TUHH
Technische Universität Hamburg-Harburg

Diplomarbeit

Tobias Kaupat

Reverse Engineering und Analyse eines auf RS-232 basierenden Protokolls zur Reimplementierung

19.07.2012

supervised by:
1. Examiner: Frau Prof. Dr. Sibylle Schupp (TUHH)
2. Examiner: Prof. Dr.-Ing. Andreas Timm-Giel (TUHH)
Supervisor: Oliver Langer (Olympus W&I)

Hamburg University of Technology (TUHH)
Technische Universität Hamburg-Harburg
Institute for Software Systems
21073 Hamburg

TUHH STS Software Technology Systems

Sperrvermerk

Zu der vorliegenden Arbeit existiert eine weitere Version, die vertrauliche Inhalte der Firma Olympus Winter & Ibe GmbH enthält. Vertrauliche Stellen der Arbeit sind durch einen entsprechenden Kommentar markiert. Einsicht in die vertrauliche Version der Arbeit bedarf einer schriftlichen Genehmigung der Firma Olympus Winter & Ibe GmbH.

Erklärung gemäß Diplomprüfungsordnung

Ich erkläre an Eides Statt, dass ich die Diplomarbeit mit dem Titel „Reverse Engineering und Analyse eines auf RS-232 basierenden Protokolls zur Reimplementierung“ selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und alle den benutzten Quellen wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Hamburg, 20. Juli 2012

_____

Tobias Kaupat

Danksagung

An dieser Stelle möchte ich mich bei all den Menschen bedanken, die mich bei der Erstellung dieser Arbeit unterstützt haben. Ich danke besonders meiner Betreuerin Frau Prof. Dr. Sibylle Schupp, die es mir ermöglicht hat, den Kontakt zu Olympus herzustellen. Ihr Feedback war immer sehr hilfreich. Ein besonderer Dank gilt auch Oliver Langer, der mich bei Olympus während der gesamten Dauer meiner Diplomarbeit betreut hat und sich viel Zeit für mich genommen hat. Während der Zusammenarbeit in den sechs Monaten habe ich viele neue Erfahrungen gesammelt. Außerdem danke ich meinen Eltern Margit und Wolfgang für die Hilfe beim Korrekturlesen und meiner Freundin Julia für den seelischen Beistand.

Abstract

Im Systemverbund von medizinischen und nicht-medizinischen Geräten werden Kommunikationsprotokolle zur Integration von Teilsystemen verwendet. Zur Sicherstellung der Funktionalitäten darf eine Reimplementierung des Kommunikationsprotokolls die eigentlichen Steuerungs- und Benachrichtigungsfunktionen zwischen den existierenden Systemen nicht verändern. Modellierungsverfahren unterstützen die Sicherstellung der korrekten Funktionsweise. Ziel dieser Diplomarbeit ist es, auf Basis einer existierenden Spezifikation und Implementierung, ein Kommunikationsprotokoll zwischen einem medizinischen und einem nicht-medizinischen System modellgestützt zu verifizieren. Dabei wird das in natürlicher Sprache spezifizierte Protokoll in Computational Tree Logic (CTL) formalisiert und mit Hilfe des Tools Uppaal verifiziert. Anschließend wird das verifizierte Protokoll implementiert und die Implementierung mit Hilfe des erarbeiteten Modells getestet. Für die Tests auf Basis des verifizierten Modells, wird ein neuer Ansatz entwickelt und in einem Framework implementiert. Das Framework stellt Annotationen in C# bereit, um die Beziehung zwischen dem Kontrollfluss des Programms und dem verifizierten Modell herzustellen. Auf Basis der Annotationen können Abweichungen zwischen Modell und Implementierung aufgedeckt werden.

Inhaltsverzeichnis

Teil I.

Einleitung

Im Rahmen der Systemintegration bei Olympus wird ein Kommunikationsprotokoll zwischen zwei medizinischen Geräten verifiziert und reimplementiert. Zudem wird ein neuer Ansatz vorgestellt, um die Reimplementierung basierend auf den Ergebnissen der Verifikation zu testen.

Bei einem chirurgischen Eingriff am Menschen werden heutzutage verschiedene technische Geräte und Systeme genutzt. Geräte und Systeme, die direkt am Patienten zum Einsatz kommen, wie zum Beispiel Generatoren für Schneidwerkzeuge und Pumpen, unterliegen hohen Sicherheitsstandards, sogenannten Risikoklassen. Andere Systeme und Geräte im Operationssaal, wie zum Beispiel Videokonferenzsysteme, Dokumentationssysteme und Audiosteuerungen sind weniger sicherheitskritisch. Die Systemintegration beschäftigt sich mit der Integration medizinischer und nichtmedizinischer Geräte und Systeme in einem krankenhausweiten Gesamtsystem. Die Firma Olympus entwickelt zur Integration von Medizingeräten ein Komplettpaket bestehend aus zwei Komponenten: einem medizinischem Controller, genannt UCES, sowie einem nicht-medizinischem Controller, genannt VMC.

Eine technische Trennung von medizinischem und nicht-medizinischem Controller ist aus mehreren Gründen notwendig. Zum einen können beide Systeme unabhängig voneinander entwickelt und eingesetzt werden, zum anderen unterliegen die Systeme unterschiedlichen Risikoklassen. Der Aufwand für die Entwicklung eines medizinischen Controllers ist deutlich höher als für einen nicht-medizinischen Controller, weil Fehler im medizinischem Controller Verletzungen oder im schlimmsten Falle den Tod eines Patienten zur Folge haben können. Die Entwicklung ist bei Olympus auf die Standorte Hamburg für den VMC und Japan für die UCES aufgeteilt.

Trotz der technischen Trennung beider Systeme soll das Gesamtsystem für den Benutzer transparent über eine einzelne Schnittstelle bedienbar sein. Dies erfordert einen Datenaustausch zwischen UCES und VMC. Hierfür wird eine serielle Schnittstelle verwendet, über welche Daten mit einem von Olympus entwickelten Kommunikationsprotokoll ausgetauscht werden. Die Kommunikation zwischen UCES und VMC ist ein besonders kritischer Teil des Systems. Um die Sicherheit des Patienten zu garantieren, ist das Protokoll speziell für die Kommunikation zwischen UCES und VMC entworfen und stellt unter anderem sicher, dass der VMC nur eine passive Rolle spielt und nicht aktiv auf medizinische Geräte zugreifen kann. Für eine neue Softwareversion des VMC soll das Kommunikationsprotokoll neu implementiert werden.

Die Arbeit beschäftigt sich daher zuerst mit dem Reverse Engineering des Kommunikationsprotokolls zwischen UCES und VMC. Reverse Engineering ist ein

Prozess, der das betrachtete System analysiert, um seine Komponenten und ihre Beziehungen zu identifizieren. Zweck dieses Prozesses ist es, für das analysierte System neue Abbildungen zu schaffen, meist in anderer Form und auf höherer Abstraktionsebene [CCI90].

Reverse Engineering ist somit der erste Arbeitsschritt im Reengineering-Prozess. Reengineering ist ein Prozess, um vorhandene Software zu verbessern oder zu verändern, so dass sie einfacher verstanden, gewartet und wieder verwendet werden kann. Grundsätzlich kommt Reengineering dann zum Einsatz, wenn die vorhandene Software teuer in der Wartung ist oder die Architektur nicht mehr zeitgemäß ist. Die Software wird dann mit Hilfe aktueller Software und/oder Hardwaretechnologie angepasst oder neu geschrieben [RH08]. Im gesamten Prozess können auch neue Kundenanforderungen berücksichtigt werden. Eine Reimplementierung des Protokolls ist im Rahmen einer neuen Softwareversion für den VMC von Olympus geplant und dient als konkretes Anwendungsbeispiel für das durchgeführte Reengineering und die Verifikation.

