagramm in Abbildung 7.12 widerspiegelt. Auch bei der Laufzeiteffizienz sind die Tests relativ kompakt

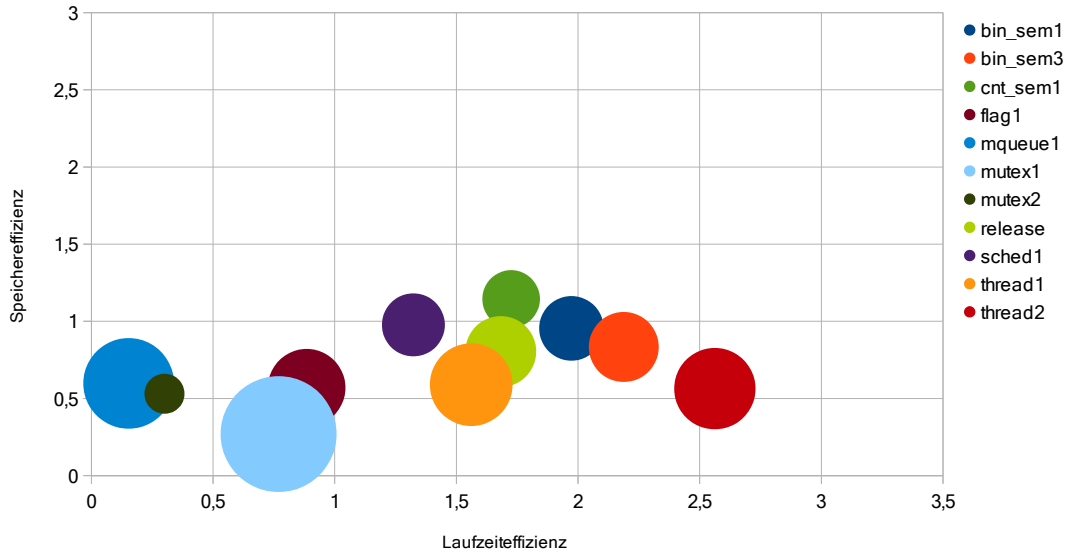


Abbildung 7.12: Veranschaulichung der Effizienz der Argumentprüfung, wobei ein kleinerer Wert besser ist. Die Radien repräsentieren die Höhe der Fehlererkennungsrate.

angeordnet und bewegen sich im Bereich bis $2,56$. Um weitere Aussagen über die Ergebnisse zu machen, sind die Erkennungsraten allerdings zu gering. Hier müssten vielleicht andere Tests, die vermehrt Gebrauch von Zeigern und Referenzen in der Argumentliste machen, zur Hilfe genommen werden.

7.4.3.2 That-Zeiger-Prüfung

Bei der Validierung des Zeigers, auf das der Methode zugehörigem Objekt, wurden durchschnittlich die höchsten Fehlererkennungsraten der drei Prüfungstypen erreicht, wie bereits in Abbildung 7.11 zu sehen. Wie Tabelle 7.23 zu entnehmen, reichen diese bis zu $16,51\%$ und betragen lediglich bei den bereits angesprochenen „Null-Tests“, `thread0`, `flag0` und `cnt_sem0` weniger als zwei Prozent.

Um die stark unterschiedlichen Fehlererkennungsraten, sowie die starke Streuung der Ergebnisse bei der Effizienz in Abbildung 7.13 zu erklären, kann man sich die beiden Tests `mutex2`, der bei einer hohen Erkennungsrate eine gute Effizienz aufweist und `sched1`, der bei niedriger Erkennungsrate die schlechteste Effizienz vorzuweisen hat, näher betrachten.

Dabei vollführt ein Thread in Test `sched1`, viele Scheduleraufrufe hintereinander ohne Kontextwechsel, sorgt also dafür, dass stets der gleiche Objektzeiger überprüft wird, während bei Test `mutex2` vier Threads sowie zugehörige Synchronisationsobjekte, abwechselnde Aktionen durchführen, die eingebrachten Objektprüfungen also auf einen großen Bereich, sowie zeitlich verteilt sind.

Test	**FDR**	**Total**	**ROM**	**RAM**		**Laufzeit**
			text	*data*	*bss*	
bin_sem1	8,45%	61310 (+ 9,12%)	33150	1024	27136	5124 (+16,61%)
bin_sem3	11,77%	61950 (+ 9,50%)	36602	1024	27136	12848 (+18,28%)
clock0	5,36%	57594 (+12,08%)	31194	1024	19968	8251 (+ 8,88%)
cnt_sem0	0,26%	51962 (+ 9,95%)	30938	1024	20000	4883 (+ 1,62%)
cnt_sem1	8,02%	61182 (+ 8,83%)	33022	1024	27136	8783 (+16,55%)
flag0	0,84%	52026 (+10,01%)	31002	1024	20000	3425 (+ 1,54%)
flag1	14,31%	68826 (+10,20%)	37082	1024	30720	37641 (+13,30%)
mqueue1	9,82%	67514 (+10,18%)	38842	1024	27648	129071 (+ 2,23%)
mutex1	16,51%	64126 (+ 8,39%)	32350	1024	30752	11035 (+28,19%)
mutex2	13,02%	69246 (+ 7,78%)	33918	1024	34304	67185 (+ 3,85%)
release	10,75%	60574 (+ 9,30%)	32414	1024	27136	7936 (+24,54%)
sched1	2,85%	59290 (+ 8,42%)	31162	1024	27104	4503 (+ 8,40%)
thread0	1,17%	55070 (+ 9,14%)	30654	1024	23392	4523 (+ 9,97%)
thread1	7,49%	59838 (+ 8,97%)	31710	1024	27104	6448 (+16,05%)
thread2	10,45%	65050 (+ 8,54%)	33306	1024	30720	10726 (+30,55%)

