

Bachelorarbeit

**Entwicklung eines Datenloggers
für Beatmungsdaten und
Mutationsanalyse für
Robustheitstests**

Autor

Tobias Schulz

Erstprüferin

Prof. Dr. Sibylle Schupp

Zweitprüfer

Prof. Dr.-Ing. Andreas Timm-Giel

26. August 2011

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt und durch meine Unterschrift, dass die vorliegende Arbeit von mir selbstständig und ohne fremde Hilfe angefertigt worden ist. Inhalte und Passagen, die aus fremden Quellen stammen und direkt oder indirekt übernommen worden sind, wurden als solche kenntlich gemacht. Ferner versichere ich, dass ich keine andere außer der im Literaturverzeichnis angegebenen Literatur verwendet habe. Diese Versicherung bezieht sich sowohl auf Textinhalte, als auch auf alle enthaltenen Abbildungen, Skizzen und Tabellen. Diese Arbeit wurde bisher keiner Prüfungsbehörde vorgelegt und noch nicht veröffentlicht.

Ort, Datum

Unterschrift

Abstrakt – Daten von Beatmungsgeräten aus klinischem und privatem Umfeld werden zur Analyse der medizinischen Therapie genutzt. Nichtinvasive Beatmungsgeräte der Firma Weinmann Geräte für Medizin GmbH + Co. KG sichern Beatmungsdaten intern nur kurzfristig. Die Bachelorarbeit entwickelt einen externen Beatmungsdatenlogger, der die Daten aus Weinmann-Beatmungsgeräten über eine USART-Schnittstelle ausliest und auf SD-Karte speichert. Die Robustheit des Datenloggers wird über eine Mutationsanalyse untersucht. Dafür werden die Mothramutationsoperatoren an die Programmiersprache C++ adaptiert. Die Adaption beinhaltet eine Erweiterung der angewendeten Restriktionen und reduziert dadurch unkompilierbare, doppelte und äquivalente Mutanten.

Inhaltsverzeichnis

1. Einleitung	1
2. Beatmungsdatenlogger	2
2.1. Projektfestlegung	3
2.1.1. Vorgaben	3
2.1.2. Anforderungen	3
2.2. Systementwurf	5
2.2.1. Hardwarewahl	5
2.2.2. Softwaredesign	6
2.3. Implementierung	9
2.3.1. Frameworkeinbindung	9
2.3.2. Threadmanagement	11
2.3.3. USART Kommunikation	13
2.3.4. Implementierungsprogress	13
3. Mutationsanalyse	14
3.1. Theorie	14
3.1.1. Äquivalenter Mutant	15
3.1.2. Analyseverfahren	16
3.1.3. Nutzen der Mutationsanalyse	16
3.2. Adaption und Analyse	17
3.3. Auswertung	23
4. Zusammenfassung und Ausblick	28
Literatur	30
Anhang	I

1. Einleitung

Diese Bachelorarbeit ist extern bei der Firma Weinmann Geräte für Medizin GmbH + Co. KG im Bereich der Business Unit Homecare entstanden. Die Business Unit Homecare entwickelt Systemlösungen für Beatmung, Schlaf- und Sauerstoffmedizin [10]. Medizintechnische Beatmung von Patienten findet im klinischen Umfeld und im häuslichen Bereich der Patienten statt. In beiden Bereichen werden Beatmungsdaten aus den Beatmungsgeräten zur Analyse der Beatmungstherapie ausgewertet.

Aktuelle Weinmann-Beatmungsgeräte ohne SD-Kartenslot speichern Beatmungsdaten nur kurzfristig auf einem internen Ringspeicher. Sobald der Ringspeicher voll ist, werden die ältesten Daten überschrieben. Die Daten können mit Weinmann-Zubehör auf einen PC ausgelesen werden. Das erzeugte Dateiformat ist proprietär und kann nur durch eine Weinmann-Software ausgewertet werden.

Ziel dieser Bachelorarbeit ist es, einen externen Beatmungsdatenlogger zu entwickeln und dessen Robustheit durch eine Mutationanalyse zu untersuchen. Die Beatmungsdaten sollen über eine einheitliche USART-Schnittstelle aus den Beatmungsgeräten ausgelesen werden und auf einer SD-Karte im freien öffentlichen EDF+ Format [7] langfristig abgespeichert werden.

Durch eine kurze Analyse der Prozesse, in denen die Beatmungsgeräte eingebunden sind, werden die Vorteile und Nachteile der Entwicklung des Beatmungsdatenloggers in Abschnitt 2 aufgezeigt. In Abschnitt 2.1 wird das Beatmungsdatenloggerprojekt in Hinblick auf Vorgaben, Anforderungen und mögliche zukünftige Anforderungen bestimmt. Darauf basierend wird ein Systementwurf für den Beatmungsdatenlogger in Abschnitt 2.2 designt und in Abschnitt 2.3 implementiert.

Für die Untersuchung der Robustheit des Beatmungsdatenloggers wird eine Mutationsanalyse verwendet. Die Mutationsanalysetechnik wird in der Firma bisher nicht eingesetzt. In Rahmen des Bachelorarbeitprojekts wird die Anwendbarkeit in Firmenprojekten und der Arbeitsaufwand im Verhältnis zum Nutzen bewertet. Dazu wird ein Überblick über die Mutationsanalyse in Abschnitt 3 erlangt. In Abschnitt 3.1 wird die Theorie der Mutationsanalyse erarbeitet. Die Mutationsanalyse ist zeit- und rechenaufwändig, deswegen wird sie im Rahmen der Bachelorarbeit auf einen Teilaspekt des Beatmungsdatenloggers beschränkt. Abschnitt 3.2 beschäftigt sich mit der Durchführung der Mutationsanalyse. Dazu muss diese an das Projekt angepasst werden. Die Ergebnisse der Analyse werden in Abschnitt 3.3 ausgewertet.

Abschließend werden in Abschnitt 4 die Ergebnisse zusammengefasst und es wird ein Ausblick auf die weitere Entwicklung des Projekts gegeben.

2. Beatmungsdatenlogger

Der Softwareentwicklungsaspekt der Bachelorarbeit wird in Kooperation mit der Firma Weinmann Geräte für Medizin GmbH + Co. KG [10] abgestimmt. Grundlegendes Ziel ist es, innerhalb und im Anschluss an die Bachelorarbeit, einen externen Datenlogger für Weinmann-Beatmungsgeräte zu entwickeln, der Beatmungsdaten ausliest und auf einer SD-Karte abspeichert.

Aktuell besitzen nur die invasivtauglichen Beatmungsgeräte VENTIllogic LS und VENTIllogic plus einen SD-Kartenslot, über den Beatmungsdaten abgespeichert werden. Das rein nichtinvasive Beatmungsgerät VENTImotion 2 speichert Beatmungsdaten auf einem internen Speicher. Dieser speichert maximal 10 Beatmungsperioden mit einer insgesamt maximalen Zeitspanne von 10 Stunden. Der Speicher kann mit entsprechendem Zubehör ausgelesen werden.

Beatmungsgeräte werden vorrangig im Bereich Schlaflabor, Beatmungs- und Intensivstation und im privatem Umfeld des Patienten verwendet. Im Schlaflabor werden Patienten voruntersucht, das Beatmungsgerät optimal auf den Patienten eingestellt und diesem zur privaten Anwendung ausgegeben. Das Einstellen des Beatmungsgerätes dauert in der Regel ein bis zwei Nächte. Dazu werden am Tag die in der Nacht aufgezeichneten Beatmungsdaten ausgewertet. Auf der Beatmungs- und Intensivstation werden Patienten regelmäßig oder auch dauerhaft beatmet und von medizinischem Fachpersonal betreut. Eine Einbindung des Beatmungsgerätes in das Kliniksystem ist üblich. Die Beatmungsdaten werden in Echtzeit oder in regelmäßigen Abständen ausgelesen. Patienten im privaten Umfeld werden regelmäßig von Fachpersonal besucht, um die Beatmungsdaten auszulesen. Bei auftretenden Problemen besteht keine Möglichkeit, die Beatmungsdaten anderweitig zu übermitteln.

Durch die Verwendung eines externen Datenloggers für die nichtinvasiven Beatmungsgeräte kann die Beatmungstherapie erleichtert werden. Für eine kontinuierliche verlustfreie Aufzeichnung der Beatmungsdaten muss nur in regelmäßigen Abständen die SD-Karte ausgewechselt werden. Dies können auch ungeschulte Personen ohne spezielles Zubehör durchführen. Die hohe Kapazität aktueller SD-Karten ermöglicht eine Aufzeichnung der Beatmungsdaten über einen Zeitraum, der deutlich größer als 10 Stunden ist. Die beschriebenen SD-Karten können leicht transportiert werden und die gespeicherten Beatmungsdaten können über ein normales SD-Kartenlesegerät auf einen PC übertragen werden. Voraussetzung dafür ist jedoch, dass jeder Patient zu seinem Beatmungsgerät den externen Beatmungsdatenlogger besitzt. Zur Entwicklung des Beatmungsdatenloggers werden die firmenseitigen Vorgaben und Anforderungen bestimmt, ein Softwareentwurf angefertigt und implementiert.

2.1. Projektfestlegung

Die Entwicklung des externen Beatmungsdatenloggers hält sich an Vorgaben, die für das Firmenprojekt festgelegt sind. Aus den Zielen des Projekts wird in einem iterativen Besprechungsprozess ein Anforderungsprotokoll erstellt. In Anbetracht der kurzen zur Verfügung stehenden Zeit für die Entwicklung des Beatmungsdatenloggers werden die Anforderungen nach Wichtigkeit strukturiert. Mögliche Anforderungen, die nicht im Zeitraum der Bachelorarbeit und in der ersten Projektversion umgesetzt werden können, werden in der Entwicklung des Datenloggers berücksichtigt, um eine spätere Einbindung zu ermöglichen.

2.1.1. Vorgaben

Der Beatmungsdatenlogger muss modular aufgebaut und leicht in die Beatmungsgeräte VENTI-motion 2, VENTIlogic plus und VENTIlogic LS integrierbar sein. Der Datenlogger wird als externe Platine realisiert, soll aber auch für die Weiterentwicklung der Beatmungsgerätesoftware verwendet werden können. Genutzt wird eine 32 Bit Evaluationsplatine von STMicroelectronics (STM). STM32 Mikrocontroller werden bereits in der Firma eingesetzt. Es kann auf Erfahrungen mit der STM32 Produktlinie zurückgegriffen werden und der Datenlogger kann leicht in andere STM32 Projekte portiert werden. Als Programmiersprache wird, wie in den Beatmungsgeräten der VENTI-Produktreihe, C/C++ verwendet. Für die eingebettete STM32 Umgebung steht die Entwicklungsumgebung Atollic True Studio STM32 Pro zur Verfügung.

2.1.2. Anforderungen

Folgende Anforderungen an den Beatmungsdatenlogger sind in der Firma bestimmt worden.

1. Der Beatmungsdatenlogger kommuniziert mit den Beatmungsgeräten VENTI-motion 2, VENTIlogic plus und VENTIlogic LS über eine RS232-Schnittstelle (115200 Baudrate, 8 Datenbits, keine Parität, 1 Stopbit) mit einheitlichen Befehlen. Die für die Anforderungen 2, 3 und 4 benötigten Beatmungsdaten werden über bestimmbare Pollbefehle ausgegeben.
2. Therapiedaten werden vom Beatmungsdatenlogger gepollt. Bei einem Auslesefehler wird das letzte gültige Therapiedatum verwendet. Die Übertragung der Therapiedaten kann über eine CRC-16 (zyklische Redundanzprüfung mit 16 Bit) überprüft werden. Folgende Therapiedaten werden, soweit vorhanden, mit fester Frequenz ausgelesen.