In sicherheitskritischen Bereichen wie der Medizintechnik reicht das Testen von Software nicht aus, um die Funktionalität von Softwarekomponenten sicherzustellen. Mit Hilfe von Methoden der Softwareverifizierung können Eigenschaften von Systemen bereits beim Systementwurf sichergestellt werden [CAB+98]. Zur Verifikation eines Protokolls werden eine Protokolldefinition und Aussagen benötigt. Aussagen beschreiben hierbei die Eigenschaften der Protokollausführung [Gou93]. Das Ergebnis der Verifikation gibt an, ob die gegebenen Aussagen tatsächlich der Protokolldefinition entsprechen.

Zur Analyse des Kommunikationsprotokolls zwischen UCES und VMC wird zunächst die vorliegende informelle Spezifikation betrachtet, welche die sicherheitsrelevanten und technischen Anforderungen an das Protokoll festlegt. Da automatisierte Techniken zur Protokollverifikation eingesetzt werden, ist es notwendig, dass die Spezifikation von einer Maschine verstanden werden kann. Dafür wird eine formale Spezifikation in Form eines Modells auf Basis der informellen Spezifikation und der vorliegenden Implementierung erstellt. Zu verifizierende Eigenschaften des Protokolls werden formal in temporaler Logik beschrieben. Mit Hilfe von entsprechenden Werkzeugen kann die Gültigkeit der Formeln im erstellten Modell nachgewiesen werden und die Spezifikation verifiziert werden. Fehler im Protokolldesign und in der Spezifikation können so aufgedeckt und bereits vor der Implementierung gezielt behoben werden. Das Modell und die Ergebnisse der Verifikation unterstützen weiterhin die Entwicklung der Reimplementierung des Protokolls.

Die Spezifikation umfasst 28 Aussagen, die mit 53 Formeln beschrieben werden.

Für fünf der Aussagen lassen sich keine Formeln finden. Die Arbeit diskutiert die Gültigkeit der Aussagen und ob das Protokoll den erwarteten Anforderungen genügt. Dabei werden die Formeln auf einem Modell bestehend aus zwei Echtzeitautomaten verifiziert. Je ein Automat bildet einen Kommunikationspartner ab.

Neben dem Protokolldesign, welches in dieser Arbeit analysiert und verifiziert wird, soll auch garantiert werden, dass die Reimplementierung fehlerfrei ist und der Protokollspezifikation entspricht. An die Reimplementierung werden verschiedene Anforderungen gestellt. Zum einen soll die Protokollimplementierung an die serviceorientierte Projektstruktur der VMC Software angepasst sein, zum anderen muss sie hohen Sicherheitsanforderungen genügen.

Die Reimplementierung muss allen Aussagen genügen, die auf dem Modell verifiziert sind. Dabei kann das Modell dem Entwickler sowohl während der Implementierung als auch bei der Testerstellung unterstützten. Eine formale Beziehung zwischen Modell und Quellcode besteht jedoch nicht. Um diese Beziehung herzustellen, ist weiterer Aufwand notwendig. Es werden verschiedene Ansätze vorgestellt, die eine Beziehung zwischen Modell und Implementierung herstellen.

Für das Protokoll zwischen UCES und VMC wird ein neuer Ansatz entwickelt, der eine formale Beziehung zwischen Modell und Implementierung herstellt und sich in dieser Form von bestehenden Ansätzen unterscheidet. Dabei liefert das Modell die Rahmenbedingungen für die Ausführungsreihenfolge des Programms. Der Ansatz wird in Form eines Testframeworks realisiert. Mit dem Framework wird die Implementierung auf Basis des verifizierten Modells getestet und Abweichungen der Implementierung vom Modell werden erkannt.

Die Arbeit gibt Einblick in Methoden der Protokollverifizierung und verifiziert, welche Eigenschaften das Protokoll zwischen UCES und VMC erfüllt. Es wird eine Reimplementierung des Protokolls erstellt, die in der nächsten Version des VMC genutzt wird. Dabei wird das entwickelte Testframework eingesetzt, um die Reimplementierung auf Basis des verifizierten Modells zu testen. Das entwickelte Framework bietet neue Möglichkeiten für Softwaretests, die auch für andere Projekte hilfreich sein können.

Kapitel 1 behandelt die Thematik der Protokollverifikation. Kapitel 2 beschreibt das zur Modellierung und Verifikation verwendete Tool Uppaal. Kapitel 3 stellt verschiedene Ansätze zur modellgestützten Softwareverifikation vor. Kapitel 4 beschreibt das betrachtete System mit UCES und VMC. Kapitel 5 behandelt die Modellierung und Verifizierung der Protokollspezifikation. In Kapitel 6 wird die Reimplementierung des Protokolls beschrieben. Kapitel 7 geht auf verschiedene

Ansätze ein, wie eine Relation zwischen Modell und Implementierung hergestellt werden kann und stellt den neuen Ansatz vor, um die Implementierung mit Hilfe des verifizierten Modells testen zu können. Kapitel 8 fasst die Ergebnisse und während der Arbeit aufgetretenen Probleme zusammen und Kapitel 9 gibt einen Ausblick für die weitere Nutzung und Erforschung der erarbeiteten Ergebnisse.

Teil II.

Grundlagen und Methodik

1. Protokollverifikation

Die Protokollverifikation beschäftigt sich mit dem formalen Nachweis der Korrektheit von Kommunikationsprotokollen. In dieser Arbeit wird ein Kommunikationsprotokoll aus der Medizintechnik verifiziert, welches hohe Anforderungen an das Risikomanagement stellt, da Fehlfunktionen im schlimmsten Fall den Tod eines Patienten zur Folge haben.

Verifikation ist die „Bestätigung durch Bereitstellung eines objektiven Nachweises, dass festgelegte Anforderungen erfüllt worden sind“ [ISO9000]. Gerade in der Informatik ist die Verifikation von Software ein viel diskutiertes Forschungsthema [GQ11]. Trotz intensiver Forschungen auf dem Gebiet, werden Verifikationsmethoden in der Praxis oft vernachlässigt. In dieser Arbeit wird dies zum Anlass genommen, um Methoden und Tools zur Verifikation im medizinischen Umfeld zu evaluieren und anzuwenden. Im Speziellen ist das Ziel, die Kommunikation zwischen einem medizinischen und einem nicht-medizinischem Controller zu verifizieren. Dabei muss sowohl die Spezifikation verifiziert werden als auch deren Implementierung.

Viele Kommunikationsprotokolle sind Echtzeitsysteme, da das Verhalten der Protokolle oft abhängig von Timeouts oder Übertragungszeiten ist. Gegenüber der Verifikation von zeitunabhängigen Systemen stellt die Zeitkomponente viele neue Herausforderungen bei der Verifikation. Eine Übersicht von Modelltypen, Spezifikationssprachen, Verifikationsframeworks und Zustandsraum-Repräsentationen von Echtzeitsystemen liefert Farn Wang [Wan04].

Die Protokollverifikation kann als Funktion mit zwei Eingabeparametern und einer Ausgabe gesehen werden. Eingaben sind zum einen die Spezifikation und zum anderen Anforderungen an das Protokoll [Gou93]. Die Spezifikation wird in einem Modell abgebildet und muss gegebene Aussagen erfüllen. Die Aussagen werden zum Teil aus der Spezifikation abgeleitet oder sind allgemeine Kriterien, wie z. B. kein Auftreten eines Deadlocks, und werden als Formeln in einer formalen Logik formuliert. Die Ausgabe gibt für jede Formel an, ob sie im Modell gilt und damit, ob die Aussage der Formel auf die Spezifikation zutrifft. Ist eine Formel im Modell nicht gültig, kann in einigen Fällen ein Gegenbeispiel generiert werden.

Das Vorgehen bei nicht verifizierten Formeln liegt im Ermessen des Entwicklers. Die Verifikation kann so Fehler in der Spezifikation, dem Modell oder den Formeln aufdecken. Die Verifikation kann auch Missstände in den Anforderungen aufdecken, wenn sich z. B. zwei Anforderungen ganz oder teilweise widersprechen.