Tabelle 7.23: Untersuchung der Fehlererkennungsrate (FDR), sowie des Speicher- und Laufzeitoverheads bei der That-Zeiger-Prüfung. Die Laufzeit bezeichnet dabei die Instruktionen eines kompletten Testprogrammdurchlaufs, die Größe ist in Byte angegeben.

7.4.3.3 Rückgabetypprüfung

Beim letzten Prüfungstyp, die des Rückgabezeigers, deren Ergebnisse in Tabelle 7.24 aufgelistet sind, wurden vor allem bei den Tests, die viel Gebrauch vom Scheduler machen, gute Resultate erzielt, da dieser einige essenzielle Funktionen mit Zeigern auf Thread-Objekte als Rückgabetyp, wie bspw. `Cyg_Scheduler_Base::get_Current_thread` oder `Cyg_Scheduler_Implementation::schedule`, besitzt. Hierbei wurden Fehlererkennungsraten von bis über acht Prozent erreicht.

Der Test `mqueue1` verletzt die Annahme (siehe Abschnitt 6.1), wonach Funktionszeiger nur in das Textsegment zeigen und dementsprechend nicht Null sein dürfen. Die Funktion `Cyg_Mqueue::setnotify`, die eine Callback-Funktion an die Nachrichtenwarteschlange übergibt und die alte Callback-Funktion als Rückgabewert an den Aufruf weiterreicht, gibt für den Fall, dass vorher keine Funktion gesetzt war, einen Nullzeiger zurück und würde so einen Fehler erzeugen.

```
1 // CustomType.hxx
2 template <typename T>
3 struct pointerType<void (*)(Cyg_Mqueue&, CYG_ADDRWORD)> {
4   static const result_type result = static_cast<result_type>(MQUEUE_FUNCTION);
5 };
6
7 // CustomCheck.hxx
8 template <>
```

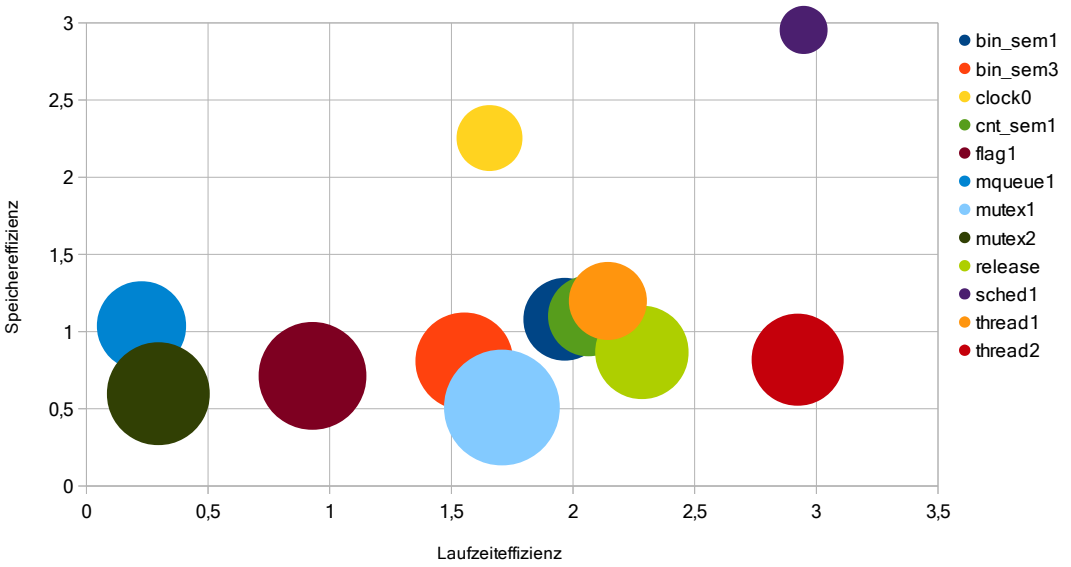


Abbildung 7.13: Veranschaulichung der Effizienz der That-Prüfung, wobei ein kleinerer Wert besser ist. Die Radien repräsentieren die Höhe der Fehlererkennungsrate.

```
9  struct boundary<TypeTraits::MQUEUE_FUNCTION> {
10   template<typename T>
11   static inline void doCheck(T ptr) {
12     const void* addr = reinterpret_cast<const void*>(ptr);
13     if (addr != (void*)0 && addr < (void*)SECTION_TEXT_START_ADDRESS || addr >= (void*)
         SECTION_TEXT_END_ADDRESS) {
14       doOnError();
15     }
16   }
17 };
```

Listing 7.2: Fehlerbehebung bei der Prüfung der Callback-Funktion in `mqueue1`, mithilfe der adaptiven Einzelobjektprüfung.