Therapiedatum	Frequenz
Druck	10 Hz
Fluss	10 Hz
Volumen	10 Hz
Atemphase	10 Hz
Beatmungsinfos	1 Hz
Leckagefluss	$\frac{1}{4}$ Hz
Ti/T Verhältnis	$\frac{1}{60}$ Hz
Atemfrequenz	$\frac{1}{60}$ Hz
Sauerstoffsättigung	$\frac{1}{60}$ Hz
Spontaninspirationsanteil	$\frac{1}{60}$ Hz
Spontanexpirationsanteil	$\frac{1}{60}$ Hz
Anzahl Efforts	$\frac{1}{60}$ Hz
Anzahl Fightings	$\frac{1}{60}$ Hz
Tidalvolumen	$\frac{1}{60}$ Hz
Atemminutenvolumen	$\frac{1}{60}$ Hz

3. Parametereinstellungen werden vom Beatmungsdatenlogger gepollt. Bei einem Auslesefehler werden die letzten gültigen Parametereinstellungen verwendet. Die Übertragung der Parametereinstellungen wird über eine CRC-16 überprüft. Alle im jeweiligen Beatmungsgerät vorhandenen Parametereinstellungen werden mit einer Frequenz von $\frac{1}{15}$ Hz ausgelesen.
4. Alarmer werden vom Beatmungsdatenlogger gepollt. Bei einem Auslesefehler werden die letzten gültigen Alarmer verwendet. Die Übertragung der Alarmer wird über eine CRC-16 überprüft. Alle im jeweiligen Beatmungsgerät vorhandenen Alarmer werden mit einer Frequenz von 1 Hz ausgelesen.
5. Acht externe analoge Signale werden mit 10 Hz über einen 12 Bit AD-Wandler mitgeloggt.
6. Alle Daten werden im EDF+ Format auf SD-Karte geschrieben. Das EDF+ Format verwendet eine 60-sekündige Datenblockdauer. Es wird eine EDF+ Datei für jede Therapieperiode angelegt. Die Therapiedaten, Alarmer und analogen Signale werden, sobald ein Datenblock voll ist, als Signal auf die SD-Karte geschrieben. Die Parametereinstellungen werden zu Beginn der Aufzeichnung und bei auftretenden Änderungen als TAL-Annotation („Time-stamped Annotations List“) [7] gespeichert.

Die Anbindung der Hardwarekomponente SD-Karte (Anforderung 6) ist im Firmenprojekt enthalten, wird aber aufgrund Zeitmangels aus dem Umfang der Bachelorarbeit ausgeschlossen. Diese Komponente ist nur über ein definiertes Interface mit dem Beatmungsdatenlogger verknüpft. Es bildet das Ende des Beatmungsdatenstroms und ist somit nicht für die Funktionalität der anderen Komponenten notwendig.

Berücksichtigt werden zudem folgende mögliche Anforderungen an spätere Versionen des Beatmungsdatenloggers (folgend zukünftige Anforderungen genannt).

1. Bereitstellen eines Webinterfaces über Ethernet.
2. Bereitstellen einer USB-Schnittstelle.
3. Anzeige und Modifikation der Beatmungsgeräteparameter über das Webinterface.
4. Auslesen der gespeicherten Beatmungsdaten von der SD-Karte über das Webinterface oder die USB-Schnittstelle.
5. Bereitstellen der Therapiedaten, Alarme und analogen Signale als Livestream in aufgearbeiteter Darstellung über das Webinterface.
6. Variation der Abtastfrequenz des AD-Wandlers und Angabe der Signalinformationen der AD-Kanäle für das EDF+ Format über das Webinterface.
7. Bereitstellen eines Kommunikationsterminals über das Webinterface.
8. Bereitstellen der von der Analogbox (Weinmann-Zubehör) benötigten RS232-Schnittstelle.

2.2. Systementwurf

Ausgehend von den in Abschnitt 2.1.2 aufgezeigten Anforderungen und Berücksichtigungen wird ein Systementwurf für den Beatmungsdatenlogger entwickelt. Das Beatmungsdatenloggerprojekt wird, nach der Beatmungsgerätereihe „VENTI“ und der bereitgestellten Kommunikationsschnittstelle „remote“, VENTIREMOTE benannt.

2.2.1. Hardwarewahl

Die für VENTIREMOTE verwendete Evaluationsplatine (STM32CMICOS-EVAL) beinhaltet einen STM32F107VC Mikrocontroller. Nur die STM32 Produktfamilien STM32F107xx und STM32F207xx bieten einen Ethernetanschluss, der für die zukünftige Anforderung 1 benötigt wird. Die STM32F107xx Familie bietet eine mit 72 MHz ausreichende Arbeitsfrequenz und ist günstiger als die STM32F207xx Familie mit 120 MHz. Der Baustein besitzt bis zu 5 USARTs (Anforderung 1, zukünftige Anforderung 8), zwei 12 Bit AD-Wandler mit insgesamt 16 Kanälen (Anforderung 5), drei SPI Schnittstellen, über die jeweils eine SD-Karte angesteuert werden kann (Anforderung 6) und einen USB-Controller (zukünftige Anforderung 2). Die Evaluationsplatine bindet bereits die SD-Kartenslot-, USB- und Ethernethardware an den Mikrocontroller an.

2.2.2. Software design

Um die Modularität des Beatmungsdatenloggers zu gewährleisten, wird dieser, so weit möglich, in C++ objektorientiert entworfen. Abbildung 1 zeigt eine UML Klassendiagrammübersicht von VENTIremote. Die Softwareanbindungen der Hardware-

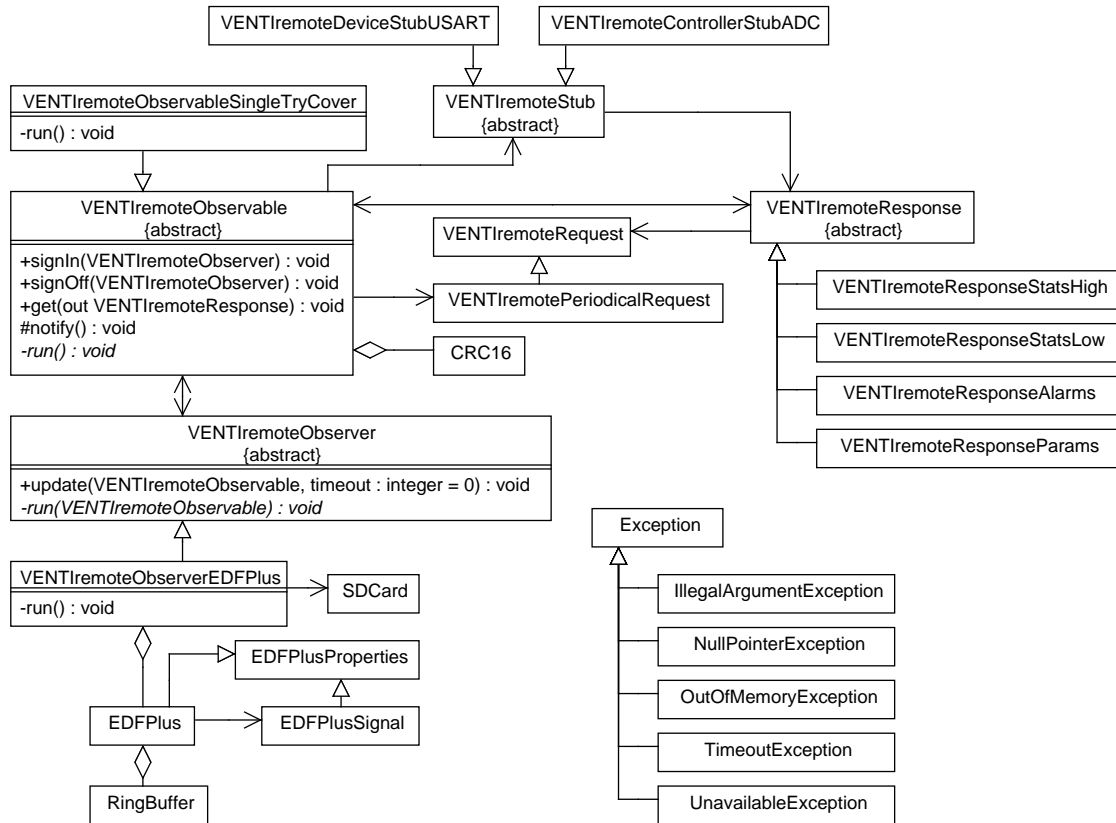


Abbildung 1: UML Klassendiagramm – Übersicht Beatmungsdatenloggerstruktur mit den zentralen Elementen „RingBuffer“ und dem Observer-Pattern.

datenquellen USART/RS232 (VENTIremoteDeviceStubUSART, Anforderung 1) und AD-Wandler (VENTIremoteControllerStubADC, Anforderung 5) werden als Singleton designet, um die einmalige Verfügbarkeit der Hardware in der Software widerzuspiegeln. Die Datenquellen nutzen ein einheitliches Interface (VENTIremoteStub), das in C++ als abstrakte Klasse umgesetzt wird. Dem Interface können weitere Hardwarekomponenten (z.B. durch die zukünftigen Anforderungen 2, 3 und 7) hinzugefügt werden, ohne die Struktur der in Relation stehenden Softwarekomponenten anpassen zu müssen. Die Hardwaredatensinke SD-Karte (SDCard, Anforderung 6 und zukünftige Anforderung 4) wird aus dem selben Grund als Singleton designet.

Als zentrales Element des Beatmungsdatenloggers dient ein Observer-Pattern. Das Observer-Pattern ist ein objektorientiertes Verhaltensmuster, das eine „one-to-many“ Abhängigkeit zwischen einem Subjekt (Observableobjekt) und Observerobjekten dar-

stellt. Ändert sich der Zustand des Observables, werden alle abhängigen Observer benachrichtigt und aktualisiert [2]. Dieses Verhaltensmuster wird verwendet, um Daten vom Observableobjekt über Benachrichtigung und Aktualisierung an Observerobjekte weiterzuleiten, ohne eine komplexe Bindung herzustellen. Die abstrakte minimale Bindung realisiert ein Schichtenmodell. Observable und Observer können unabhängig voneinander variiert werden, solange die in Interfaces definierte Schnittstelle eingehalten wird. Einem Observable können Observer hinzugefügt werden, ohne den Observable oder andere Observer anzupassen. Die Benachrichtigung des Observables wird automatisch an alle interessierten angemeldeten Observer geschickt.

Aufgrund dieser Eigenschaften eignet sich das Observer-Pattern für das VENTIREMOTE Projekt (Modularitäts- und Erweiterbarkeitsanforderungen) als Verbindung zwischen Datenquellen und Datensinken. Der Observable stellt die Daten bereit. Die Observer beziehen die benötigten Daten, indem sie sich am Observable anmelden. Die zukünftigen Anforderungen 1, 5, 8 können als zusätzliche Observer umgesetzt werden. Damit der Observable die Beatmungsdaten über das VENTIREMOTEStub Interface pollen kann, wird er als Thread designet. Dies hat den Vorteil, dass eine feste Pollfrequenz einfach vorgegeben werden kann und der Thread nach der Datenbeschaffung suspendiert wird und somit keine Rechenzeit verbraucht. Der Observablethread beschränkt sich im Datenaustauschprozess darauf, die Observer zu benachrichtigen. Die Aktualisierung der Observer wird von diesen eigenständig durchgeführt. Daraus resultiert für die Observer, dass diese einen eigenen Thread benötigen. Diese Threadverwendung wird eingesetzt, damit das Timing des Observable (nahezu) unabhängig von der Anzahl und Art der Observer bleibt. Verbraucht die selbstständige Aktualisierung der Observer mehr Rechenzeit als durch die Pollfrequenz verfügbar, stehen die Daten im Observable dennoch aktuell zur Verfügung, da dieser sich durch seinen Thread unabhängig Daten pollt.