Die zur Verifikation benötigte formale Spezifikation des Protokolls wird manuell aus einer informellen Protokollspezifikation erstellt und erfordert ein gutes Verständnis des zu modellierenden Systems. Zusammen mit den Anforderungen, welche auf einem gegebenen Modell verifizierbar sind, ergibt sich ein formales Abbild des modellierten Systems. Die Schwierigkeit besteht darin, alle Aspekte der informellen Spezifikation korrekt auf das Modell zu übertragen. Für die Übersetzung von formalen Sprachen in natürliche Sprachen wie Englisch gibt es wissenschaftliche Studien [Joh05]. Auch wenn es für einige informelle Systembeschreibungen wie z. B. Use-Cases Diagramme Möglichkeiten gibt, eine Übersetzung in eine formale Form zu automatisieren [GH04], ist die automatische Übersetzung heute nicht auf beliebige informelle Spezifikationen anwendbar. Daher muss die Spezifikation manuell formalisiert werden.

Zur Verifikation stehen verschiedene Modellformen und damit verschiedene Tools zur Verfügung. Je nach Tool variieren auch die unterstützten Logiken, die zur Formulierung der Aussagen genutzt werden können. Kapitel 1.1 beschreibt die für diese Arbeit relevante Modellform. Kapitel 1.2 beschreibt die Grundlagen für die in dieser Arbeit eingesetzte Logik.

1.1. Modellformen

Formale Verifikation basiert auf formalen Logiken. In dieser Arbeit wird temporale Logik [BM97] verwendet, um Systemeigenschaften zu beschreiben. Die verwendete Logik definiert die Grammatik (Syntax) und die Bedeutung (Semantik). Eine Grammatik kann verschiedene Bedeutungen haben. Die Bedeutung einer logischen Formel ist über eine Menge von Modellen definiert. Ein Modell definiert das Verhalten einer Systembeschreibung (oder Spezifikation). Abhängig vom verwendeten Framework kann ein Modell zum Beispiel eine Menge von Zuständen, eine Zustandssequenz, eine Ereignissequenz oder einen Zustandsbaum beschreiben [Wan04]. Es stehen Tools zu Verfügung, die unterschiedliche Frameworks und damit verschiedene Modellformen implementieren. Einige der gängigen Modellformen, mit denen sich Echtzeitsysteme beschreiben lassen, sind Echtzeitautomaten, Petri-Netze und logische Relationsmodelle.

Petri-Netze werden z. B. vom Modelchecker Romeo [GLMR05] verwendet. Petri-Netze sind eine spezielle Form von Zustandsautomaten. Jedoch sind viele spezifische Eigenschaften von Petri-Netzen für diese Arbeit nicht notwendig und führen zu weniger übersichtlichen Modellen ohne Mehrwert gegenüber klassischen Zustandsautomaten. Von der Verwendung von logischen Relationsmodellen, wie sie z. B. im Modelchecker Spin [Hol97] verwendet werden, wird abgesehen, da eine Modellierung von Zuständen und Zustandsübergängen hier nur implizit möglich ist. In dieser Arbeit werden daher Echtzeitautomaten zur Modellierung verwendet. Diese erfüllen alle Anforderungen, bieten eine anschauliche Form der Darstellung von Kommunikationsprozessen und haben eine gute Toolunterstützung. Das verwendete Tool Uppaal wird in Kapitel 2 näher beschrieben.

Echtzeitautomaten Die in dieser Arbeit verwendeten Modelle sind eine Komposition von Echtzeitautomaten. Ein Automat wird durch ein 6-Tupel definiert: (L, l_0, C, A, E, I). Dabei ist L eine Menge von Positionen mit der initialen Position $l_0 \in L$, C ist eine Menge von Uhren, A ist eine Menge von Aktionen, $E \subseteq L \times A \times B(C) \times 2^C \times L$ ist eine Menge von Kanten zwischen Positionen mit einer Aktion, einer Bedingung und einer Menge von Uhren, die zurückgesetzt werden. $I : L \to B(C)$ weist Invarianten einer Position zu [GB06].

Automaten lassen sich grafisch darstellen. Dabei werden Positionen L als Kreise und Kanten E als Pfeile abgebildet. Transitionen und Positionen können annotiert werden. Die Annotationen beschreiben sowohl die Invarianten von Positionen als auch die Aktionen und Übergangsbedingungen von Kanten.

Zustandsübergänge werden über Transitionen realisiert. Eine Transition ist an Bedingungen geknüpft, die erfüllt sein müssen, damit ein Zustandsübergang möglich ist. Man spricht bei einem Zustandsübergang vom „Feuern“ der Transition. Zustände im Automaten werden im Folgenden nach den Positionen benannt, auf denen sich der Automat befindet. Wird keine Aussage über die zum Zustand gehörenden Variablen gemacht, sind diese beliebig.

Ein Beispiel eines Automaten ist in Abbildung 1.1 gegeben. Der Automat zählt eine Variable `i` hoch und kann den Wert von `i` auf 0 zurücksetzen, falls `i > 0`. Ob und wann der Wert von `i` zurückgesetzt wird, ist nicht festgelegt.

Die Menge der Positionen ist $L = \{\texttt{Init}, \texttt{Inc}, \texttt{Reset}\}$ mit dem Anfangszustand $l_0 =$`Init`. Die Menge der Uhren ist die leere Menge $C = \emptyset$. Die Aktionen A sind durch Annotationen an den Kanten gegeben: $A = \{(\texttt{i = 0}), (\texttt{i++})\}$. Bedingungen sind ebenfalls Annotationen an Kanten: $B(C) = \{(\texttt{i > 0})\}$. E beinhaltet die Kan-

ten zwischen den Positionen mit den entsprechenden Aktionen und Bedingungen. Die Kanten sind:

$$\{(\texttt{Init}, \texttt{Inc}), (\texttt{Inc}, \texttt{Init}), (\texttt{Init}, \texttt{Reset}), (\texttt{Reset}, \texttt{Init})\}$$

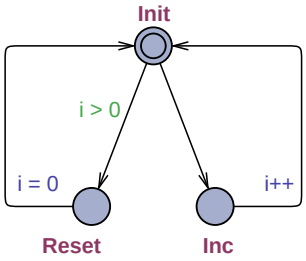


Abbildung 1.1.: Beispiel eines Automaten

Ein solcher Automat kann für die Modellierung von Zuständen, Zustandsübergängen und Echtzeiteigenschaften von Protokollen eingesetzt werden. Er stellt dann die Spezifikation des Protokolls dar, auf der Eigenschaften des Protokoll verifiziert werden können. Um die Eigenschaften zu verifizieren, müssen sie z. B. in temporaler Logik formalisiert werden.

1.2. Temporale Logiken

Zur formalen Beschreibung von Anforderungen an Kommunikationsprotokolle wird eine formale Beschreibungssprache benötigt. Temporale Logiken sind eine Form für die formale Beschreibung der Anforderungen auf einem gegebenen Modell. Temporale Eigenschaften wie Sendereihenfolgen von Nachrichten können beschrieben und an einem Modell verifiziert werden. Mit temporaler Logik werden keine zeitlichen Abläufe beschrieben, sondern Zustände und deren Änderungen in Systemabläufen. Es wird im Wesentlichen zwischen zwei temporalen Logiken unterschieden: Linear Time Logic (LTL) und Computational Tree Logic (CTL). Beide sind Erweiterungen der Aussagenlogik. LTL und CTL besitzen eine gemeinsame Teilmenge. Es existiert eine Obermenge von LTL und CTL, die als CTL* bezeichnet wird.

CTL ist die Grundlage für die in dieser Arbeit verwendete Logik. CTL Formeln geben quantitative Aussagen über Pfade, die von einem gegebenen Zustand erreichbar sind. Dafür werden neben den bekannten logischen Operatoren wie „Nicht“, „Und“, „Oder“, „Implikation“ und „Äquivalenz“ weitere Temporaloperatoren eingeführt, um logische Aussagen auf Folgen von Zuständen anwenden zu können. In einer

CTL Formel werden zunächst Eigenschaften durch logische Operatoren formuliert, die auf einer Menge von Zuständen gelten sollen. Anschließend muss quantifiziert werden, auf welche Zustände oder Zustandsfolgen die Eigenschaften zutreffen. Dafür werden die folgenden einstelligen Temporaloperatoren eingeführt: F oder $\Diamond$ für *einen irgendwann folgenden Zustand* und G oder $\Box$ für *alle folgenden Zustände*. Die CTL definiert noch weitere einstellige und auch zweistellige temporale Operatoren, die für diese Arbeit keine Relevanz haben und hier daher nicht beschrieben werden.