Um dieses Problem zu lösen, sind mehrere Ansätze denkbar. Zum einen lässt sich die Funktionszeigerprüfung dahin gehend ändern, sodass zukünftig Nullzeiger zulässig sind, zum Anderen lässt sich die Callback-Funktion dahingehen ändern, dass sie keinen Nullzeiger zurückgibt, sondern bspw. einen Zeiger auf eine extra definierte Default-Funktion. Hier wurde das Problem mithilfe der adaptiven Einzelopbjektprüfung (vgl. Abschnitt 6.4) umgangen, in dem für die Callback-Funktion eine Ausnahmeregelung eingeführt wurde, wie in Listing 7.2 zu sehen.

Bei der Darstellung der Effizienz der einzelnen Tests in Abbildung 7.14, ist zu sehen, dass Overhead und Fehlererkennungsrate bei der Rückgabetypvalidierung gut skalieren, da das Ergebnisfeld am kompaktesten von allen Prüfungstypen ist.

Der Test `release` schneidet hierbei am schlechtesten ab, was daran liegt, dass innerhalb des Tests in einer Schleife einhundert Mal hintereinander der Scheduler aufgerufen wird und so derselbe Zeiger innerhalb kurzer Zeit wiederholt überprüft wird, was sich vor

Test	**FDR**	**Total**	**ROM**	**RAM**		**Laufzeit**
			text	*data*	*bss*	
bin_sem1	4,64%	57946 (+3,13%)	29786	1024	27136	7290 (+ 5,03%)
bin_sem3	5,66%	58814 (+3,96%)	30654	1024	27136	11609 (+ 6,88%)
clock0	0,00%	52990 (+3,12%)	31998	1024	19968	7604 (+ 0,34%)
cnt_sem0	0,32%	48730 (+3,11%)	27706	1024	20000	4805 (+ 0,00%)
cnt_sem1	4,50%	57918 (+3,02%)	29758	1024	27136	7948 (+ 5,47%)
flag0	0,28%	48798 (+3,18%)	27774	1024	20000	3373 (+ 0,00%)
flag1	7,17%	64794 (+3,74%)	33050	1024	30720	34779 (+ 4,68%)
mqueue1	5,42%	63454 (+3,56%)	34590	1024	27840	127265 (+ 0,80%)
mutex1	6,98%	60634 (+2,49%)	28858	1024	30752	9468 (+ 9,99%)
mutex2	8,13%	65790 (+2,40%)	30462	1024	34304	65495 (+ 1,24%)
release	6,47%	57338 (+3,46%)	29178	1024	27136	7237 (+13,58%)
sched1	2,46%	56314 (+2,98%)	28186	1024	27104	4258 (+ 2,50%)
thread0	0,13%	51930 (+2,92%)	27514	1024	23392	4239 (+ 3,06%)
thread1	6,07%	56730 (+3,31%)	28602	1024	27104	5844 (+ 5,18%)
thread2	8,02%	61690 (+2,94%)	29946	1024	30720	8970 (+ 9,18%)

Tabelle 7.24: Untersuchung der Fehlererkennungsrate (FDR), sowie des Speicher- und Laufzeitoverheads bei der Rückgabetypüberprüfung. Die Laufzeit bezeichnet dabei die Instruktionen eines kompletten Testprogrammdurchlaufs, die Größe ist in Byte angegeben.

allem negativ auf die Laufzeit auswirkt.

Auch der Test `sched1` fällt hier bei der Speichereffizienz etwas aus der Reihe, was durch die Tests von vielen verschiedenen Schedulerfunktionen bedingt ist. Die anderen Tests bewegen sich jeweils im Bereich von unter eins bei der Speichereffizienz sowie im Bereich von unter $1,5$ bei der Laufzeiteffizienz.

7.4.4 Untersuchung der Heapprüfung

Da keiner der Kernel-Tests Speicher außerhalb der eigenen Anwendung alloziert, also keinen Gebrauch vom Heap macht, wurde um die dynamische sowie die statische Heapprüfung zu vergleichen ein eigenes Testprogramm erstellt. Hierzu wurde der Test für die Treiberschnittstelle `driver3` verwendet und modifiziert. Zu Beginn wird nun, anstatt das Handle mit Null zu initialisieren, diesem eine Adresse aus dem Heap zugewiesen, durch einen Aufruf von `malloc(sizeof(cyg_devtab_entry_t))` aus der `<stdlib.h>`.

Wie in Tabelle 7.25 zu sehen, ist der Speicherplatzverbrauch des Programms mit Fehlertoleranzmechanismen und aktivierter Heapprüfung mehr als doppelt so groß, als beim original Test, was vor allem dem Zuwachs an Quelltext geschuldet ist. Dies hängt damit zusammen, dass zusätzliche Teile der für die dynamische Allokation eingebundenen Bibliothek `<stdlib.h>` gelinkt werden.