Abbildung 2 zeigt die Struktur des threadbasierten Observer-Patterns als UML Diagramm. Die „signIn“ und „signOff“ Methoden des Observableinterfaces dienen zum An- und Abmelden der Observer. Die „get“ Methode stellt die Daten in Form einer Antwortklasse zur Verfügung. Die „protected“ Methode „notify“ benachrichtigt die angemeldeten Observer. Sie wird im Observable intern aufgerufen, wenn sich die Daten durch einen Poll erneuert haben. Der Poll wird durch die Threadmethode „run“ ausgelöst. Das Observerinterface beinhaltet die „update“ Methode. Diese wird durch die „notify“ Methode des Observable aufgerufen. Nach dem Aufruf der „update“ Methode aktualisiert sich der Observer selbstständig, indem seine Threadmethode „run“ die „get“ Methode des Observables aufruft. VENTIREMOTE verwendet beim Pollen eine Datenclustering. Argumente in der Überlegung für und gegen die Datenclustering sind folgend aufgelistet.

- + Die „round trip time“ (RTT) der RS232-Kommunikation entfällt nur einmalig auf ein Datencluster. Im Vergleich zum Pollen einzelner Daten kann eine höhere Pollfrequenz erreicht werden.
- + Die Verwendung einer CRC-16-Prüfsumme für einzelne Daten erzeugt einen großen Kommunikationsmehraufwand. Eine Prüfsumme für mehrere einzelne

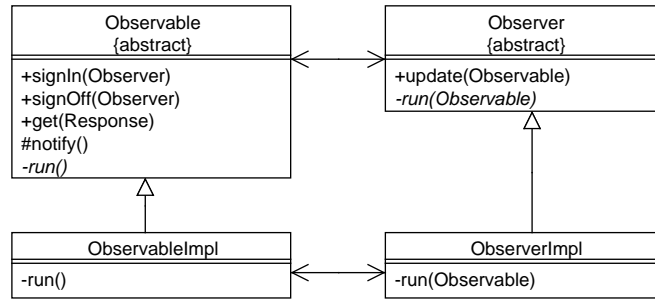


Abbildung 2: UML Klassendiagramm – Threadbasiertes Observer-Pattern [2].

Daten ist komplex umzusetzen. Ein Datencluster kann einfach und mit geringem Mehraufwand mit einer Prüfsumme versehen werden.

- + Eine Datenclustering nach Typ und Pollfrequenz ist möglich.
- Innerhalb eines Datenclusters besteht eine feste Pollfrequenz.
- Die Abfrage einzelner Daten ist nicht gezielt möglich.
- Für verschiedene Beatmungsgeräte und Beatmungsgeräteversionen mit unterschiedlichen Daten sind eigene Cluster nötig.
- Große Cluster benötigen beim Pollen eine größere zusammenhängende RS232-Kommunikationszeitspanne als einzelne Daten. Eine Unterbrechung der Kommunikation durch höherfrequente (kürzere) Abfragen ist nicht möglich.

Besonders die Argumente der geringeren Kommunikationszeitbelegung durch die RTT und die Anwendung einer CRC-16-Prüfsumme haben den Ausschlag für die Verwendung der Datenclustering gegeben. Es wird ein Datencluster für die Alarmer („alarms\r“), die Parameter (Pollbefehl „params\r“), die höherfrequenten Therapiedaten (Pollfrequenz $> \frac{1}{60}$ Hz, Pollbefehl „stats high\r“) und die niederfrequenten Therapiedaten (Pollfrequenz $\leq \frac{1}{60}$ Hz, Pollbefehl „stats low\r“) und die AD-gewandelten Signale verwendet. Für jedes Datencluster gibt es einen eigenen Observable der Klasse VENTIREMOTEObservableSingleTryCover. Die Observable haben Zugriff auf eine Klasse (CRC16, Anforderung 2, 3 und 4) zur Berechnung und Überprüfung der CRC-16-Prüfsumme, die bei der RS232-Kommunikation eingesetzt wird. In der aktuellen Projektversion wird nur ein Observer (VENTIREMOTEObserverEDFPlus) für das Loggen der Beatmungsdaten benötigt. Dieser beinhaltet Klassen zur Umwandlung der Daten in das EDF+ Format und hat Zugriff auf die SD-Karte über die Klasse SDCard (Anforderung 6). Da ein EDF+ Block Daten einer 60 sekündigen Aufzeichnung zusammenfasst, müssen diese Daten zwischengespeichert werden. Verwendet wird dafür eine selbst entworfene Ringpufferklasse nach dem FIFO Prinzip mit Namen „RingBuffer“. Zwischen den Observer-Pattern- und Datenquellenklassen werden Nachrichtenklassen

zur Kommunikation genutzt. Die `VENTIremotePeriodicalRequest` Klasse ist eine Spezialisierung der Anfrageklasse `VENTIremoteRequest` und wird von den Observables verwendet, um die periodischen Pollbefehle weiterzuleiten. Das `VENTIremoteResponse` Interface vereinheitlicht Datenantworten und wird vom Observer genutzt, um Rohdaten vom Observable abzurufen. Jedes Datencluster hat eine eigene Spezialisierung dieser Antwortklasse, die den aufbereiteten Zugriff auf die Rohdaten über Membermethoden bietet.

2.3. Implementierung

Nachdem im vorangegangenen Abschnitt das System des Beatmungsdatenloggers entworfen wurde, fokussiert dieser Abschnitt sich auf Implementierungsdetails und erläutert den abschließenden Implementierungsstand des Beatmungsdatenloggers.

2.3.1. Frameworkeinbindung

Die Anbindung der Hardware erfolgt mit dem STM32 Framework 3.4.0 [15], das in der Vorbereitung der Bachelorarbeit integriert worden ist. Das Framework stellt eine Hardwareschnittstelle bereit, die von dem direkten Registerzugriff auf den Mikrocontroller abstrahiert und die Mikrocontrollerinterrupts über eine Interruptvektortabelle mit C-Funktionen verknüpft. Für das Threadmanagement des Beatmungsdatenloggers wird das Echtzeitbetriebssystem FreeRTOS [9] integriert. Dazu werden die Exceptionhandler aus der „port.c“ Datei anstelle der standard STM32 Framework Handler in die Interruptvektortabelle eingetragen. In der Datei „FreeRTOSConfig.h“ werden Konfigurationen für das Echtzeitbetriebssystem definiert. Das FreeRTOS bietet unter anderem ein Softwareinterrupt-, Thread-, Semaphoren- und Speichermanagement an. Das FreeRTOS ist für die Programmiersprache C ausgelegt.

Es steht kein Äquivalent zur standard C++ Funktionalität `New/Delete` zur Verfügung, sondern nur ein Äquivalent zur C Funktionalität `Malloc/Free` („pvPortMalloc“/„vPortFree“). Die Nutzung der speicherreservierenden C++ `New/Delete` Funktionalität wird auf dem Mikrocontroller nicht unterstützt. Deswegen wird in der Headerdatei „FreeRTOSNewDelete.h“ ein Äquivalent implementiert. Ein Auszug der Datei ist in Abbildung 3 dargestellt. Der template Funktion „`__new`“ und „`__delete`“ wird als erstem Parameter eine Referenz eines Zeigers t auf die zu erzeugende Klasse T übergeben. Durch die Referenzierung kann der Zeiger innerhalb der Funktion modifiziert werden und behält diese Änderung auch nach dem Ende der Funktion bei. Im ersten Schritt des „`new`“ Äquivalents wird über die FreeRTOS Funktion „`pvPortMalloc`“ Speicher in Größe der Klasse T alloziert und dem referenzierten Zeiger t zugewiesen. Im zweiten Schritt wird der Konstruktor der Klasse T über die standard Funktion „`new`“ aufgerufen, ohne dabei Speicher zu allozieren. Als Parameter wird der bereits allozierte Speicherbereich bei t und die Klasse T mit den Konstruktorparametern a_1, \dots, a_n übergeben. Die Konstruktorparameter werden der „`__new`“ Funktion nach dem referenzierten Zeiger t als template Parameter übergeben. Die „`__new`“ Funktion ist für eine verschiedene Anzahl von Konstruktorparametern überladen. Anstelle der Rückgabe

New/Delete Äquivalent – FreeRTOSNewDelete.h

```

140 template <typename T, typename A1, typename A2, typename A3,
      typename A4, typename A5, typename A6, typename A7>
      void _new(T *& t, A1 a1, A2 a2, A3 a3, A4 a4, A5 a5, A6 a6,
      A7 a7) {

142     /* Allocate memory */
143     t = (T *) pvPortMalloc(sizeof(T));
144     if (t == NULL) {
145         throw OutOfMemoryException();
146     }
147
148     /* Call constructor */
149     try {
150         new (t) T(a1, a2, a3, a4, a5, a6, a7);
151     }
152     catch (...) {
153         vPortFree(t);
154         throw;
155     }
156 }

157
158 /**
159  * FreeRTOS version of C++ delete
160  */
161 template <class T> void _delete(T *& t) {
162     if (t != NULL) {
163         /* Call destructor */
164         t->~T();
165
166         /* Free memory */
167         vPortFree(t);
168
169         t = NULL;
170     }
171 }

```

Abbildung 3: Implementierung einer speicherreservierenden C++ New/Delete Äquivalenz mit der FreeRTOS Funktionalität pvPortMalloc/vPortFree.

eines Nullzeigers wird bei fehlerhafter Speicherallokation eine „OutOfMemoryException“ geworfen. Die „_delete“ Funktion ruft den Destruktor der Klasse *T* manuell auf und befreit den Speicherbereich über die FreeRTOS Funktion „vPortFree“, bevor der referenzierte Zeiger *t* zu „NULL“ gesetzt wird. Abbildung 4 zeigt als Beispiel die Erzeugung des EDF+ Observers durch die „_new“ Funktion.

Beispiel einer Objekterzeugung durch „_new“ aus VENTIREMOTE.cpp

```

301  /*** Create observers here ***/
302  VENTIREMOTEObserverEDFPlus * observerEDFPlus = NULL;
303  try {
304      _new<VENTIREMOTEObserverEDFPlus>(observerEDFPlus, "EDFPlus
          observer", 2500, 1);

307  }
308  catch (OutOfMemoryException &) {
309      _delete(observerEDFPlus);
310  }
```

Abbildung 4: Beispiel einer Objekterzeugung durch die „new“ Äquivalenz „_new“ (siehe Abschnitt 2.3.1 und Abbildung 3). Der „_new“ Funktion werden ein Zeiger auf die zu erzeugende Klasse und die Konstruktorparameter übergeben.

2.3.2. Threadmanagement

Beide Observer-Pattern-Elemente erzeugen und starten Threads und besitzen die für Threads typische „run“ und „start“ Methoden (siehe Abschnitt 2.2.2). Das FreeRTOS Framework bietet eine auf die Programmiersprache C ausgelegte Threadverwaltung an. Ein Thread wird über die Funktion „xTaskCreate“ erzeugt und benötigt als Parameter einen Zeiger auf die auszuführende C-Funktion mit „void“ Rückgabetyt und einem Argument vom Typ void-Zeiger. Die „start“ Methoden der Observable- und Observerklasse sind private, nicht-statische Klassenmethoden. Ein Zeiger auf diese Methoden ist inkompatibel zu C-Funktionszeigern und kann durch die „private“ Deklaration nur innerhalb der Klasse verwendet werden. Abbildung 5 zeigt, wie die Threads im Observer (und analog im Observable) erzeugt und gestartet werden. Im Konstruktor wird eine als „friend“ deklarierte C++ Funktion (Zeile 80) an die FreeRTOS Funktion als Zeiger übergeben (Zeile 40). C++ Funktionszeiger sind kompatibel zu C Funktionszeigern. Durch die Deklaration als Friendfunktion kann diese Funktion die private Klassenmethode „start“ aufrufen (Zeile 18). Im Destruktor der Observable- und Observerklasse wird der erzeugte Thread über die FreeRTOS Funktion „vTaskDelete“ beendet.

```

Threaderzeugung – VENTIRemoteObserver::VENTIRemoteObserver(...)
39  /* Create task */
40  if (xTaskCreate(&startTaskVENTIRemoteObserver ,
                 (signed portCHAR *)taskName ,
41                 (unsigned portSHORT)taskStackDeep ,(void *)this ,
42                 (unsigned portBASE_TYPE)taskPriority , &this->xTask)
                 != pdPASS) {
43      throw
          OutOfMemoryException("VENTIRemoteObserver::VENTIRemote
          Observer (...) – Task");
44  }

```

Deklaration der Friendfunktion – class VENTIRemoteObserver

```

77  /**
78   * Friend declaration for task start function
79   */
80  friend void startTaskVENTIRemoteObserver(void * arg);

```

Aufruf der Klassenmethode „start“ – void startTaskVENTIRemoteObserver(void * arg)

```

7  void startTaskVENTIRemoteObserver(void * arg) {
8      /* Cast argument to observer */
9      VENTIRemoteObserver * observer = NULL;
10     try {
11         observer = (VENTIRemoteObserver *)arg;
12     }
13     catch (...) {
14         return;
15     }
16
17     /* Start observer */
18     observer->start();
19 }

```

Abbildung 5: Erzeugung und Start eines Threads im Konstruktor eines Observers. Der Thread führt eine als „friend“ deklarierte Funktion aus, die die private Klassenmethode „start“ aufruft. Threads in Observables werden analog erzeugt und gestartet.