Für den Automaten in Abbildung 1.2 kann nun z. B. die Formel $\Diamond$`i` $==$ 1 definiert werden, die aussagt, dass irgendwann `i` den Wert 1 annimmt. Dabei beschreibt die Formel nicht, auf welchem Pfad ein Zustand mit `i` $==$ 1 angenommen werden soll und gehört nicht zur Klasse der CTL Formeln. Hierfür werden noch weitere Operatoren benötigt, die Pfadquantoren. Dabei steht E für *entlang (mindestens) eines Pfades* und A für *entlang aller Pfade*. In einer CTL Formel wird jedem Temporaloperator ein Pfadquantor vorangestellt. Die Formel aus dem Beispiel kann so zu folgender CTL Formel erweitert werden: $E\Diamond$ `i` $==$ 1, es existiert ein Pfad, auf dem irgendwann `i` $==$ 1 gilt. Die Formel gilt auf dem Automaten aus Abbildung 1.2, da der Pfad (`Start`, `Loop`, `Inc`, `Loop`) die Variable `i` von 0 auf 1 inkrementiert.

Ein anderes Beispiel ist die Formel $E\Diamond$ `End` $\wedge$ `i` $==$ 4. Entlang eines Pfades existiert ein Zustand, in dem gilt, der Name des Zustands ist `End` und `i` $==$ 4. Diese Formel wird von dem Automaten in Abbildung 1.2 nicht erfüllt, da `i` nur inkrementiert wird, wenn `i` $<$ 3 ist. `i++` gilt, also nicht für `i >= 3`. Somit wird `i` nie größer 3.

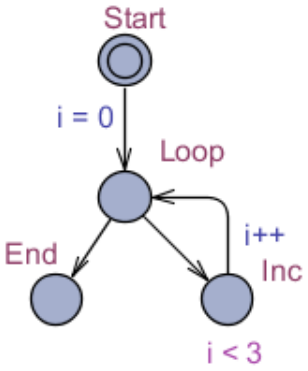


Abbildung 1.2.: Beispielautomat

2. Werkzeug zur Verifikation: Uppaal

Für die Verifikation von Kommunikationsprotokollen gibt es eine Vielzahl verschiedener Tools, die verschiedene Modellformen und Logiken unterstützen. Für diese Arbeit wird Uppaal [GB06] zur Modellierung und Verifizierung des Kommunikationsprotokolls genutzt, da es die erforderten Anforderungen erfüllt.

Zur Protokollverifikation müssen in den meisten Fällen, wie auch in dieser Arbeit, Echtzeiteigenschaften modelliert werden. Eine wichtige Anforderung an Tools zur Protokollverifizierung ist daher die Unterstützung von synchronisierten Uhren zur Modellierung von Echtzeitverhalten.

Uppaal ist ein Tool zur Modellierung, Simulation und Verifizierung von Netzwerken aus Echtzeitautomaten. Die Beschreibungssprache erlaubt es, abstrakte Zustandsmaschinen zu modellieren, welche über Handshake-Synchronisation und gemeinsame Variablen kommunizieren. Eines der Designziele von Uppaal ist Effektivität in Bezug auf Speicherverbrauch und Zeit, so wurde Uppaal bereits für eine Vielzahl von praktischen Anwendungsfällen erfolgreich eingesetzt [BGK+02, BFK+98].

In Kapitel 2.1 werden zuerst die einzelnen Softwarekomponenten von Uppaal vorgestellt. Anschließend wird in Kapitel 2.2 auf die Möglichkeiten zur Modellierung eingegangen und eine Übersicht der grafischen Oberfläche gegeben. Die von Uppaal zur Verifizierung unterstützte Timed Computational Tree Logic (TCTL) wird am Ende in Kapitel 2.3 näher beschrieben.

2.1. Softwarekomponenten

Uppaal besteht aus drei Teilen und ist als Client/Server-Architektur aufgebaut. Eine grafische Oberfläche dient als Benutzerschnittstelle zum Modellieren und Verifizieren. Die eigentliche Verifikation findet im Server statt. Die Anbindung der

grafischen Oberfläche ist über eine Javabibliothek realisiert. Das Uppaal beiliegende Programm „verifyta“ bietet zudem eine Kommandozeilenversion zur Verifikation.

Die grafische Oberfläche von Uppaal ist in Java geschrieben, der Server ist für verschiedene Plattformen kompiliert und sowohl für Windows als auch Linux verfügbar.

Der Kern von Uppaal ist der Server. Mit dem Server können TCTL-Formeln auf einem Modell verifiziert werden. Eingabedaten sind ein Modell und eine Menge von TCTL-Formeln. Als Ausgabe wird für jede Formel entschieden, ob sie auf dem gegebenen Modell gültig ist. Kann eine Formel nicht verifiziert werden, liefert der Server als Gegenbeispiel eine Ausführungsreihenfolge des Automaten, welche im integrierten Simulator nachvollzogen werden kann. Verschiedene Optionen können gesetzt werden, um die Verifizierung zu steuern. So kann z. B. angegeben werden, ob Gegenbeispiele generiert werden und wie das Modell bei der Verifizierung traversiert wird. Eine genaue Beschreibung aller Optionen ist der Uppaal beiliegenden Hilfe zu entnehmen [GB06].

Uppaal liegt eine Javabibliothek bei, mit der die Serverschnittstelle zum Verifizieren und Simulieren von Modellen angesteuert werden kann. Der Server bietet die Möglichkeit, Informationen über alle Zustände, Variablen und Transitionen eines Modells abzufragen und TCTL-Formeln zu verifizieren. Der Server wird vom Simulator und dem Verifizierer in der grafischen Oberfläche genutzt. Es ist möglich, den Server lokal oder im Netzwerk zu betreiben.

Die grafische Oberfläche von Uppaal besteht aus drei Teilen: einem Editor, einem Simulator und einem Verifizierer. Der Editor bietet eine grafische Oberfläche zum Erstellen von Modellen und einen Texteditor zum Editieren der Modelldeklarationen. Im Simulator lassen sich die erstellten Automaten schrittweise manuell ausführen. Alle Variablen und Werte der Automaten sind einsehbar und es wird parallel zur Ausführung ein Sequenzdiagramm erstellt. Die Gegenbeispiele vom Server können im Simulator grafisch nachvollzogen werden. Der Verifizierer bietet eine Eingabemaske für TCTL-Formeln und nutzt den Server zur Verifizierung der Formeln.

2.2. Modelle in Uppaal

Mit Uppaal können Modelle erstellt werden. Die Beschreibungssprache und grafische Darstellung für Modelle wird von Uppaal vorgegeben. Die Modelle sind in XML-

Dateien beschrieben und können mit dem grafischen Editor in Uppaal erstellt und editiert werden. Ein Modell besteht in Uppaal aus einem oder mehreren Templates für Automaten, einer Systemdeklaration und einer Modelldeklaration. In der Systemdeklaration werden Instanzen von Automaten aus den Templates erstellt. In der Modelldeklaration werden globale Variablen und Funktionen definiert.

Jedes Template beschreibt einen Automaten und kann innerhalb des Modells beliebig oft instanziiert werden. Bei der Instanziierung können Parameter zur Konfiguration übergeben werden. Jedes Template besteht aus der Beschreibung des Automaten und einer Templatedeklaration. Die Templatedeklaration gleicht der Modelldeklaration, nur dass alle hier deklarierten Funktionen und Variablen lokal für jede Instanz des Templates gültig sind. Die Beschreibungssprache für Variablen und Funktionen in den Deklarationen hat eine C-ähnliche Syntax.

In den Deklarationen können Variablen und Funktionen definiert werden. Die genaue Syntax ist der Uppaal Hilfe zu entnehmen [GB06]. Es stehen Datentypen für Ganzzahlen (`int`), Wahrheitswerte (`bool`), Channels (`chan`) und Uhren (`clock`) zur Verfügung.