Insgesamt unterscheiden sich die dynamische Heapprüfung, die nach jedem Heapzu-

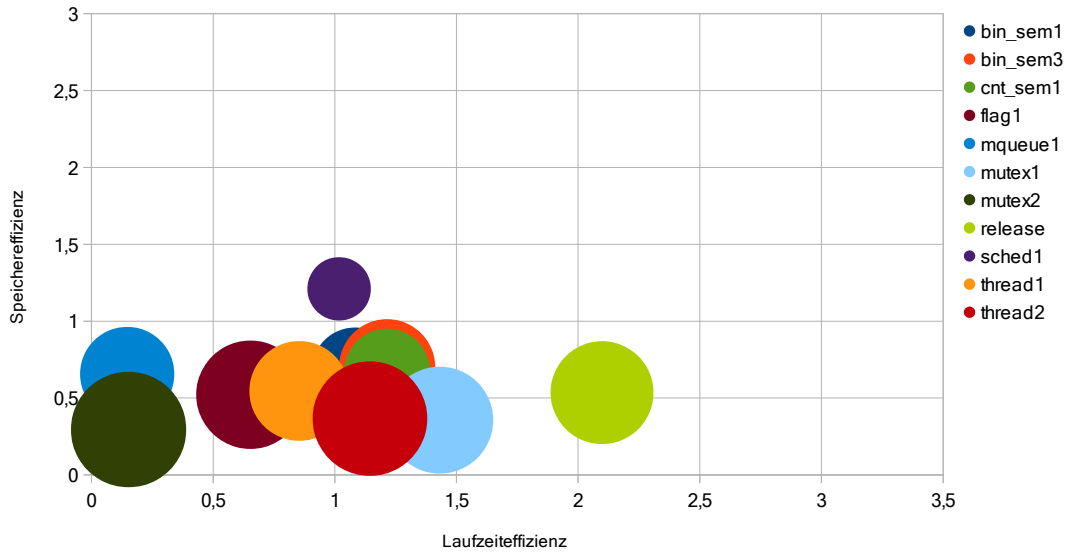


Abbildung 7.14: Veranschaulichung der Effizienz der Rückgabetypprüfung, wobei ein kleinerer Wert besser ist. Die Radien repräsentieren die Höhe der Fehlererkennungsrate.

Test	**FDR**	**Total**	**ROM**	**RAM**		**Laufzeit**
			text	*data*	*bss*	
original	-	50302	29214	896	20192	23813
heap static	14,09%	107793	84353	932	22508	24648
		+114,29%	+188,74%	+4,02%	+11,47%	+3,51%
heap dynamic	17,73%	109265	85793	932	22540	25887
		+117,22%	+193,67%	+4,02%	+11,63%	+8,71%

Tabelle 7.25: Ergebnisse zum Vergleich zwischen dynamischer und statischer Heapprüfung. Die Laufzeit bezeichnet dabei die Instruktionen eines kompletten Testprogrammdurchlaufs, die Größe ist in Byte angegeben.

griff die aktuelle Heapgröße für die Bereichsprüfung zwischenspeichert und die statische Heapprüfung, die den kompletten möglichen Heap als Bereich für die Validitätsprüfung annimmt, um drei Prozent beim Speicheroverhead und fünf Prozent beim Laufzeitoverhead, bei einer um ein Viertel höheren Fehlererkennungsrate.

Da hier lediglich ein Test zur Evaluation der Heapprüfung herangezogen wurde, müssten für verlässlichere Werte allerdings weitere Tests erfolgen.

7.4.5 Generische Zeigervalidierung in der Treiberschnittstelle

Zum Abschluss soll ein Vergleich der Fehlererkennung durch die generische Zeigervalidierung, mit der in Abschnitt 5 beschriebenen Absicherung der Treiberschnittstelle

erfolgen.

```
1  pointcut criticalFunctions() = ("% cyg_io_%(...)"
2                 || "% cyg_devio_table::%(...)"
3                 || "% cyg_devtab_entry::%(...)"
4                 || "% serial_%(...)"
5                 || "% tty_%(...)"
6                 || "% pc_serial_%(...)");
7
8  pointcut criticalClasses() = ("");
```

Listing 7.3: Pointcut Expression für die generische Bereichsprüfung in der Treiberschnittstelle.

Die Auswahl der durch die generische Bereichprüfung zu schützenden Funktionen, ist durch die Pointcut Expressions in Listing 7.3 festgelegt, wobei bei den Tests aus Tabelle 7.26 mit einem „API“ am Ende, die Einbringung der generischen Zeigervalidierung lediglich in die Treiberschnittstelle und nicht in die Gerätetreiber selbst erfolgt, d. h. ohne die Pointcuts `"% serial_%(...)"`, `"% tty_%(...)"` und `"% pc_serial_%(...)"`.

Test	**FDR**	**Total**	**ROM**	**RAM**		**Laufzeit**
			text	*data*	*bss*	
driver3 API	9,87%	50202 (+ 1,48%)	29210	1024	19968	23103 (+1,17%)
driver3 all	13,13%	51994 (+ 5,10%)	31002	1024	19968	23336 (+2,19%)
tty2 API	4,62%	54906 (+ 1,35%)	30330	1024	23552	88153 (+0,32%)
tty2 all	9,40%	56634 (+ 4,54%)	32058	1024	23552	90039 (+2,47%)
loop API	7,65%	58170 (+ 6,95%)	29978	896	27296	75964 (+1,68%)
loop all	10,13%	60030 (+10,18%)	31838	896	27296	79070 (+5,84%)

Tabelle 7.26: Ergebnisse der generische Zeigervalidierung in der Treiberschnittstelle. Die Laufzeit bezeichnet dabei die Instruktionen eines kompletten Testprogrammdurchlaufs, die Größe ist in Byte angegeben.