2.3.3. USART Kommunikation

Die USART-Kommunikation mit RS232 ist in der Klasse `VENTIremoteDeviceStubUSART` implementiert. In einer Initialisierungsmethode werden die benötigten Voreinstellungen vorgenommen. Es werden die RS232-Parameter Fullduplex, Baudrate 115200, 8 Datenbits, 1 Stopbit und keine Parität eingestellt. Zudem werden die Hardwareinterrupts für eine abgeschlossene Übertragung (TC, „transmission complete“) und Empfang einer Byteeinheit (RXNE, „receiver not empty“) initialisiert.

Das Empfangen von Daten wird über den RXNE-Interrupt realisiert. Die Verwendung eines Hardwareinterrupts ermöglicht es, sofort auf ein empfangenes Datum zu reagieren. Der interrupt sichere synchrone Zugriff auf die empfangenen Daten erfolgt über eine FreeRTOS-Queue, in die die Daten vom RXNE-Interrupt eingefügt werden. Zudem ist es durch die Verwendung der FreeRTOS-Queue möglich, blockierend mit vorgegebenem Timeout auf den Empfang von Daten zu warten.

Das Versenden der Anfragenachricht von Observables über die RS232-Schnittstelle zum Beatmungsgerät erfolgt über die „request“ Methode der `VENTIremoteDeviceStubUSART` Klasse. Der Zugriff auf die „request“ Methode wird über einen FreeRTOS-Mutex synchronisiert, um die parallele Ausführung durch Observablethreads zu verhindern. Analog zur FreeRTOS-Queue wird das blockierende Warten auf den Mutex mit vorgegebenem Timeout ausgenutzt. Observablethreads mit einer höheren Priorität werden vorrangig dem Mutex zugeteilt. Dadurch lassen sich wichtige (z.B. hochfrequente) Observableanfragen bei der Observableimplementierung bewusst priorisieren. Das Versenden der Anfragenachricht erfolgt über eine STM32 Frameworkfunktion. Dabei wird die Nachricht byteweise in das Versenderegister geschrieben und blockierend auf den TC-Interrupt gewartet. Dadurch wird der aktive Rechenzeitaufwand der „request“ Nachricht minimiert.

2.3.4. Implementierungsprogress

Die Implementierung des Softwaresystementwurfs ist darauf fokussiert worden, eine erste lauffähige Version des Beatmungsdatenloggers zu erzeugen. Die Anbindung der Hardwarekomponenten, das Observer-Pattern und die Generierung des EDF+ Formats sind abgeschlossen. Der abschließende Implementierungsprogress beschränkt sich darauf, die hoch- und niederfrequenten Therapiedaten mitzuloggen. Für diese Daten ist der Beatmungsdatenlogger in Hinblick auf die Anforderungen (keine SD-Karte) funktionsfähig. Aus Zeitgründen sind die Alarm-, Parameter- und AD-Daten nicht implementiert worden. Durch die vorhandenen Interfaces und Vorlagen der Therapiedaten können diese fehlenden Daten unkompliziert nachimplementiert werden. Das Beatmungsdatenloggerprojekt `VENTIremote` hat über 4800 Zeilen selbstgeschriebenen Sourcecode und 24 Klassen. Ein Großteil der Sourcecodezeilen entfällt auf das Observer-Pattern (1145), die EDF+ Formatierung (775), die Hauptfunktion (411), die USART-Kommunikation (454) und die Ringpufferklasse (363). Zudem sind über 750 Zeilen Testsuitecode für die Ringpufferklasse entwickelt worden, um die nachfolgende Mutationsanalyse durchführen zu können.

3. Mutationsanalyse

Die Mutationsanalyse ist eine in den 1970er Jahren vorgeschlagene Softwareanalysetechnik [6]. Sie ist eine fehlerbasierte White-Box-Analysemethode für Softwaremodule, die die Qualität von Tests analysiert und dem Nutzer hilft, Testdaten zu erzeugen und zu optimieren.

Grundgedanke ist die Simulation von Programmierfehlern durch kleine syntaktische Abweichungen des vorliegenden Programmmoduls; zudem ist es möglich, Testadäquanzkriterien [17] (z.B. Code-Coverage) zu integrieren. Ausgehend vom Originalprogrammmodul wird eine Menge von modifizierten Programmmodulen (Mutanten) erzeugt, die jeweils genau eine Mutation repräsentieren. Diese werden durch den zu analysierenden Test überprüft. Weicht das Ergebnis eines Testfalls vom korrespondierenden Ergebnis des Originalprogrammmoduls ab, ist der Mutant entdeckt („killed“, „dead“), andernfalls nicht entdeckt („survived“, „alive“).

Die Mutationsanalyse liefert ein Mutationsadäquanzkriterium („mutation adequacy score“, MAS) für den analysierten Test, welches das Verhältnis von entdeckten „killed“ Mutanten zur Anzahl der detektierbaren Mutanten angibt.

Die Mutationsanalyse ist rechenaufwändig in der Auswertung und beinhaltet manuelle Durchführungsschritte. Veröffentlichungen in den letzten drei Dekaden beschäftigen sich mit Möglichkeiten, den manuellen Aufwand und den Berechnungsaufwand der Mutationsanalyse zu reduzieren, sowie mit der Portierung der Mutationsanalyse in weitere Programmiersprachen und in die Designebene der Softwareentwicklung [6].

3.1. Theorie

Die Analyse der Qualität eines Tests durch die Mutationsanalyse basiert auf zwei grundlegenden Hypothesen.

Der „coupling effect“ besagt, dass ein Test, der so sensitiv (qualitativ hochwertig) ist, dass er alle leicht modifizierten Programme vom Originalprogramm unterscheidet, auch implizit komplexere Fehler erkennt. Komplexe Fehler sind gekoppelt mit einfachen Fehlern [1]. Die von A. J. Offutt für die Mutationsanalyse erweiterte „coupling effect“ und „mutation coupling effect“ Hypothese verknüpft einfache Fehler mit einfachen Mutanten und komplexe Fehler mit komplexen Mutanten. Einfache Mutanten werden durch eine einzige syntaktische Modifikation erzeugt, komplexe Mutanten durch mehrere syntaktische Modifikationen [6, 12, 13].

Die „competent programmer“ Hypothese besagt, dass kompetente Programmierer Software entwickeln, die nahe daran ist, korrekt zu sein [1]. Es wird angenommen, dass eine korrekte Software durch mehrere kleine syntaktische Änderungen erreicht werden kann.

Um eine qualitative Aussage über einen Test zu erhalten, beschränkt sich die Mutationsanalyse darauf, einfache Mutanten zu erzeugen. Mutanten werden über Mutationsoperatoren (Regeln zur syntaktischen Mutation) erzeugt. Die Mutationsoperatoren grenzen die unendliche Menge der einfachen Mutanten für ein Programmmodul auf eine endliche Menge ein. Üblicherweise werden durch die Mutationsoperatoren typische

Programmierfehler simuliert und Testadäquanzkriterien [17] (z.B. Code-Coverage) umgesetzt. Die Qualität der Mutationsanalyse hängt direkt von den verwendeten Mutationsoperatoren ab.

3.1.1. Äquivalenter Mutant

Ein äquivalenter Mutant ist, obwohl syntaktisch modifiziert, funktional äquivalent zum Originalprogramm. Ein äquivalenter Mutant kann somit durch keinen Testfall entdeckt werden. Abbildung 6 zeigt ein Beispiel für einen äquivalenten Mutanten. In Sourcecodezeile 176 ist der „relational operator replacement“ (ROR) Mothramu-

```
Äquivalenter Mutant – RingBuffer::size()
171 unsigned int RingBuffer::size() const {
172
173     if (this->start > this->end) {
174         return ((this->maximumSize - this->start) +
175             this->end);
176
177     /* Original */
178     //else if (this->start == this->end) {
179
180     /* Mutant */
181
182     else if (this->start >= this->end) {
183         if (this->empty) {
184             return 0;
185         }
186         else {
187             return this->maximumSize;
188         }
189     }
190     else {
191         return (this->end - this->start);
192     }
193 }
```

Abbildung 6: Äquivalenter Mutant in C++ erzeugt durch eine adaptierte „relational operator replacement“ Mutation aus Mothra [8]. Die Ersetzung des Vergleichsoperators durch den Größer-Gleich-Operator in Sourcecodezeile 176 ist, aufgrund der Bedingung im IF-Statement in Zeile 173, funktional äquivalent zum Originalprogramm. Der mutierte Sourcecode ist ein Auszug aus der size()-Methode der „RingBuffer“-Klasse des Beatmungsdatenloggers.

tationsoperator angewendet worden. Dadurch ist der Vergleichsoperator durch den Größer-Gleich-Operator ersetzt worden. Alle Fälle, in denen „this->start“ größer als

„this->end“ ist, lassen diese ELSE-IF-Bedingung dadurch zusätzlich wahr werden. Diese Fälle werden aber schon in Sourcecodezeile 173 durch die IF-Bedingung behandelt und erreichen die ELSE-IF-Bedingung nicht. Das mutierte Programm verhält sich identisch zum Originalprogramm, der Mutant ist äquivalent.

Äquivalente Mutanten müssen fast immer manuell erkannt werden. Automatische Testgenerierung kann nicht unterscheiden, ob ein Mutant äquivalent ist oder der Test, der den Mutanten entdeckt, noch nicht generiert worden ist. Eine Möglichkeit einfache äquivalente Mutanten automatisiert zu entdecken ist, diese Mutanten als Compiler(de)optimierung des Originalprogramms nachzuweisen.

3.1.2. Analyseverfahren

Die Mutationsanalyse benötigt den zu analysierenden Test T und das zugrundeliegende Programmmodul P . Im ersten Schritt wird P durch T getestet und korrigiert, bis T für P korrekt ist. Im zweiten Schritt wird ausgehend von P durch die verwendeten Mutationsoperatoren die Menge der Mutanten P^* erstellt. Die Qualität des Tests T wird analysiert, indem alle Mutanten P^* durch T getestet werden. Schlägt ein Testfall aus T für einen Mutanten aus P^* fehl, wird dieser als „dead“ Mutant markiert und aus P^* entfernt. Die in P^* verbliebenen „survived“ Mutanten müssen üblicherweise manuell auf Äquivalenz überprüft werden (Abschnitt 3.1.1). Äquivalente Mutanten werden ebenfalls aus der Menge P^* entfernt. Die verbliebenen „survived“ Mutanten in P^* sind entdeckbar, werden von dem analysierten Test T aber nicht entdeckt. Der Test T kann iterativ optimiert werden, indem gezielt für die Mutanten aus P^* zusätzliche Testfälle generiert werden und die Analyse (ohne Äquivalenzüberprüfung) für die verbliebenen Mutanten aus P^* wiederholt wird. Die Menge P^* wird dadurch um die Menge der zusätzlich entdeckten Mutanten reduziert. Die Mutationsanalyse wird abgeschlossen, wenn alle entdeckbaren Mutanten entdeckt worden sind ($P^* = \emptyset$, $MAS = 100\%$) oder, in einem optimistischen Ansatz [11], wenn ein definierter Prozentsatz der entdeckbaren Mutanten (MAS-Vorgabe) entdeckt worden ist.