Uhren dienen zum Modellieren von Zeit. Eine Variable vom Typ `clock` bildet Zeit diskret ab. Alle Uhren im System laufen synchron vorwärts. Es ist möglich, einer Uhr einen Wert zuzuweisen, ohne andere Uhren zu beeinflussen.

Jeder Automat besteht aus Positionen und Transitionen, wobei für jede Transition *Guard*, *Update*, *Sync* und *Select* Ausdrücke angegeben werden können. Positionen können *Invarianten* haben. Abbildung 2.1 zeigt Beispiele für alle Elemente in Automaten.

Positionen sind als Kreise dargestellt, Transitionen als Pfeile. Eine Farbe für Positionen und Transitionen kann frei gewählt werden und hat keinen Einfluss die Semantik des Automaten. Die verschiedenen Ausdrücke können durch ihre Färbung unterschieden werden. Eine Übersicht der Annotationen und Farben findet sich in Tabelle 2.1. Im folgenden wird die Bedeutung der verschiedenen Annotationen erklärt.

Guard ist ein Boolscher Ausdruck, der angibt, ob eine Transition gefeuert werden kann. Abbildung 2.1a zeigt einen Automaten mit zwei Guards. Die Transition von `CheckMessage` zu `MessageOkay` kann nur genommen werden, wenn `msg.Error` nicht wahr ist.

Update ist ein Ausdruck, der angibt, welche Variablen sich wie ändern, wenn die Transition gefeuert wird. Abbildung 2.1b erweitert den Automaten aus Abbildung

Annotation	Farbe
Guard	grün
Update	dunkel blau
Sync	hell blau
Select	gelb
Zustandsinvariante	lila
Zustandsnamen	dunkel rot

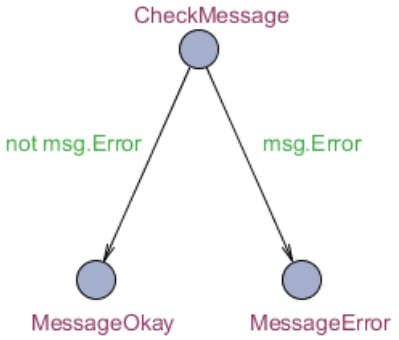
Tabelle 2.1.: Annotationen an Zuständen und Transitionen in Uppaal

2.1a, um eine Transition zum Zustand `Error`. Die Transition hat das Update-Statement `LineEst = false`, welches die Variable `LineEst` aktualisiert, wenn die Transition gefeuert wird.

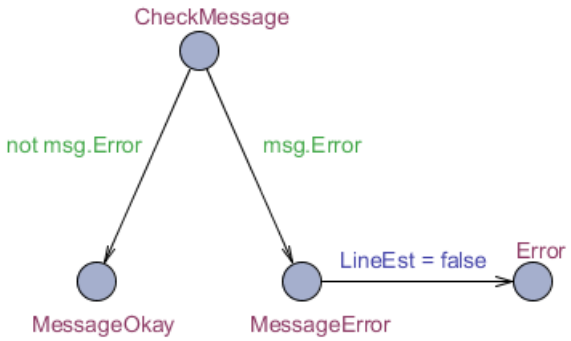
Sync dient zur Synchronisation zwischen einem oder mehreren Automaten. Zur Synchronisation zweier Kanten im Modell wird für jede Kante ein Channel angegeben und ein Fragezeichen oder ein Ausrufezeichen angehängt. Dabei symbolisiert ein Ausrufezeichen den Sender und ein Fragezeichen den Empfänger. Abbildung 2.1c zeigt ein Modell aus zwei Automaten, die über den Channel `Ack` synchronisiert sind. `Ack!` benachrichtigt den Channel und `Ack?` wartet auf eine Benachrichtigung. Wird die Transition von `MessageReceived` nach `AckSend` gefeuert, wird der Channel `Ack` benachrichtigt und damit zeitgleich die Transition im zweiten Automaten gefeuert.

Mittels *Select* können zufällige Werte aus Sets und Arrays gewählt werden. Abbildung 2.1d zeigt einen Automaten mit dem *Select* Statement `s:int[0,1]`, welches der temporären Variable `s` einen Wert zwischen 0 bis 1 zuweist. Der zufällig gewählte Wert wird der Variablen `i` zugewiesen und hat Einfluss auf die mögliche Folgetransition.

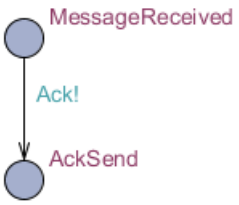
Invarianten für Zustände definieren Bedingungen, die erfüllt sein müssen, während sich der Automat im entsprechen Zustand befindet. Mögliche Transitionen zum Zustand `Idle` in Abbildung 2.1e können somit nur gefeuert werden, wenn der Wert `t` kleiner als `timeout` ist, und der Zustand muss verlassen werden, bevor die Bedingung verletzt wird. Invarianten werden vor und nach jedem Feuern einer Transition überprüft.

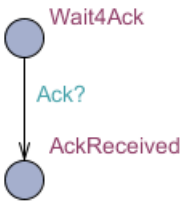


(a) Uppaal Guards

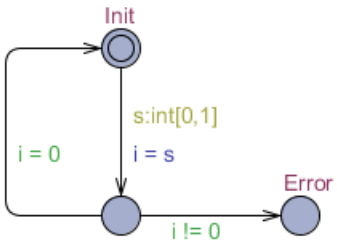


(b) Uppaal Update

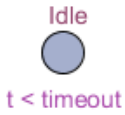




(c) Uppaal Syncronisations Channels



(d) Uppaal Select



(e) Uppaal Zustand mit Invariante

Abbildung 2.1.: Beispiele für Annotationen in Uppaal

2.3. Uppaal TCTL

Timed Computational Tree Logic (TCTL) ist die von Uppaal unterstützte Logik [GB06]. TCTL ist eine Erweiterung der Computational Tree Logic (CTL) um symbolische Zeit. Das Konzept der Zeit ist durch die Unterstützung von Uhren in den Modellen gegeben und wird beim Evaluieren von TCTL-Formeln berücksichtigt. Uhrvariablen können Bestandteil von TCTL-Formeln sein.

TCTL-Formeln können schnell komplex werden, so dass sie auf heutigen Rechnern

nicht mit angemessenen Zeit- und Speicherressourcen verifizierbar sind. Um eine gewissen Effizienz zu garantieren ist in Uppaal nur eine Teilmenge der TCTL implementiert, welches die Operatoren $A\Box$, $A\Diamond$, $E\Box$, $E\Diamond$ und $\rightsquigarrow$ unterstützt, aber keine Verschachtelung zulässt. Die unterstützen Operatoren sind Teil der CTL (siehe Kapitel 1.2) und stehen in Tabelle 2.2. Abbildung 2.2 zeigt beispielhaft die Aussagen von TCTL-Formeln an einem Automaten.

In den Formeln werden Positionen und Variablen der Automaten referenziert. Eine Referenz auf eine Position ist immer dann wahr, wenn sich das Modell auf der entsprechenden Position im Automaten befindet. Im Bezug auf Abbildung 2.1d wäre die Formel $E\Diamond$ `Error` also gültig, wenn es einen Pfad gibt, der in den Zustand `Error` führt.