Während die *GuardPointer* und Akzessoren Fehlererkennungsraten von annähernd 80% erreichen (vgl. Tabelle 7.18), werden hier lediglich Fehlererkennungsraten von bis zu 13, 13% erreicht. Selbst einzeln erzielen die maßgeschneiderten Lösungen höhere Erkennungsraten, bei niedrigeren Kosten für Speicherverbrauch und Laufzeitoverhead.

Der Grund liegt hierbei natürlich im spezialisierten Verwendungszweck. Während Zugriffsfunktionen und *GuardPointer* nur zwei verschiedene Objekte überprüfen, die sich in einem sehr beschränkten Bereich aufhalten, lässt sich die generische Zeigervalidierung für multiple Verwendungszwecke einsetzen. Wie bereits in Abschnitt 7.3 zu sehen, ist gerade die Größe des zu validierenden Bereichs von entscheidendem Einfluss bei der Fehlererkennungsrate.

7.4.6 Zusammenfassung

Die Ergebnisse der generischen Zeigervalidierung kommen bei Weitem nicht an die Erkennungsraten, die bei der Absicherung der Treiberschnittstelle erreicht wurden. Feh-

lererkennungsraten von durchschnittlich unter zehn Prozent, reichen dabei als alleinige Maßnahme zur Fehlertoleranz jedoch kaum aus.

Für generische Prüfungen sind sie dennoch bei Speicherplatz- sowie Laufzeitoverhead im unteren Bereich. Zudem sind die Tests und somit die hier aufgeführten Ergebnisse stark von der Implementierung des Schedulers geprägt. Würde bspw. der Scheduler anstelle der `return`-Anweisung einen Rückgabezeiger aus der Argumentliste verwenden, könnten die Ergebnisse ganz anders aussehen.

8 Fazit und Ausblick

Ziel dieser Diplomarbeit war es, verschiedene softwarebasierte Fehlertoleranzmechanismen mittels *AspectC++* zu implementieren und ihre Performanz anschließend zu bewerten. Die Wahl der Fehlertoleranzmechanismen fiel dabei auf die Plausibilitätsprüfung, bzw. im Speziellen auf die Bereichsprüfung. Für jeweils zwei ausgesuchte Szenarien wurden Implementierungen erstellt und im Anschluss evaluiert. Die Ergebnisse sollen hier diskutiert werden.

8.1 Absicherung der Treiberschnittstelle

Für die Absicherung des E/A-Subsystems in *eCos* bzw. der Treiberschnittstelle wurden hauptsächlich zwei Verfahren entwickelt. Die sog. *GuardPointer*, die für eine Validierung der Dereferenzierungsfunktionen eingesetzt wurden, sowie verschiedene Bereichsprüfungen bei Zugriffsoperationen innerhalb der Schnittstelle, erreichten bei geschickter Kombination Fehlererkennungsraten von fast achtzig Prozent, bei gleichzeitig minimalem Speicher- und Laufzeitoverhead. Die Erkennungsrate bezieht sich dabei allerdings ausschließlich auf Fehler in den Treibern selbst.

Insgesamt waren für das Einbringen der Fehlertoleranzmechanismen, nur wenige Änderungen am vorhandenen Quellcode der Treiberschnittstelle nötig. Durch die Implementierung mithilfe von Aspekten, sind diese außerdem jederzeit ohne Aufwand konfigurierbar.

Auch wenn hier die Ergebnisse sehr zufriedenstellend ausfallen, so ist jedoch anzumerken, dass die hohen Fehlererkennungsraten vor allem dem stark eingegrenzten Bereich in dem sich die zu validierenden Objekte befanden, zu verdanken ist. So ist das grundlegende Konzept wie etwa der *GuardPointer*, auch an anderer Stelle und für andere Zwecke einsetzbar, erzielt aber bei größeren Gültigkeitsbereichen keine derart hohen Erkennungsraten. Hier könnten andere Prüfverfahren von Vorteil sein.

8.2 Generische Zeigervalidierung

Wie sich ein größerer Validitätsbereich bei der Bereichsprüfung auswirkt, kann bei der generischen Zeigervalidierung beobachtet werden. Während die Umsetzung mittels generativer Advices, gezeigt hat, wie man modular und unter Beachtung der Trennung von Belangen, Zusatzfunktionen erfolgreich implementieren kann und zudem die Konfigurierbarkeit dieser vereinfacht (siehe dazu Abbildung 8.1), fallen die abschließenden Ergebnisse eher enttäuschend aus.