3.1.3. Nutzen der Mutationsanalyse

Die Mutationsanalyse ist eine effektive Analysemethode zur Bewertung der Qualität eines Tests durch ein Testadäquanzkriterium. Das Testadäquanzkriterium der Mutationsanalyse heißt „mutation adequacy score“ (MAS) und gibt das Verhältnis von „dead“ Mutanten D zur Anzahl der entdeckbaren Mutanten an (Abschnitt 3). Die Anzahl der entdeckbaren Mutanten setzt sich aus der Anzahl der Mutanten M abzüglich der Anzahl der äquivalenten Mutanten E zusammen (Abschnitt 3.1.1).

$$MAS = \frac{D}{M - E} \quad (3.1.1)$$

Je mehr Prozent der entdeckbaren Mutanten ein Test entdeckt (MAS), desto höher wird die Qualität des Tests bewertet. Ein Test kann bestenfalls alle entdeckbaren Mutanten entdecken und somit eine MAS von 100% aufweisen. Im Vergleich zu anderen

Testadäquanzkriterien zeigt die Mutationsanalyse nicht nur eine ermittelte Testabdeckung auf, sondern zusätzlich die Abwesenheit von Fehlern für die analysierten „dead“ Mutanten.

Die Mutationsanalyse zeigt Optimierungsmöglichkeiten für den Test auf und kann zur Erzeugung von Testfällen genutzt werden (Abschnitt 3.1.2). Bestehende Testfälle können auf ihren Nutzen getestet werden, indem die Anzahl der durch diesen Testfall entdeckten Mutanten ermittelt wird.

Die Mutationsanalyse erzeugt üblicherweise eine große Anzahl an Mutanten. Die manuelle Bestimmung der äquivalenten Mutanten führt daher zu einem großen personellen Arbeitszeitaufwand. Die automatische Erzeugung und das automatisierte Testen der Mutanten führt zu einem großen Berechnungsaufwand [14].

Um die Mutationsanalyse in der Praxis zu etablieren, sind viele Techniken zur Reduzierung des Aufwands aufgezeigt worden, die nachfolgend nach der Anzahl der Veröffentlichungen laut [6] angeordnet sind. Die „selective mutation“ Methode verringert die Anzahl der erzeugten Mutanten, indem die angewendeten Mutationsoperatoren reduziert werden. Mutationsoperatoren erzeugen häufig redundante Mutanten. Durch geschickte Selektion der Mutationsoperatoren wird die Effektivität der Mutationsanalyse nicht signifikant reduziert [14]. Die „firm mutation“ Methode testet die Mutanten an einer einfachen Programmkomponente C_i , die zwischen der mutierten einfachen Komponente C_0 und der durch den Test zuletzt aufgeführten einfachen Komponente C_n liegt. W. E. Howden definiert eine einfache Programmkomponente als „in der Regel korrespondierend zu elementaren Berechnungsstrukturen in einem Programm“ und gibt Variablenreferenzen, Variablenzuweisungen, arithmetische Ausdrücke, relationale Ausdrücke und boolesche Ausdrücke als Typ an [4]. Die Programmausführung wird nach Erreichen der einfachen Programmkomponente C_i abgebrochen. Spezialfälle der „firm mutation“ sind die standard „strong“ Mutationsanalyse für $i = n$ und die „weak mutation“ Methode für $i = 1$. Eine in der „firm mutation“ entdeckte Mutation kann in der standard „strong“ Mutationsanalyse äquivalent sein. Dies führt zu einem geringeren manuellen Aufwand in der Überprüfung der „survived“ Mutanten auf Äquivalenz. Durch die begrenzte Programmausführung beim Testen der Mutanten wird der Berechnungsaufwand reduziert. Weitere Techniken beschäftigen sich mit der Aufwandsreduzierung zur Erzeugung und Ausführung der Mutanten basierend auf Interpreteroptimierung, Zwischensprachen [8], Compileroptimierung und „mutant schemata“ [16]. Verschiedene Ansätze zur automatischen Erkennung von äquivalenten Mutanten reduzieren den manuellen Aufwand der Mutationsanalyse [6].

3.2. Adaption und Analyse

Für die Mutationsanalyse des Beatmungsdatenloggers werden die für Fortran77 entwickelten Mothramutationsoperatoren [8] an C/C++ adaptiert. Aufgrund fehlender verfügbarer Mutationsanalysetools für C/C++ wird die Mutationsanalyse des Beatmungsdatenloggers manuell durchgeführt. Um innerhalb der zur Verfügung stehenden Zeit eine komplette Mutationsanalyse mit den adaptierten Mothramutationsoperatoren durchzuführen, gelten Bedingungen für das zugrundeliegende Programmmodul.

Das Programmmodul muss einen kleinen Umfang aufweisen, eine einfach zu implementierende schnelle Testsuite ermöglichen und Mutanten müssen schnell erzeugt werden können. Für die verwendete eingebettete Umgebung (STM32 Framework [15] & FreeRTOS [9]) kann die zweite Bedingung nicht ohne eine (kostenpflichtige) Toolsoftware realisiert werden, die nicht zur Verfügung steht. Die Testsuite ist mit CppUnit [5] in Eclipse integriert unter Windows realisiert (Anhang C); Mutanten können schnell erzeugt werden. Resultierende Bedingung für das zugrundeliegende Programmmodul ist die Portierbarkeit nach Eclipse unter Windows.

Verwendet wird die „RingBuffer“ Klasse (Anhang B) des in Abschnitt 2 beschriebenen Beatmungsdatenloggers. Im Gegensatz zu den anderen Klassen des Beatmungsdatenloggers ist sie von zentraler Bedeutung und leicht zu portieren, da keine Funktionalität des Mikrocontrollerframeworks und keine über Speicherallokation hinausgehende Funktion des Echtzeitbetriebssystems FreeRTOS genutzt wird.

Die 22 Mothramutationsoperatoren [8] resultieren in 22 adaptierten Mothramutationsoperatoren (Anhang A). Abbildung 7 fasst die original und adaptierten Mothramutationsoperatoren zusammen und erläutert diese kurz. Die Mothramutationsoperatoren DER, DSA und GLR entfallen, die adaptierten Mothramutationsoperatoren NPR, PNR und PPR kommen neu hinzu. Die häufigsten Adaptionsgründe sind Zeiger, Nichtstandardtypen/-klassen, Programmiersprachenkonstrukte, die Syntax und erweiterte Restriktionen zur Vermeidung von syntaktisch inkorrekten, doppelten oder äquivalenten Mutanten. Zeiger und Nichtstandardtypen/-klassen werden von C/C++ unterstützt, nicht jedoch von Fortran77. Die Syntax der Mothramutationsoperatoren muss an die Syntax von C/C++ angepasst werden, soweit es ein äquivalentes Programmiersprachenkonstrukt gibt. Durch erweiterte Restriktionen kann der manuelle Arbeitsaufwand reduziert werden. Abbildung 8 zeigt dafür als Beispiel die Adaption des UOI Mothramutationsoperators. Die adaptierte UOI Restriktion 12 reduziert doppelte Mutanten, die durch den ROR Mutationsoperator erzeugt werden würden. Die adaptierten Restriktionen 13 und 14 verhindern großteils, dass unkomplilierbare Mutanten in zeigerarithmetischen Ausdrücken erzeugt werden. Abbildung 9 fasst die von den Adaptionen betroffenen Mutationsoperatoren zusammen.

Die CppUnit-Testsuite beinhaltet, neben den bei Unittests üblichen „setUp“ und „tearDown“ Methoden, Testmethoden für die Klassenmethoden aus „RingBuffer“. Im ersten Ansatz ist die Testsuite T aus Anhang C als Eingabe für die Mutationsanalyse entwickelt worden. Die „RingBuffer“-Klasse R besteht den Testsuitedurchlauf ohne Fehler. Die 1106 Mutanten für die „RingBuffer“-Klasse R werden manuell nach den adaptierten Mothramutationsoperatoren (ohne die SAN Mutationsoperation) erzeugt. Die Mutationsoperatoren werden nacheinander auf die „RingBuffer“-Klasse angewendet, um, im Vergleich zu einer „RingBuffer“-zeilenweisen Anwendung aller Mutationsoperatoren, einen besseren Überblick in der Mutantenerzeugung zu behalten. Tritt ein Compilerfehler bei der Erzeugung eines Mutanten auf, werden die Restriktionen der adaptierten Mothraregeln erweitert.

Die erzeugten Mutanten werden einzeln durch die Testsuite T überprüft. Alle nicht „killed“ Mutanten werden manuell auf Äquivalenz geprüft. Abbildung 10 zeigt den von allen Mutanten parallel durchlaufenen Analysefluss.

Mutationsoperator		Kurzbeschreibung
Mothra	Adaption	
AAR	AAR	Ersetzung von Arrayreferenzen durch Arrayreferenzen
ABS	ABS	Arithmetische Ausdrücke auf null bzw. NULL überprüfen und durch den Absolutausdruck ersetzen
ACR	ACR	Ersetzung von Konstanten durch Arrayreferenzen
AOR	AOR	Arithmetische Operatoren vertauschen
ASR	AVR	Ersetzung von Variablen durch Arrayreferenzen
CAR	CAR	Ersetzung von Arrayreferenzen durch Konstanten
CNR	CNR	Arraynamen vertauschen
CRP	CRP	Modifizierung von Wertkonstanten
CSR	CVR	Ersetzung von Variablen durch Konstanten
DER	---	Labels in DO-Statements vertauschen
DSA	---	Modifizierung von DATA-Statements
GLR	---	Labels in GOTO-Statements vertauschen
LCR	LCR	Logische Operatoren vertauschen
ROR	ROR	Vergleichsoperatoren vertauschen
RSR	RSR	Einfügen eines RETURN-Statements
SAN	SAN	Statement-Coverage
SAR	VAR	Ersetzung von Arrayreferenzen durch Variablen
SCR	VCR	Ersetzung von Konstanten durch Variablen
SDL	SDL	Entfernen eines Statements
SRC	CCR	Ersetzung von Konstanten durch Konstanten
SVR	VVR	Ersetzung von Variablen durch Variablen
UOI	UOI	Einfügen von unären Operatoren (z.B. Negation, Inkrement, Dekrement, Komplement)
---	NPR	Ersetzung von Zeigern durch NULL
---	PNR	Ersetzung von NULL durch Zeiger
---	PPR	Ersetzung von Zeigern durch Zeiger

Abbildung 7: Übersicht über die original und adaptierten Mothramutationsoperatoren mit kurzer Erläuterung.