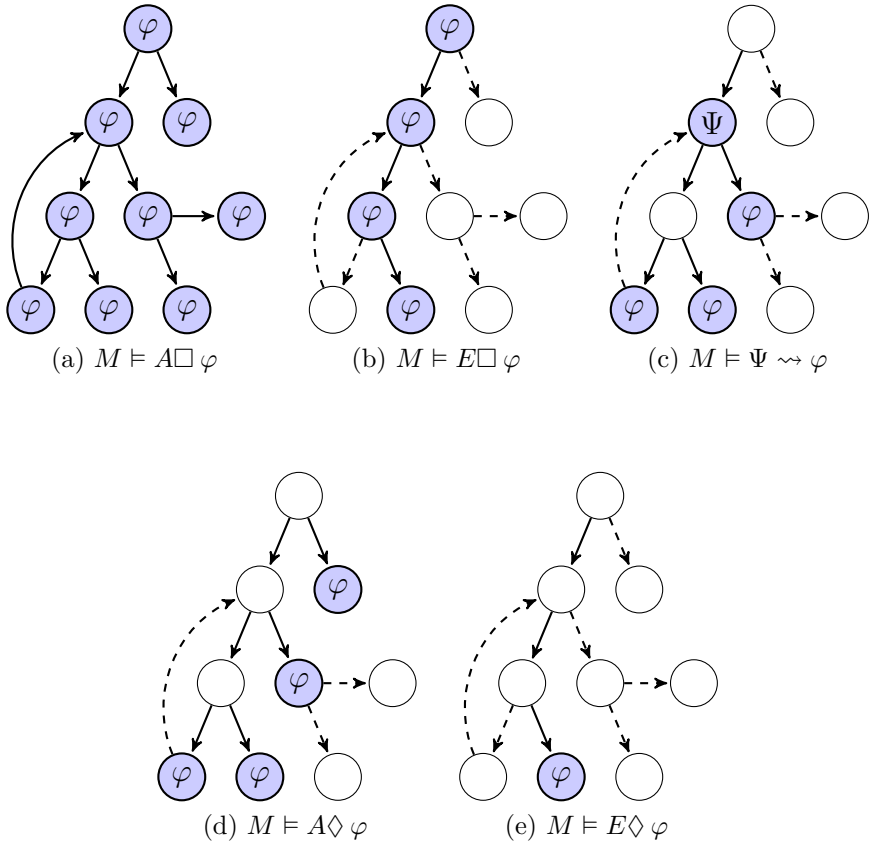
Beschreibung	**Eigenschaft**		**Äquivalent zu**
	Uppaal	Formal	
Möglich	`E<> p`	$E\Diamond\, p$	
Invariant für alle Pfade	`A[] p`	$A\Box\, p$	`not E<> not p`
Invariant für einen Pfad	`E[] p`	$E\Box\, p$	
Irgendwann	`A<> p`	$A\Diamond\, p$	`not E[] not p`
Führt zu	`p --> q`	$p \rightsquigarrow q$	`A[] (p imply A<> q)`

Tabelle 2.2.: Unterstützte TCTL-Formeln in Uppaal

Mit Hilfe der TCTL-Formeln kann z. B. die Aussage, dass Während des Line Connection Prozesses keine Nachricht mit dem Command `ChangeNotice` gesendet werden darf (siehe Spezifikation **??** in Anhang A.1) als folgende Formel formalisiert werden:

$$A\Box \texttt{ not (UCES.GotNotice and not UCES.lineEst)}$$

Wenn `UCES.lineEst = false` ist, besteht keine Verbindung und es darf keine `ChangeNotice` gesendet werden. Die Formel stellt also für das gesamte Modell ($A\Box$) sicher, dass die UCES nur eine `ChangeNotice` erhält (`UCES.GotNotice`), wenn die Verbindung hergestellt ist (`UCES.lineEst = true`).



(a) $M \vDash A\Box\,\varphi$ (b) $M \vDash E\Box\,\varphi$ (c) $M \vDash \Psi \rightsquigarrow \varphi$

(d) $M \vDash A\Diamond\,\varphi$ (e) $M \vDash E\Diamond\,\varphi$

Abbildung 2.2.: TCTL Aussagen über den Zustandsgraphen eines Modells M

3. Modellgestützte Softwareverifikation

In diesem Kapitel werden verschiedene Ansätze zur Softwareverifikation vorgestellt. Bei der Verifikation soll ein objektiver Nachweis gegeben werden, dass festgelegte Eigenschaften von der Implementierung erfüllt werden (siehe Kapitel 1). Die Eigenschaften des Protokolls sind bereits formal an einem Modell nachgewiesen. Die Implementierung in Form von Quellcode kann ebenfalls als eine Modellierung der Spezifikation angesehen werden. Es muss also nachgewiesen werden, dass beide Welten, die Modellwelt und die Quellcodewelt, dieselben spezifizierten Eigenschaften erfüllen. Die vorgestellten Ansätze lassen sich nicht oder nur mit großem Aufwand auf die Protokollimplementierung des VMC anwenden. Am Ende wird erläutert, weshalb ein eigener Ansatz zum Testen der Implementierung entwickelt wird.

T. Leavens et al. zeigen Ansätze zur Verifikation von objektorientierten Programmen mit den Tools JML für Java und Spec# für C# [BLS04, LLM07]. Dabei werden zu verifizierende Aussagen direkt in den Programmcode eingebettet. Auch wenn sich viele Vor- und Nachbedingungen bereits formulieren lassen, stehen der vollständigen Verifikation von Quellcode mit diesem Ansatz noch viele Hindernisse im Weg, für die nur Teillösungen oder theoretische Ansätze existieren, wie in [LLM07] beschrieben.

Die Verifikation von abstrakten Modellen ist oftmals einfacher als die Verifikation von Programmcode. Corbett et al. stellen einen Ansatz vor, mit dem sich ein Modell direkt aus Java-Quellcode generieren lässt [CDH+00]. Anschließend kann eine Verifikation an dem generierten Modell vorgenommen werden. Das vorgestellte Tool Bandera kann realistische, aber limitierte Klassen von Javaprogrammen handhaben. Ein solcher Ansatz scheint praktikabler als die direkte Verifizierung von Quellcode. Dennoch ist die Methode limitiert, was die Form und den Inhalt der erzeugten Modelle angeht. Für diese Arbeit wird Uppaal verwendet, um das Protokoll zu modellieren und verifizieren. Bandera erzeugt Modelle, die nur schwer mit Uppaalmodellen vergleichbar sind. Besonders die Zeitkomponenten der Uppaalmodelle lassen sich nicht einfach auf Bandera Modelle übertragen.

Die bisher genannten Ansätze berücksichtigen nicht das entwickelte Modell des Kommunikationsprotokolls (siehe Kapitel 5.1). Ein weiterer Ansatz, um eine Implementierung zu verifizieren besteht darin, den Quellcode aus dem Modell zu generieren. So können Eigenschaften des Modells direkt im Programm abgebildet werden. Es gibt unter dem Begriff des „Model Driven Development“ viele erfolgreiche Ansätze, um Quellcode aus Modellen zu generieren [CH03, Gor11]. Dabei werden Tools angewendet, die ein Modell transformieren und als Programmcode abbilden. Hierbei besteht ein entscheidender Nachteil darin, dass die Modelltransformation die Quellcodestruktur vorgibt. Entwickler müssen mit den Tools zur Modelltransformation vertraut sein und ein Editieren des Quellcodes ist nur über das Modell möglich. Zudem ist ein solcher Ansatz nicht geeignet, wenn die zu verifizierende Implementierung wie hier bereits vorliegt. Für Uppaal sind bisher keine Tools vorhanden, um ein Modell in Programmcode zu transformieren.

Sterkin stellt die Methode des „State-Oriented Programming“ vor [Ste08]. Dabei zeigt er die Schwierigkeit, Zustandsautomaten in objektorientierten Programmiersprachen abzubilden und stellt eine eigene Sprache zur Programmierung von Zustandsautomaten vor. Es wird jedoch keine Möglichkeit vorgestellt, ein solches Programm formal mit einem Modell abzugleichen.