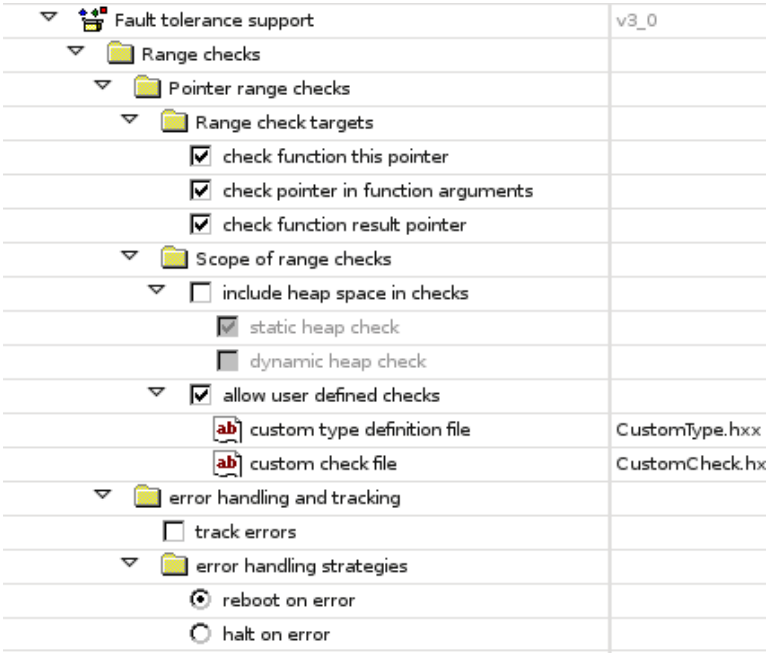


Abbildung 8.1: Exemplarische Umsetzung und Integration der generischen Bereichsprüfung und ihrer Konfigurationsmöglichkeiten in *eCos* mittels *CDL*.

Bei moderatem Ressourceneinsatz wurden lediglich Erkennungsraten für Speicherfehler, von durchschnittlich knapp zehn Prozent erreicht. Auch wenn sich dieses Verfahren sehr flexibel einsetzen lässt, so sind Erkennungsraten dieser Größenordnung für den Einsatz als allein operierender Fehlertoleranzmechanismus zu niedrig. Das Grundgerüst der generischen Zeigervalidierung hingegen, das aufweist wie Sprachmerkmale von *AspectC++* bei der Implementierung von Fehlerresilienz helfen können, wäre auf eine Eignung auf andere Prüfverfahren zu testen.

Um die Fehlererkennungsrate bei der generischen Zeigervalidierung zu steigern, wäre zudem eine Art automatischer Identifizierung von kritischen Objekten denkbar. Mit Hilfe von Metadaten über diese Objekte ließen sich so, via adaptiver Einzelobjektprüfung, angepasste und spezifische Prüfungen erstellen und einbringen. Informationen, etwa zum Speicherlayout und -ausrichtung, die auch schon bei der Absicherung der Treiberschnittstelle benutzt wurden, könnten dazu beitragen, den Validitätsbereich einzuschränken.

Abschließend lässt sich vor allem über die Bereichsprüfung allgemein sagen, dass diese für sehr kleine Validitätsbereiche gute Ergebnisse bei niedrigem Ressourceneinsatz und einfacher Implementierung erzielen kann, während die Verwendung für größere Gültigkeitsbereiche nicht zu empfehlen ist.

Literaturverzeichnis

[1] David, Francis M. ; Campbell, Roy H.: Building a Self-Healing Operating System. In: *Proceedings of the Third IEEE International Symposium on Dependable, Autonomic and Secure Computing*. Washington, DC, USA : IEEE Computer Society, 2007 (DASC '07). – ISBN 0–7695–2985–2, 3–10

[2] Swift, Michael M. ; Annamalai, Muthukaruppan ; Bershad, Brian N. ; Levy, Henry M.: Recovering device drivers. In: *ACM Trans. Comput. Syst.* 24 (2006), November, Nr. 4, 333–360. http://dx.doi.org/10.1145/1189256.1189257. – DOI 10.1145/1189256.1189257. – ISSN 0734–2071

[3] Kiczales, Gregor ; Lamping, John ; Mendhekar, Anurag ; Maeda, Chris ; Lopes, Cristina ; Loingtier, Jean marc ; Irwin, John: Aspect-oriented programming. In: *ECOOP*, SpringerVerlag, 1997

[4] Lohmann, Daniel ; Scheler, Fabian ; Tartler, Reinhard ; Spinczyk, Olaf ; Schröder-Preikschat, Wolfgang: A quantitative analysis of aspects in the eCos kernel. In: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. New York, NY, USA : ACM, 2006 (EuroSys '06). – ISBN 1–59593–322–0, 191–204

[5] Kadav, Asim ; Renzelmann, Matthew J. ; Swift, Michael M.: Tolerating hardware device failures in software. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. New York, NY, USA : ACM, 2009 (SOSP '09). – ISBN 978–1–60558–752–3, 59–72

[6] Dijkstra, Edsger W.: *A Discipline of Programming*. 1st. Upper Saddle River, NJ, USA : Prentice Hall PTR, 1997. – ISBN 013215871X

[7] Pohl, Klaus ; Böckle, Günter ; Linden, Frank J. van d.: *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA : Springer-Verlag New York, Inc., 2005. – ISBN 3540243720

[8] Lohmann, Daniel ; Spinczyk, Olaf: Developing embedded software product lines with AspectC++. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. New York, NY, USA : ACM, 2006 (OOPSLA '06). – ISBN 1–59593–491–X, 740–742