	Mothra	Adapted Mothra	Adaptioncause
Abbreviation	UOI	UOI	
Name	Unary operator insertion	Unary operator insertion	
Rule	Each arithmetic expression is negated, incremented by 1 and decremented by 1. Each logical expression is complemented.	Each arithmetic expression is negated, incremented by 1 and decremented by 1. Each logical expression is complemented. Compound assignments are altered to the corresponding assignment using the = operator.	The object orientated C++ language supports compound assignment operators (e.g. +=).
Restrictions	1. An operand following a binary + or – operator is not negated (equivalent to an AOR mutation; e.g. I + (-J) is equivalent to I - J).	1. An operand following a binary + or – operator is not negated (equivalent to an AOR mutation; e.g. "x + (-y)" is equivalent to "x - y").	
	2. An expression is not negated if it is to be used in a multiplication or division (equivalent to other UOI mutations; e.g. (-1) * J is equivalent to -(1 * J)).	2. An expression is not negated if it is to be used in a multiplication or division (equivalent to other UOI mutations; e.g. "(-x) * y" is equivalent to "-(x * y)").	
	3. An expression is not negated if its absolute value is to be computed or it is to be raised to an even power (equivalent to the original program).		There is no build-in function in the C++ language to calculate the absolute value of an expression or to raise an expression to a given value.
	4. An expression is not negated if it is to be tested for equality or inequality with zero (equivalent to the original program).	3. An expression is not negated if it is to be tested for equality or inequality with zero (equivalent to the original program).	
	5. An expression is not negated if the next operation is negation (equivalent to other UOI mutations).	4. An expression is not negated if the next operation is negation (equivalent to other UOI mutations).	
	6. The constant zero is not negated (equivalent to the original program).	5. The constant zero is not negated (equivalent to the original program).	
	7. In an arithmetic IF statement, the expression is not negated if the first and third labels are the same (equivalent to the original program; e.g. IF (1) 10,20, 10 is equivalent to IF (-1) 10,20,10).		There is no equivalent to a Fortran77 arithmetic IF statement in the C++ language.
	8. Integer constants are not incremented or decremented (equivalent to CRP mutations).	6. Constant integer values are not incremented or decremented (equivalent to CRP mutations).	CRP mutation does only effect constant integer values not constant "variables".
	9. An expression is not incremented or decremented if it is to be used in an addition or subtraction (equivalent to another UOI mutation; e.g. (++ 1) + J is equivalent to ++ (1 + J)).	7. An expression is not incremented or decremented if it is to be used in an addition or subtraction (equivalent to another UOI mutation; e.g. "(x + 1) + y" is equivalent to "(x + y) + 1").	
	10. The right operand in a relational expression is not incremented or decremented (equivalent to another UOI mutation; e.g. I.LT.(++J) is equivalent to (- I),LT.J).	8. The right operand in a relational expression is not incremented or decremented (equivalent to another UOI mutation; e.g. "x < (y + 1)" is equivalent to "(x - 1) < y").	
	11. Logical constants are not complemented (equivalent to CRP mutations).	9. Logical constants are not complemented (equivalent to CRP mutations).	
	12. Relational expressions are not complemented (equivalent to ROR mutations; e.g. .NOT. (1. LT.J) is equivalent to I. GEJ).	10. Relational expressions are not complemented (equivalent to ROR mutations; e.g. "{(x < y)" is equivalent to "x >= y"}").	
	13. A logical expression is not complemented if the next operation is complementation (equivalent to another UOI mutation).	11. A logical expression is not complemented if the next operation is complementation (equivalent to another UOI mutation).	
	12. The left operand in a relational expression is not incremented if the relation is >= or <; in addition it is not decremented if the relation is > or <=.	This mutation would be equivalent to a ROR mutation; e.g. "x >= 5" is equivalent to "(x + 1) > 5"	
	13. A pointer expression is not negated.	This mutation would create an illegal program.	
	14. A variable expression is not UOI mutated if the address is used in a (sur)expression.	This mutation would create an illegal program.	

Abbildung 8: Adaption des UOI Mothramutationsoperators.

Adaption	Betroffene Mutationsoperatoren
Erweiterung der Mutationsoperatoren um Zeiger. Fortran77 bietet im Gegensatz zu C/C++ keine Zeiger an.	ABS, NPR, PNR, PPR & UOI
Erweiterung der Mutationsoperatoren um Nichtstandardtypen/-klassen. Fortran77 bietet im Gegensatz zu C/C++ keine Nichtstandardtypen/-klassen an.	AAR, ACR, AVR, CNR, CVR, PPR, VAR, VCR & VVR
Anpassung der Mutationsoperatoren an die C/C++ Syntax.	AAR, ABS, ACR, AOR, AVR, CAR, CCR, CRP, CVR, LCR, ROR, SDL & UOI
Erweiterung und Reduzierung der Mutationsoperatoren um Programmiersprachenkonstrukte. C/C++ und Fortran77 unterstützen Programmiersprachenkonstrukte, die nicht durch äquivalente Konstrukte der anderen Programmiersprache ersetzt werden können.	ABS, AVR, CRP, CVR, DER, DSA, GLR, LCR, ROR, RSR, SAN, SDL & UOI
Erweiterung der Mutationsoperatoren um Restriktionen zur Vermeidung von syntaktisch inkorrekten, doppelten oder äquivalenten Mutanten.	AAR, ACR, AOR, AVR, CAR, CCR, CNR, CRP, CVR, PNR, PPR, ROR, RSR, SDL, UOI, VCR & VVR

Abbildung 9: Zusammenfassung der (Fortran77-)Mothramutationsoperatorenadaption [8] an C/C++. Die vollständige Adaption ist in Anhang A dokumentiert.

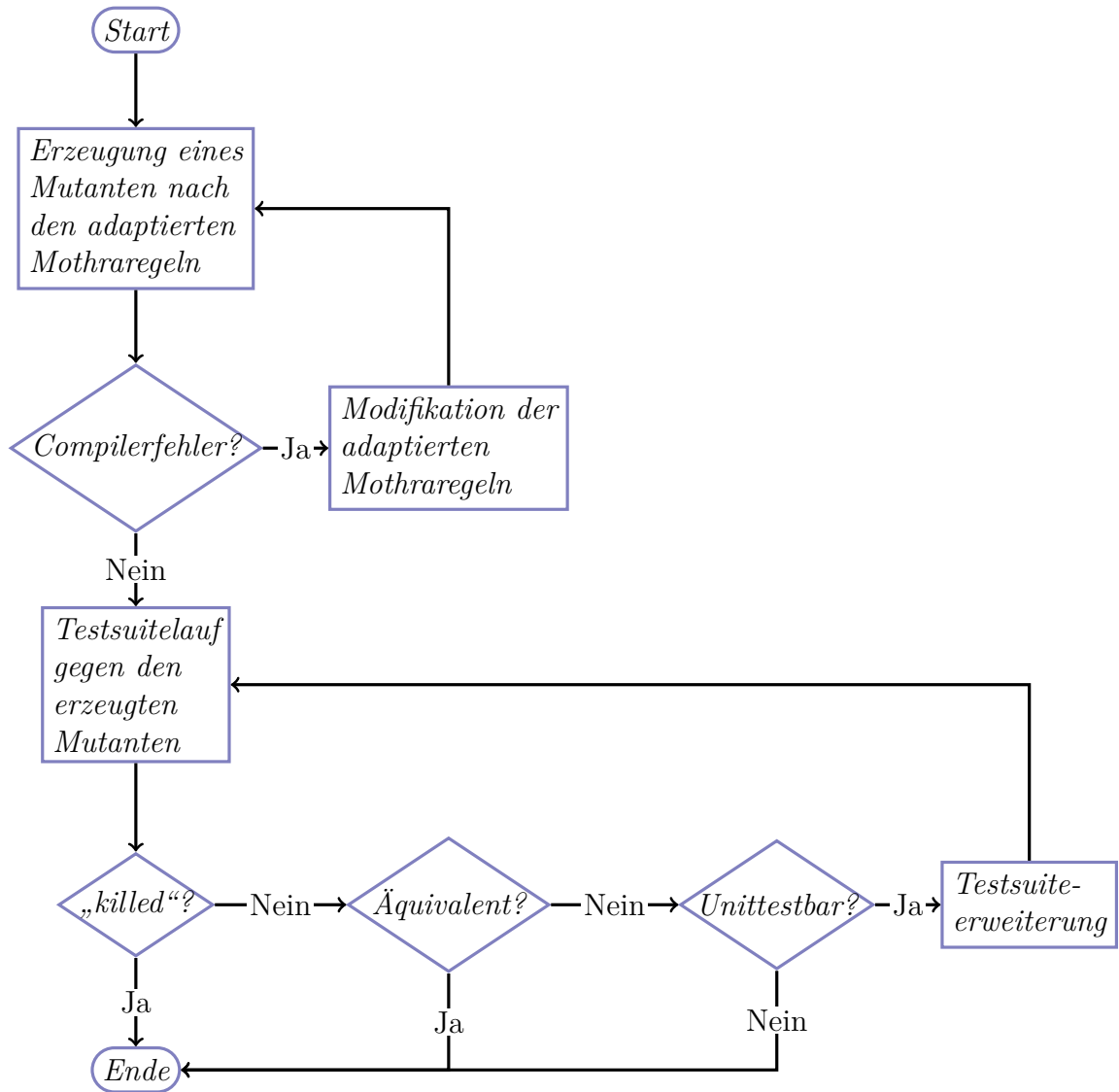


Abbildung 10: Stufenweise parallel für alle Mutanten durchlaufenes Ablaufdiagramm der Mutationsanalyse.

Die original „statement analysis“ (SAN) Mutationsoperation prüft die Ausführung eines Statements am Anfang eines „basic blocks“ und jedes Statement innerhalb eines logischen IF-Statements [8]. Ein „basic block“ ist eine Folge von Statements, die immer alle und in fester Reihenfolge ohne Unterbrechung ausgeführt werden. Diese Mutationsoperation entspricht einer Block-Coverage-Analyse und wird durch das gcc-Compilertool gcov abgedeckt. Das gcov-Tool bietet eine Line-Coverage-Analyse an. Durch den verwendeten Programmierstil (ein Statement pro Zeile) ist diese für die „RingBuffer“-Klasse äquivalent zu einer Block-Coverage-Analyse. Das gcov-Tool wird beim Kompilieren durch die Compiler- und Linkerflags „-fprofile-arcs -ftest-coverage“ bei deaktivierter Compileroptimierung integriert. Das „RingBuffer“-Programm wird einmal mit der Testsuite ausgeführt und über die in Eclipse integrierte „Post-build steps“-Kommandozeile mit dem gcov-Befehl „gcov -l -f RingBuffer.cpp“ ausgewertet.

3.3. Auswertung

Die Mutationsanalyse für die Testsuite T und das „RingBuffer“-Programm P erzeugt 1106 Mutanten, von denen 97 äquivalent und 926 „dead“ sind. Alle aufgetretenen Compilerfehler sind in Restriktionen der adaptierten Mutationsoperatoren (Anhang A) eingeflossen. Die 83 verbliebenen, nicht äquivalenten „survived“ Mutanten zeigen die Optimierungsmöglichkeiten des Tests T auf. Die häufigsten Gründe für Optimierungsmöglichkeiten sind, dass ...

- ... Pufferüberläufe beim Auslesen durch die „peek“ Methode nicht überprüft werden,
- ... Exceptions nur auf ihren Klassentyp und nicht zusätzlich auf ihre Beschreibung geprüft werden,
- ... Speicherlecks, die im Destruktor auftreten, nicht entdeckt werden und
- ... ungünstige Testdaten verwendet werden.

Acht Mutanten sind äquivalent zum Originalprogramm, wenn sichergestellt ist, dass die in der „RingBuffer“-Klasse verwendete Maximalgröße des Speicherbereichs größer als null ist. Bei der Implementierung der „RingBuffer“-Klasse ist diese Bedingung vorausgesetzt, aber nicht umgesetzt worden.

Ausgehend von diesen Auswertungsergebnissen wird das Programm (die „RingBuffer“-Klasse) und die Testsuite verbessert. Die „RingBuffer“-Klasse P wird um eine Überprüfung der Maximalgröße erweitert. Abbildung 11 zeigt die erweiterte „RingBuffer“-Klasse P^+ . Die Testsuite T wird um die Methode „ut_killLivingMutants“ erweitert, die gezielt „survived“ Mutanten des ersten Analysedurchlaufs testet, indem neue Testfälle, resultierend aus dem manuellen Vergleich der „survived“ Mutanten mit dem Originalprogramm, hinzugefügt werden. Diese optimierte Testsuite T^+ entdeckt alle bisher von der Testsuite T entdeckten Mutanten, da alle Testfälle aus T in T^+ enthalten sind. Die Testsuiten T und T^+ liegen als Anhang C bei.