Bei der Protokollimplementierung im VMC ist es gewünscht, übliche Programmiermethodiken beizubehalten und die Struktur der Software frei vorgeben zu können. Es wird daher ein Ansatz gesucht, mit dem vorhandener Quellcode mit einem gegebenen Modell abgeglichen werden kann. Ein praktikabler Ansatz wird von Murphy et al. beschrieben, bei dem mittels Softwarereflexionsmodell Software Reflection Modellen die Beziehung zwischen Quellcode und Modellen hergestellt wird [MNS95]. Dort gibt der Programmierer ein Modell vor, von dem er erwartet, dass es der Implementierung entspricht. Das Software Reflection Modell zeigt Übereinstimmungen und Abweichungen zwischen dem gegebenen Modell und dem Quellcode. Dabei besteht das gegebene Modell aus Blöcken und Kanten. Den Blöcken werden Teile (Quellcodedateien) des Programms zugewiesen, und das erzeugte Software Reflection Modell gibt für jede Kante an, ob die Kante auch im Quellcode als möglicher Kontrollfluss vorhanden ist. Weiterhin zeigt das Reflection Modell alle Kanten, die im gegebenen Modell nicht existieren, aber einem möglichen Kontrollfluss im Programm entsprechen. Die Methode eignet sich für Programmierer, um Software zu verstehen, indem ein erwartetes Modell in mehreren Iterationen mit dem Programmcode abgeglichen wird. Für die Verifikation des Programmcodes anhand eines gegebenen Modells muss eine Zuordnung von Modellblöcken zu Quellcode jedoch genauer sein als auf Dateiebene.

Für die VMC-Implementierung muss der Quellcode dem Modell der Spezifikation

genügen. Statt wie bei Murphy et al. Relationen zwischen Elementen des Modells und Softwarekomponenten herzustellen, wird ein neuer Ansatz entwickelt, bei dem Zustände des Modells direkt mit Zeilen und Variablen im Quellcode in Beziehung gebracht werden. Unterschiede zwischen Modell und Implementierung können so einfach zur Laufzeit erkannt werden. Obwohl eine Verifikation der Implementierung wünschenswert ist, wird der neue Ansatz zunächst mit dem Ziel entwickelt, die Implementierung basierend auf einem verifizierten Modell zu testen. Das Modell wird in Kapitel 5.1 beschreiben und das entwickelte Testframework wird in Kapitel 7 beschrieben.

4. Betrachtetes System

In diesem Kapitel wird das System mit allen für die Arbeit relevanten Teilen beschrieben. Das betrachtete System kommt im Operationssaal zum Einsatz. Eine wichtige Anforderung ist, dass Geräte effizient genutzt werden können, um Zeit zu sparen. Das hier beschriebene System wird mit dem Ziel entworfen, dem operierenden Arzt eine zentralisierte Schnittstelle zur technischen Ausstattung im sterilen Bereich zu bieten. Die Kombination aus einem medizinischen Controller und einem nicht-medizinischem Controller kann über einen einzelnen Touch Panel (TP) bedient werden und bietet dem Arzt eine Möglichkeit, schnell verschiedene Geräte und Systeme zu bedienen, ohne sie zu berühren. Dies führt zu Zeitersparnis während der Operation und bei einer Sterilisation der Bedienelemente muss nur ein Bildschirm zusätzlich desinfiziert werden.

Kapitel 4.1 beschreibt den Systemaufbau, bestehend aus verschiedenen Hardwarekomponenten, und die Aufgaben der einzelnen Komponenten. Kapitel 4.2 gibt eine informelle Beschreibung des Kommunikationsprotokolls zwischen medizinischem und nicht-medizinischem Controller.

4.1. Systemaufbau

Die Arbeit beschäftigt sich mit einem Kommunikationsprotokoll im medizinischen Umfeld. Dieses Kapitel beschreibt den Systemaufbau der beteiligten Hardwarekomponenten. Das System besteht aus drei Komponenten: als medizinischer Controller dient die UCES, als nicht-medizinischer Controller der VMC und zur Ein- /Ausgabe wird ein TP verwendet. Weiterhin gibt es eine serielle Datenverbindung zwischen UCES und VMC und einen Schalter für den TP. Der TP kann so als Benutzerschnittstelle für beide Geräte verwendet werden und durch den Schalter zwischen UCES und VMC wechseln. Die UCES ist die Steuereinheit für chirurgische Ausrüstung und steuert z. B. Endoskope, Pumpen und Spannungen von Werkzeugen. Der VMC dient als Steuerung für die Peripherie und kann z. B. das Raumlicht steuern, bietet Zugriff auf Dokumentationssoftware und Videokonferenzen. So kann

eine Operation live aus dem Operationssaal in einen Schulungsraum übertragen werden, oder der Arzt kann auf Knopfdruck eine Videotelefonie mit einem Kollegen im Büro starten. Abbildung 4.1 zeigt den beschriebenen Systemaufbau.

Grund für die Aufteilung der Steuerung auf zwei Geräte sind die unterschiedlichen Anforderungen an das Risikomanagement der Systeme. Risiko bezieht sich in diesem Kontext auf die möglichen Folgen am Patienten bei Fehlfunktionen und der damit verbundenen Einstufung der Geräte in eine Risikoklasse. Die Steuerung von chirurgischer Ausrüstung fällt unter eine höhere Risikoklasse und unterliegt strengeren Vorschriften als die Steuerung der Peripherie. Die Risikoklasse eines Gerätes hat direkten Einfluss auf den Entwicklungsaufwand und die damit verbundene Kosten. Für die serielle Schnittstelle zwischen den Geräten hat die Risikostufe ebenfalls einen Bedeutung. Beim Protokolldesign wird daher darauf geachtet, dass der VMC die UCES nicht aktiv beeinflussen kann. Die UCES agiert als Master und der VMC folgt als Slave den Anweisungen der UCES.

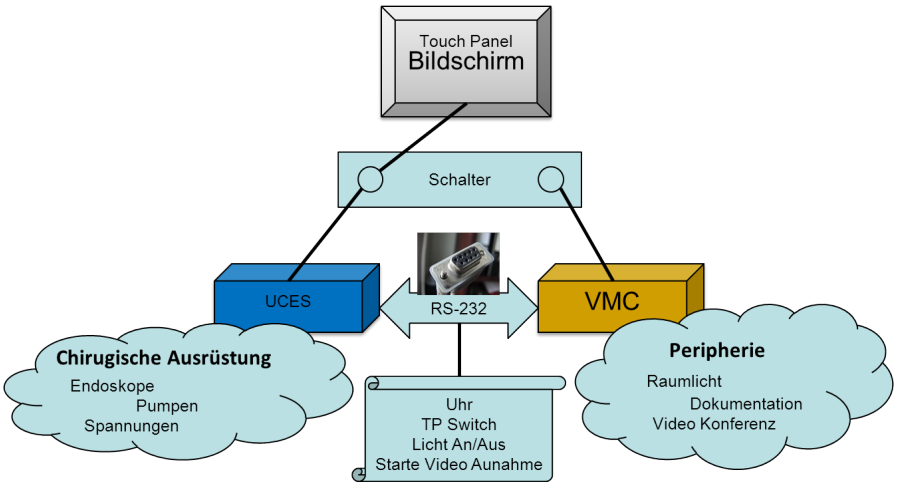


Abbildung 4.1.: Systemübersicht

UCES und VMC teilen sich einen Bildschirm. Der Wechsel zwischen beiden Systemen soll für den Benutzer transparent stattfinden. Dies bedingt, dass die grafischen Oberflächen bis auf den anwendungsspezifischen Inhalt gleich sind. Dafür müssen u. a. Informationen über Uhrzeit, Patientendaten und Lautstärke synchronisiert werden. Sowohl für Steuerbefehle als auch zur Synchronisation von Daten ist daher eine Kommunikation zwischen UCES und VMC notwendig. Realisiert ist diese Verbindung über eine RS-232 Schnittstelle. Eine Beschreibung der hier verwendeten Schnittstelle findet sich in Kapitel 4.2.1.

4.2. Informelle Protokollbeschreibung

— Vertraulicher Inhalt wurde aus dieser Version entfernt —

4.2.1. Netzwerkschicht (RS-232)

— Vertraulicher Inhalt wurde aus dieser Version entfernt —

4.2.2. Protokollschicht

— Vertraulicher Inhalt wurde aus dieser Version entfernt —

Teil III.