[9] Spinczyk, Olaf ; Gal, Andreas ; Schröder-Preikschat, Wolfgang: AspectC++: an aspect-oriented extension to the C++ programming language. In:

Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications. Darlinghurst, Australia, Australia : Australian Computer Society, Inc., 2002 (CRPIT '02). – ISBN 0–909925–88–7, 53–60

[10] Spinczyk, Olaf ; Lohmann, Daniel: The design and implementation of AspectC++. In: *Know.-Based Syst.* 20 (2007), Oktober, Nr. 7, 636–651. http://dx.doi.org/10.1016/j.knosys.2007.05.004. – DOI 10.1016/j.knosys.2007.05.004. – ISSN 0950–7051

[11] Kiczales, Gregor ; Hilsdale, Erik ; Hugunin, Jim ; Kersten, Mik ; Palm, Jeffrey ; Griswold, William G.: An Overview of AspectJ. In: *Proceedings of the 15th European Conference on Object-Oriented Programming*. London, UK, UK : Springer-Verlag, 2001 (ECOOP '01). – ISBN 3–540–42206–4, 327–353

[12] Urban, Matthias ; Spinczyk, Olaf: *AspectC++ Language Reference*. http://www.aspectc.org/fileadmin/documentation/ac-languageref.pdf. Version: 04 2011

[13] Urban, Matthias ; Lohmann, Daniel ; Spinczyk, Olaf: Transactions on aspect-oriented software development VIII. Version: 2011. http://dl.acm.org/citation.cfm?id=2028556.2028563. Berlin, Heidelberg : Springer-Verlag, 2011. – ISBN 978–3–642–22030–2, Kapitel PUMA: an aspect-oriented code analysis and manipulation framework for C and C++, 141–162

[14] Spinczyk, Olaf ; Lohmann, Daniel ; Urban, Matthias: Advances in AOP with AspectC++. In: *Proceedings of the 2005 conference on New Trends in Software Methodologies, Tools and Techniques: Proceedings of the fourth SoMeT W05*. Amsterdam, The Netherlands, The Netherlands : IOS Press, 2005. – ISBN 1–58603–556–8, 33–53

[15] Alexandrescu, Andrei: *Modern C++ design: generic programming and design patterns applied*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2001. – ISBN 0–201–70431–5

[16] Czarnecki, Krzysztof ; Eisenecker, Ulrich W.: *Generative programming: methods, tools, and applications*. New York, NY, USA : ACM Press/Addison-Wesley Publishing Co., 2000. – ISBN 0–201–30977–7

[17] Lohmann, Daniel ; Blaschke, Georg ; Spinczyk, Olaf: Generic Advice: On the Combination of AOP with Generative Programming in AspectC++. In: Karsai, G. (Hrsg.) ; Visser, E. (Hrsg.): *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE '04)* Bd. 3286, Springer-Verlag, Oktober 2004 (Lecture Notes in Computer Science), S. 55–74

[18] Avizienis, Algirdas ; Laprie, Jean-Claude ; Randell, Brian ; Landwehr, Carl: Basic Concepts and Taxonomy of Dependable and Secure Computing. In: *IEEE*

Trans. Dependable Secur. Comput. 1 (2004), Januar, Nr. 1, 11–33. http://dx.doi.org/10.1109/TDSC.2004.2. – DOI 10.1109/TDSC.2004.2. – ISSN 1545–5971

[19] Abbott, Russell J.: Resourceful systems for fault tolerance, reliability, and safety. In: *ACM Comput. Surv.* 22 (1990), März, Nr. 1, 35–68. http://dx.doi.org/10.1145/78949.78951. – DOI 10.1145/78949.78951. – ISSN 0360–0300

[20] Baumann, Robert: Soft Errors in Advanced Computer Systems. In: *IEEE Des. Test* 22 (2005), Mai, Nr. 3, 258–266. http://dx.doi.org/10.1109/MDT.2005.69. – DOI 10.1109/MDT.2005.69. – ISSN 0740–7475

[21] Wilfredo, Torres: Software Fault Tolerance: A Tutorial / NASA. NASA Langley Technical Report Server, 2000. – Forschungsbericht

[22] Wahbe, Robert ; Lucco, Steven ; Anderson, Thomas E. ; Graham, Susan L.: Efficient software-based fault isolation. In: *Proceedings of the fourteenth ACM symposium on Operating systems principles*. New York, NY, USA : ACM, 1993 (SOSP '93). – ISBN 0–89791–632–8, 203–216

[23] Mahmood, A. ; McCluskey, E. J.: Concurrent Error Detection Using Watchdog Processors-A Survey. In: *IEEE Trans. Comput.* 37 (1988), Februar, Nr. 2, 160–174. http://dx.doi.org/10.1109/12.2145. – DOI 10.1109/12.2145. – ISSN 0018–9340

[24] Alexandersson, Ruben ; Karlsson, Johan: Fault injection-based assessment of aspect-oriented implementation of fault tolerance. In: *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*. Washington, DC, USA : IEEE Computer Society, 2011 (DSN '11). – ISBN 978–1–4244–9232–9, 303–314