Die Mutationsanalyse wird auf die erweiterte „RingBuffer“-Klasse P^+ angewendet

```

„RingBuffer“-Erweiterung – RingBuffer::RingBuffer(const unsigned int maxSize)
15 RingBuffer::RingBuffer(const unsigned int maxSize) :
16     buffer(NULL),
17     start(0),
18     end(0),
19     maximumSize(maxSize),
20     empty(true) {

21     /* Size check – Mutationanalysis extension*/
22     if (this->maximumSize == 0) {
23         throw
24             IllegalArgumentException("RingBuffer::RingBuffer
25             (...) – RingBuffer max. size has to be greater
26             zero.");
27     }

28     /* Allocate memory */
29     #ifndef UNIT_TEST
30     try {
31         this->buffer = new char[this->maximumSize];
32     }
33     catch (std::bad_alloc &) {
34         throw
35             OutOfMemoryException("RingBuffer::RingBuffer (...)
36             – Buffer allocation failed.");
37     }
38     #else
39     this->buffer = (char *) pvPortMalloc(this->maximumSize);
40     if (this->buffer == NULL) {
41         throw
42             OutOfMemoryException("RingBuffer::RingBuffer (...)
43             – Buffer allocation failed.");
44     }
45     #endif
46 }

```

Abbildung 11: Erweiterung der „RingBuffer“-Klasse um eine Überprüfung der Maximalgröße im Konstruktor. Die Erweiterung ist Resultat der Mutationsanalyse der „RingBuffer“-Klasse mit der (unoptimierten) Testsuite.

und mit der optimierten Testsuite T^+ durchgeführt. Abbildung 12 zeigt die MASs für die Kombinationen der „RingBuffer“-Klassen P und P^+ mit den Testsuiten T und T^+ . Die MAS wird durch die Optimierung der Testsuite in beiden Fällen (P und

	Testsuite	Optimierte Testsuite
„RingBuffer“- Klasse	$MAS_{T,P} = \frac{926}{1106-97} \approx 91,8\%$	$MAS_{T^+,P} = \frac{992}{1106-97} \approx 98,3\%$
Erweiterte „RingBuffer“- Klasse	$MAS_{T,P^+} = \frac{942}{1128-110} \approx 92,5\%$	$MAS_{T^+,P^+} = \frac{1008}{1128-110} \approx 99,0\%$

Abbildung 12: MAS („mutation adequacy score“) [17] der Kombinationen aus Testsuite/optimierter Testsuite und „RingBuffer“-/erweiterter „RingBuffer“-Klasse. Die Optimierung der Testsuite verbessert die MAS um ca. 6,5%. Durch beide aus der Mutationsanalyse abgeleiteten Verbesserungen wird eine MAS von über 99% erreicht.

P^+) um ca. 6,5% verbessert (66 zusätzlich entdeckte Mutanten). Die Erweiterung der „RingBuffer“-Klasse trägt nur mit ca. 0,7% zur Verbesserung der MAS bei (22 zusätzliche Mutanten, von denen 13 äquivalent sind). Für die optimierte Testsuite T^+ mit der erweiterten „RingBuffer“-Klasse P^+ wird eine MAS von über 99% erreicht. Sieben der 10 verbliebenen „survived“ Mutanten sind Speicherlecks durch fehlende Deallokation im Destruktor.

Abbildung 13 zeigt einen Vergleich der Mutationsanalyse der Tests T und T^+ mit der erweiterten „RingBuffer“-Klasse P^+ . Die 22 adaptierten Mothramutationsoperatoren mit Ausnahme von SAN sind absteigend nach der Anzahl der Mutanten sortiert, die die optimierte Testsuite T^+ zusätzlich zur Testsuite T entdeckt. Besonders stark tragen die Variablen- und Konstantenvertauschungsmutationsoperatoren VCR (11), VVR (9) und CCR (6), die Konstantenmodifikationsmutationsoperation CRP (16) und die arithmetischen Operatoren vertauschende Mutationsoperation AOR (11) zur Optimierung der Testsuite bei. Die Anzahl der erzeugten äquivalenten Mutanten für die ABS Mutationsoperation ist mit 54 deutlich höher als für die anderen Mutationsoperatoren mit höchstens 15. Die ABS Mutationsoperation prüft unter anderem, ob eine Variable oder ein Teilstatement den Wert 0 (arithmetisch) bzw. NULL (zeigerarithmetisch) annimmt. Im vorliegenden Programm ist dies nur im rund der Hälfte der Anwendungsfälle (58 von 112) möglich, in den anderen Fällen wird dies durch das Programm explizit ausgeschlossen (äquivalente Mutanten).

Die Verteilung der Anzahl der Mutanten auf die Mutationsoperatoren weicht deutlich von der in [14] vorgestellten Verteilung für 28 Programme ab, bei der die Mothramutationsoperatoren SVR, ASR, SCR, CSR und ACR zusammen ca. 60% der Mutanten erzeugen. Dies spiegelt die Besonderheiten der „RingBuffer“-Klasse wider, die keine Arrayreferenzen verwendet und im Verhältnis zu der Anzahl der verwendeten Konstanten wenig Variablen nutzt. Besonders die für einen prozentual hohen Mutantenanteil

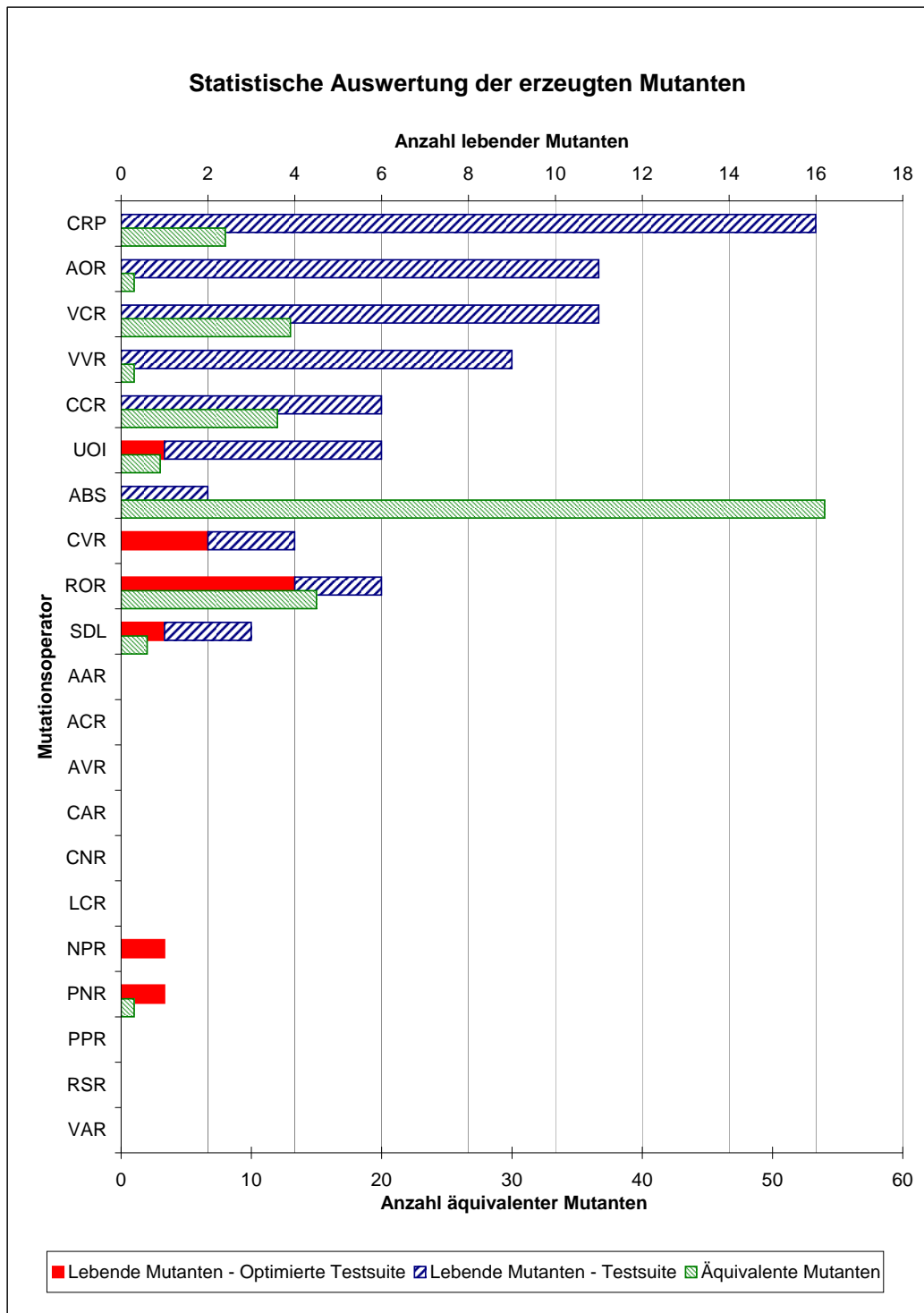


Abbildung 13: Statistische Auswertung der erzeugten Mutanten nach Äquivalenz und Überleben. Die Mutationsoperatoren sind absteigend nach der Anzahl der durch die optimierte Testsuite zusätzlich getöteten Mutanten angeordnet.

verantwortlichen Mutationsoperatoren SVR (VVR), ASR (AVR), ACR, SAR (VAR) und AAR sind deshalb in der „RingBuffer“-Klasse kaum bzw. nicht vertreten. Dieser Umstand wirkt auf die Mutationsanalyse ähnlich einer „selective mutation“.

Die durch das gcov-Tool ausgegliederte SAN Mutationsoperation erreicht mit der optimierten Testsuite auf der erweiterten „RingBuffer“-Klasse eine 100%ige Statement-Coverage-Abdeckung. Abbildung 14 zeigt die gcov-Tool Analyseausgabe mit der Abdeckung für alle Methoden der „RingBuffer“-Klasse. Die Abdeckung der Konstruktor- und Destruktormethode der „RingBuffer“-Klasse beträgt ebenfalls 100%. Dies zeigt die 100%ige Gesamtabdeckung und die detaillierte Ausgabe des gcov-Tools der „RingBuffer.cpp.gcov“ Datei (Anhang D).

```
gcov -l -f RingBuffer.cpp
0.00% of 8 source lines executed in function RingBuffer::RingBuffer(unsigned int)
No executable source lines in function RingBuffer::RingBuffer(unsigned int)
0.00% of 4 source lines executed in function RingBuffer::~~RingBuffer()
No executable source lines in function virtual RingBuffer::~~RingBuffer()
No executable source lines in function virtual RingBuffer::~~RingBuffer()
100.00% of 20 source lines executed in function void RingBuffer::push(const char*, unsigned int)
100.00% of 10 source lines executed in function void RingBuffer::pop(char*, unsigned int)
100.00% of 19 source lines executed in function void RingBuffer::peek(char*, unsigned int, unsigned int)
100.00% of 2 source lines executed in function bool RingBuffer::isEmpty() const
100.00% of 2 source lines executed in function bool RingBuffer::isFull() const
100.00% of 4 source lines executed in function void RingBuffer::flush()
100.00% of 8 source lines executed in function unsigned int RingBuffer::size() const
100.00% of 2 source lines executed in function unsigned int RingBuffer::free() const
100.00% of 2 source lines executed in function unsigned int RingBuffer::maxSize() const
100.00% of 5 source lines executed in function void RingBuffer::moveLabel(unsigned int, unsigned int&)
100.00% of 86 source lines executed in file ../Sources/FIFOBuffer/RingBuffer.cpp
Creating RingBuffer.cpp.gcov.
```

Abbildung 14: Statement-Coverage-Analyseergebnisausgabe des gcc-Compilertools gcov. Die Analyseausgabe ist das Ergebnis des einmaligen Durchlaufs der optimierten Testsuite gegen die erweiterte RingBuffer-Klasse und erreicht eine 100%ige Abdeckung.

4. Zusammenfassung und Ausblick

Für die Entwicklung des Beatmungsdatenloggers sind die Anforderungen an das VENTI-remote-Projekt bestimmt, ein Systementwurf erarbeitet und implementiert worden. Durch die strukturierte Herangehensweise wurde die zur Verfügung stehende Zeit ausgenutzt, so dass der Beatmungsdatenlogger für die hoch- und niederfrequenten Therapiedaten fertig implementiert werden konnte. Explizit ausgenommen davon ist die Anforderung einer SD-Kartenanbindung. Als Hardware wird eine STM32CMICOS-EVAL Evaluationsplatine mit STM32F107VC Mikrocontroller genutzt. Diese bietet alle angeforderten und zu berücksichtigenden Hardwareanschlüsse. Die Software wird aus Modularitäts- und Erweiterbarkeitsgründen objektorientiert in C++ entworfen und beinhaltet als zentrales Softwareelement ein Observer-Pattern. Ein Interface für die beatmungsdatenbereitstellenden Hardwarekomponenten ermöglicht eine spätere Erweiterung der mitgeloggtten Daten. Die Umwandlung der Beatmungsdaten in das vorgegebene EDF+ Format benötigt eine Zwischenspeicherung der Daten für 60 Sekunden. Dies wird durch eine Ringpufferkomponente realisiert. Das STM32 Framework ist in das Projekt eingebunden worden, um die Hardwarekomponenten einfach anzubinden. Zudem ist das FreeRTOS Echtzeitbetriebssystem integriert worden, um eine Threadverwaltungs Umgebung nutzen zu können.

Die Ringpufferkomponente des Beatmungsdatenloggers ist auf Robustheit durch eine Mutationsanalyse untersucht worden. Für den Ringpuffer ist eine Testsuite mit CppUnit entwickelt worden. Über die Mutationsanalyse wurde eine Übersicht und die Theorie erarbeitet. Um die Mutationsanalyse auf das Projekt anwenden zu können, wurden die Mothramutationsoperatoren aus Fortran77 an C++ adaptiert. Zudem sind die Restriktionen der Mothramutationsoperatoren erweitert worden, um compilerfehlerverursachende, doppelte und äquivalente Mutanten zu reduzieren. Da keine Mutationsanalysetools zur Verfügung standen, wurde die Analyse vollständig manuell durchgeführt. Die Auswertung der Mutationsanalyse hat zu einer Erweiterung der Ringpufferkomponente um eine Parameterüberprüfung geführt. Die Testsuite wurde dahingehend optimiert, dass 66 zusätzliche Mutanten entdeckt werden. Die optimierte Testsuite entdeckt über 99% aller entdeckbaren Mutanten („mutation adequacy score“, MAS) des erweiterten Ringpuffers. Eine Analyse der Mutationsoperatoren zeigt im Vergleich, wie effektiv diese die Testsuite optimiert haben und wie viele äquivalente Mutanten durch diese manuell überprüft werden mussten. Zudem ist eine Mutationsoperation durch eine Code-Coverage-Analyse mit gcov durchgeführt worden. Diese erreicht eine Testabdeckung von 100%.

Die Verwendung einer Mutationsanalyse zur Untersuchung der Robustheit des Beatmungsdatenloggers hat gezeigt, dass der Nutzen einer manuellen Analyse in keinem Verhältnis zum Arbeitsaufwand steht. Die Auswertung der Mutationsanalyse hat zwar zu einer Erweiterung der Ringpufferkomponente und Optimierung der zugehörigen Testsuite geführt, der Arbeitszeitaufwand entspricht jedoch der Entwicklungszeit des gesamten Beatmungsdatenloggers. Stellt man die Sourcecodezeilen des Ringpuffers in Relation zu denen des gesamten Beatmungsdatenloggers, entspricht der Testaufwand ca. dem 13-fachen Entwicklungsaufwand.

Eingebettete Komponenten, die nicht ohne weiteres auf einem PC simuliert werden können, erschweren zusätzlich die Erzeugung von Mutanten (jeder Mutant muss auf den Mikrocontroller geflasht werden) und die Entwicklung einer Testsuite. Ein Einsatz einer Mutationsanalyse in anderen Firmenprojekten profitiert davon, dass die Einarbeitung in die Theorie der Mutationsanalyse und die Adaption der Mutationsoperatoren bereits erfolgt ist. Der Arbeitszeitaufwand kann durch den Einsatz einer Mutationsanalysesoftware zudem deutlich verbessert werden. Das Kosten-Nutzen-Verhältnis einer Mutationsanalyse für Firmenprojekte hängt somit stark von der Verfügbarkeit von Mutationsanalyse- und automatischen Testtools ab. Ohne die Unterstützung durch diese Tools ist die Mutationsanalysetechnik für Firmenprojekte ineffizient.

Das Beatmungsdatenloggerprojekt wird nach der Bachelorarbeit in der Firma fortgeführt. Die fehlende Implementierung der Alarme, Parameter und AD-Signale erfolgt, in Hinblick auf die Anfrage-, Antwort- und Observableklasse, analog zu den implementierten hoch- und niederfrequenten Therapiedaten. Zudem wird die Anbindung der SD-Karte über das bestehende Interface implementiert.

Die Untersuchung der Robustheit des Beatmungsdatenloggers wird auf alle Komponenten ausgeweitet werden. Dabei muss beurteilt werden, ob die Anwendung der Mutationsanalyse für diese Komponenten ein akzeptables Kosten-Nutzen-Verhältnis aufweist.

Die adaptierten Mothramutationsoperatoren stehen in einem engen Bezug zu den original Mothramutationsoperatoren. Die Programmiersprache C++ unterscheidet sich jedoch stark von Fortran77. Besonders in Hinblick auf die Unterschiede der Programmiersprachen C++ und Fortran77 (z.B. Objektorientiertheit) können die adaptierten Mutationsoperatoren weiter verbessert und erweitert werden.

Das Projekt wird firmenintern nach der Bachelorarbeit durch eine Entwicklung einer eigenen Hardwareplatine und eine Gehäuseentwicklung begleitet. Im Zuge dessen steht eine Erweiterung des Beatmungsdatenloggers um die in Abschnitt 2.1.2 aufgezeigten zukünftigen Anforderungen in Aussicht.

Literatur

- [1] R. A. DeMillo, R. J. Lipton und F. G. Sayward. „Hints on Test Data Selection: Help for the Practicing Programmer“. In: *IEEE Computer* 11 (4) (Apr. 1978), S. 34–41.
- [2] E. Gamma, R. Helm, R. E. Johnson und J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994, S. 293–303.
- [3] M. Harman, S. Danicic und R. Hierons. „The Relationship Between Program Dependence and Mutation Analysis“. In: *Mutation Testing for the New Century*. 2000, S. 5–13.
- [4] W. E. Howden. „Weak Mutation Testing and Completeness of Test Sets“. In: *IEEE Transactions on Software Engineering* 8 (4) (Juli 1982), S. 371–379.
- [5] Geeknet Inc. *CppUnit Website*. <http://sourceforge.net/projects/cppunit/>. Stand 08. Aug. 2011.
- [6] Y. Jia und M. Harman. „An Analysis and Survey of the Development of Mutation Testing“. In: *IEEE Transactions on Software Engineering* PrePrint (99) (Juni 2010), S. 1–31.
- [7] B. Kemp. *European Data Format Website*. <http://www.edfplus.info/index.html>. Stand 30. Juni 2011.
- [8] K. N. King und A. J. Offutt. „A Fortran Language System for Mutation-based Software Testing“. In: *Softw. Pract. Exper.* 21 (7) (Juli 1991), S. 685–718.
- [9] Real Time Engineers Ltd. *FreeRTOS Website*. <http://www.freertos.org>. Stand 08. Aug. 2011.
- [10] Weinmann Geräte für Medizin GmbH + Co. KG. *Weinmann Website*. <http://www.weinmann.de/>. Stand 08. Aug. 2011.
- [11] A. J. Offutt. „A Practical System for Mutation Testing: Help for the Common Programmer“. In: *Proceedings of the IEEE International Test Conference on TEST: The Next 25 Years*. 1994, S. 824–830.
- [12] A. J. Offutt. „Investigations of the Software Testing Coupling Effect“. In: *ACM Transactions on Software Engineering and Methodology* 1 (1) (Jan. 1992), S. 5–20.
- [13] A. J. Offutt. „The Coupling Effect: Fact or Fiction“. In: *ACM SIGSOFT Software Engineering Notes* 14 (8) (Dez. 1989), S. 131–140.
- [14] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch und C. Zapf. „An Experimental Determination of Sufficient Mutant Operators“. In: *ACM Transactions on Software Engineering and Methodology* 5 (2) (Apr. 1996), S. 99–118.
- [15] STMicroelectronics. *STM32 Microcontroller Website*. <http://www.st.com/internet/mcu/class/1734.jsp>. Stand 08. Aug. 2011.

- [16] R. H. Untch, A. J. Offutt und M. J. Harrold. „Mutation Analysis Using Mutant Schemata“. In: *Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1993, S. 139–148.
- [17] H. Zhu, P. A. V. Hall und J. H. R. May. „Software Unit Test Coverage and Adequacy“. In: *ACM Computing Surveys* 29 (4) (Dez. 1997), S. 366–427.

Anhang

A. Adaption der Mothramutationsoperatoren

Die Dokumentation der (Fortran77-)Mothramutationsoperatorenadaptation befindet sich elektronisch als PDF-Datei unter dem Namen „Anhang_A.pdf“ auf der beiliegenden CD.

B. „RingBuffer“ Klasse

Der Sourcecode der „RingBuffer“-Klassen befindet sich elektronisch im Ordner „Anhang_B“ auf der beiliegenden CD.

C. Testsuite

Die CppUnit-Testsuite für die „RingBuffer“ Klasse befindet sich elektronisch im Ordner „Anhang_C“ auf der beiliegenden CD. Die Methode „void UT_RingBuffer::ut_killLivingMutants(void)“ der optimierten Testsuite beinhaltet die Optimierungstests, die aus der ersten Iteration der Mutationsanalyse des Beatmungsdatenloggers resultieren. Die komplette Testumgebung befindet sich im Ordner „UnitTest“. Der Unterordner „CppUnit“ beinhaltet die statische Bibliothek des CppUnit-Tools. Der Unterordner „VENTIremoteCppUnit“ beinhaltet das Unittest-Eclipse-Projekt. Der Unterordner „Programm“ beinhaltet das CppUnit Programm.

D. Code-Coverage

Die gcov-Tool Analyseausgabe in der „RingBuffer.cpp.gcov“ Datei befindet sich im Ordner „CodeCoverage“. Eine aufbereitete Version liegt als PDF-Datei unter dem Namen „Anhang_D.pdf“ auf der beiliegenden CD vor.

E. Mutationsanalyse

Die Mutationsanalyseergebnisse befinden sich im Ordner „Anhang_E“ auf der beiliegenden CD.

F. Beatmungsdatenloggerprojekt VENTIremote

Das Beatmungsdatenloggerprojekt VENTIremote befindet sich als Atollic True Studio STM32 Projekt im Ordner „VENTIremote“ auf der beiliegenden CD.

G. Verwendete Soft- und Hardware

Diese Bachelorarbeit verwendet die folgende Hard- und Software.

PC Intel Core i5 3,33Ghz; 3,49 GB RAM; Windows XP Professional 2002 Service Pack 3

Mikrocontroller Entwicklungsumgebung Atollic True Studio STM32 Pro 2.1.0

Mikrocontroller Framework STM32F10x_StdPeriph_Lib_V3.4.0

Mikrocontroller RTOS FreeRTOSV6.1.1

Mutationsanalyse Entwicklungsumgebung Eclipse Galileo; MinGW GCC Toolchain 3.1.0

Sonstige Software Office 2007