[25] Borchert, Christoph ; Schirmeier, Horst ; Spinczyk, Olaf: Protecting the Dynamic Dispatch in C++ by Dependability Aspects. In: *Proceedings of the 1st GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '12)*, German Society of Informatics, September 2012 (Lecture Notes in Informatics), S. 1–15. – to appear

[26] Borchert, Christoph ; Schirmeier, Horst ; Spinczyk, Olaf: Generative Software-based Memory Error Detection and Correction for Operating System Data Structures. In: *Proceedings of the 43nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*, IEEE Computer Society Press, Juni 2013. – to appear

[27] Carreira, João ; Madeira, Henrique ; Silva, João: Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. In: *IEEE Trans. Softw. Eng.* 24 (1998), Februar, Nr. 2, S. 125–136. http://dx.doi.org/10.1109/32.666826. – DOI 10.1109/32.666826. – ISSN 0098–5589

[28] Winter, Stefan ; Sârbu, Constantin ; Suri, Neeraj ; Murphy, Brendan: The impact of fault models on software robustness evaluations. In: *Proceedings of the 33rd International Conference on Software Engineering*. New York, NY, USA : ACM, 2011 (ICSE '11). – ISBN 978–1–4503–0445–0, 51–60

[29] Hsueh, Mei-Chen ; Tsai, Timothy K. ; Iyer, Ravishankar K.: Fault Injection Techniques and Tools. In: *Computer* 30 (1997), April, Nr. 4, 75–82. http://dx.doi.org/10.1109/2.585157. – DOI 10.1109/2.585157. – ISSN 0018–9162

[30] Schirmeier, Horst ; Hoffmann, Martin ; Kapitza, Rüdiger ; Lohmann, Daniel ; Spinczyk, Olaf: FAIL*: Towards a Versatile Fault-Injection Experiment Framework. In: Mühl, Gero (Hrsg.) ; Richling, Jan (Hrsg.) ; Herkersdorf, Andreas (Hrsg.): *25th International Conference on Architecture of Computing Systems (ARCS '12), Workshop Proceedings* Bd. 200, German Society of Informatics, März 2012 (Lecture Notes in Informatics). – ISBN 978–3–88579–294–9, S. 201–210

[31] Schirmeier, Horst ; Hoffmann, Martin ; Kapitza, Rüdiger ; Lohmann, Daniel ; Spinczyk, Olaf: Revisiting Fault-Injection Experiment-Platform Architectures. In: *Proceedings of the 17th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '11)*. Pasadena, USA : IEEE Computer Society Press, Dezember 2011, S. 284–285. – Fast abstract

[32] The Bochs Project: *bochs: The Open Source IA-32 Emulation Project*. http://bochs.sourceforge.net/

[33] Marwedel, P.: *Embedded System Design*. Secaucus, NJ, USA : Springer-Verlag New York, Inc., 2006. – ISBN 1402076908

[34] Afonso, Francisco ; Silva, Carlos ; Montenegro, Sergio ; Tavares, Adriano: Applying aspects to a real-time embedded operating system. In: *Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software*. New York, NY, USA : ACM, 2007 (ACP4IS '07). – ISBN 978–1–59593–657–8

[35] Afonso, Francisco ; Silva, Carlos ; Brito, Nuno ; Montenegro, Sergio ; Tavares, Adriano: Aspect-oriented fault tolerance for real-time embedded systems. In: *Proceedings of the 2008 AOSD workshop on Aspects, components, and patterns for infrastructure software*. New York, NY, USA : ACM, 2008 (ACP4IS '08). – ISBN 978–1–60558–142–2, 2:1–2:8

[36] Lohmann, Daniel ; Hofer, Wanja ; Schröder-Preikschat, Wolfgang ; Streicher, Jochen ; Spinczyk, Olaf: CiAO: an aspect-oriented operating-system family for resource-constrained embedded systems. In: *Proceedings of the 2009 conference on USENIX Annual technical conference*. Berkeley, CA, USA : USENIX Association, 2009 (USENIX'09), 16–16

[37] Massa, Anthony: *Embedded Software Development with eCos*. Prentice Hall Professional Technical Reference, 2002. – ISBN 0130354732

[38] Free Software Foundation, Inc.: eCos Reference Manual. (2011). http://ecos.sourceware.org/docs-3.0/

[39] Swift, Michael M. ; Bershad, Brian N. ; Levy, Henry M.: Improving the reliability of commodity operating systems. In: *Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA : ACM, 2003 (SOSP '03). – ISBN 1–58113–757–5, 207–222

[40] Herder, Jorrit N. ; Bos, Herbert ; Gras, Ben ; Homburg, Philip ; Tanenbaum, Andrew S.: Failure Resilience for Device Drivers. In: *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Washington, DC, USA : IEEE Computer Society, 2007 (DSN '07). – ISBN 0–7695–2855–4, 41–50

[41] Stroustrup, Bjarne: *The design and evolution of C++*. New York, NY, USA : ACM Press/Addison-Wesley Publishing Co., 1994. – ISBN 0–201–54330–3

[42] Edelson, Daniel R.: Smart Pointers: They're Smart, but They're Not Pointers. Santa Cruz, CA, USA : University of California at Santa Cruz, 1992. – Forschungsbericht

Abbildungsverzeichnis

Tabellenverzeichnis

Listings