Verifikation und Reimplementierung

5. Verifikation des Protokolls

Die als informelle Aussagen spezifizierten Eigenschaften des Kommunikationsprotokolls zwischen UCES und VMC (siehe Kapitel 4.2) sollen mit Hilfe eines Modells und formaler Logik verifiziert werden. Zu diesem Zweck wird ein Modell erarbeitet, welches die Kommunikation zwischen UCES und VMC abbildet. Aussagen der informellen Spezifikation werden in Timed Computational Tree Logic (TCTL) formalisiert. Mit den Formeln werden die Aussagen der Spezifikation am Modell verifiziert. Zur Modellierung und Verifikation wird Uppaal verwendet (siehe Kapitel 2).

Es wird ein Modell aus zwei Automaten erarbeitet. Das Modell bildet alle relevanten Zustände der Kommunikation ab und visualisiert die logische Abfolge von Ereignissen. Um die Spezifikation in für Uppaal verständliche TCTL-Formeln zu übersetzen, werden alle Aussagen der informellen Spezifikation einzeln betrachtet und zu jeder Aussage eine oder mehrere TCTL-Formeln gesucht. In Fällen, in denen keine Formel formuliert werden kann, werden die Gültigkeiten informell gezeigt.

In Kapitel 5.1 werden das zur Verifikation verwendete Modell und die Zusammenhänge zwischen Modell und Spezifikation beschrieben. In Kapitel 5.2 werden die Aussagen der informellen Spezifikation durch die Beschreibungen in TCTL formalisiert. Kapitel 5.3 beschreibt die eigentliche Verifikation der Spezifikation auf Basis des Modells und der TCTL-Formeln.

5.1. Modell der Spezifikation

Es wird ein Uppaalmodell gesucht, welches das Kommunikationsprotokoll zwischen UCES und VMC abbildet. In Uppaal stehen verschiedene Konzepte zur Modellierung zur Verfügung (Channels, Funktionen, etc.). Die Konzepte können auf unterschiedliche Weise eingesetzt werden, um Eigenschaften des Protokolls zu modellieren. Somit ist die Modellierung des Protokolls nicht eindeutig. Zunächst

wird der Einsatz der Konzepte zur Modellierung erläutert. Für alle modellierten Eigenschaften, wie z. B. die Behandlung von Fehlern, werden entsprechende Konzepte zur Modellierung erstellt. Mit Hilfe dieser Konzepte wird dann das Modell erarbeitet.

Für jeden Kommunikationspartner wird ein Modell erstellt. Die Automaten sind im Anhang in Abbildung B.1 für die UCES und B.2 für den VMC gegeben. Die physikalische Verbindung zwischen den Kommunikationspartnern wird nicht separat modelliert, da die möglichen Zustände der Leitung überschaubar und in die Automaten von UCES und VMC integriert sind. Zur Beschreibung der Automaten werden Spezifikationen referenziert, die in Anhang A zu finden sind.

Es gibt keinen formalen Beweis über die Korrektheit eines Modells, da die Spezifikation zunächst rein informell gegeben ist. Die Korrektheit des Modells wird daher argumentativ belegt. Dazu dienen vor allem die Aussagen der Spezifikation aus Anhang A. Der Automat soll möglichst so gestaltet werden, dass es möglich ist, alle Aussagen als Formeln zu formuliert und auf dem Automaten zu verifizieren. Zur Modellierung wird neben der Spezifikation auch das Wissen über die vorhandene Implementierung genutzt. Wenn alle Aussagen auf dem Automaten formal verifiziert oder argumentativ belegt werden können, ist der Automat in Bezug auf die Spezifikation vollständig.

Bei der Modellierung muss auf zwei verschiedene Fehlerquellen geachtet werden: zum einen können Fehler auftreten, die durch Eigenschaften des Systems entstehen und zum Beispiel zu Übertragungsfehlern führen. Zum anderen können Inkonsistenzen zwischen Modell und Realität dazu führen, dass bei der Verifizierung Fehler entdeckt werden, die in Wirklichkeit keine sind. Ein Beispiel sind sogenannte Zeno Runs [HMR02]. Bei einem Zeno Run wird eine Folge von Zuständen beliebig oft ausgeführt, ohne dass Zeit vergeht. Ein solches Verhalten kommt in der Realität nicht vor und ist daher im Modell oft nicht gewollt.

Wie die Konzepte zur Modellierung eingesetzt werden, wird in Kapitel 5.1.1 beschrieben. In den folgenden Unterkapiteln wird auf die einzelnen Automaten eingegangen und argumentiert, wieso das Modell das gegebene Kommunikationsprotokoll abbildet. Hierfür wird auf den Anhang verwiesen, der alle Aussagen der Spezifikation beinhaltet. Kapitel 5.1.2 beschreibt den Automaten der UCES und Kapitel 5.1.3 den Automaten des VMC.

5.1.1. Konzepte im Modell

Es werden drei Konzepte zur Modellierung des Kommunikationsprotokolls betrachtet. Jedes der Konzepte nutzt verschiedene Funktionen von Uppaal (siehe Kapitel 2). Zuerst werden die möglichen Fehlerfälle bei der Kommunikation beschrieben. Ein wesentlicher Bestandteil des Protokolls ist die Nachrichtenübertragung. Es wird ein Konzept beschrieben, um die physikalische Übertragung von Nachrichten im Modell mittels Uppaal Channels abzubilden. Die übertragenen Nachrichten müssen anschließend behandelt werden. Bei der Behandlung von Nachrichten werden Konzepte benötigt, um Nachrichten im Modell abzubilden und empfangene Nachrichten auf Fehler zu prüfen. Am Ende wird der Einsatz von Uhren und Zeit beschrieben.

Fehlerfälle In einem korrekten Modell müssen alle möglichen Fehler behandelt werden. Für die Modellierung der Kommunikation ist nicht die Fehlerquelle entscheidend, sondern die Auswirkung eines Fehlers. So macht es im Modell keinen Unterschied, ob die Ursache für eine fehlerhaft empfangene Nachricht ein Bitflip in der Leitung ist oder durch Hardwarefehler im Empfänger verursacht wird. Ohne diese Abstraktion wäre es kaum möglich, alle Fehler zu modellieren, da viele Fehlerquellen nicht bekannt sind. So werden in der Praxis Fehlerquellen oft erst gesucht und entdeckt, nachdem eine Fehlerursache festgestellt worden ist.

Die informelle Spezifikation behandelt nur fehlerhafte und nicht übertragene Nachrichten. Dies schließt alle Fehler ein, die durch die Sende- und Empfangsleitungen der seriellen Kommunikation verursacht werden. Für das Modell bedeutet dies, dass für jede Nachricht angenommen werden muss, dass sie fehlerhaft ist oder gar nicht übertragen wird.

Alle Fehler, die eine Trennung der Verbindung zur Folge haben, z. B. das Herausziehen eines Steckers oder Fehler bei den Request to Send (RTS)- und Clear to Send (CTS)-Leitungen, führen dazu, dass die Verbindung zurückgesetzt wird. Jeder Abbruch der Verbindung setzt also das Modell auf den Startzustand zurück. Aussagen, die auf dem Modell verifiziert werden, sind daher nur gültig, solange die Verbindung besteht und nicht unterbrochen wird. Diese Annahme ist unkritisch, da UCES und VMC auch unabhängig voneinander betrieben werden können und ein Ausfall der Kommunikation zwischen UCES und VMC keinen Einfluss auf andere Funktionen der Geräte hat.

Nachrichtenübertragung Das Modell muss die physikalische Übertragung von Nachrichten zwischen den Kommunikationspartnern modellieren. Immer, wenn eine Nachricht gesendet wird, muss sie auch empfangen werden. Das bedeutet, die Automaten müssen zum Zeitpunkt der Übertragung synchronisiert werden. Dafür bietet Uppaal das Konzept von Channels (siehe Kapitel 2.2). Zur Modellierung der Nachrichtenübertragung werden global deklarierte Channels eingesetzt. Die verwendeten Channels können in zwei Gruppen unterteilt werden: Nachrichten und Signale. Nachrichten sind Folgen von Bytes, welche über die TxD Leitung gesendet werden. Signale sind Spannungsänderungen auf den RTS- und CTS-Leitungen. Die physikalische Übertragung von Bytes und Signalen ist im Kapitel 4.2.1 erläutert.

Die Darstellung von Nachrichten wird abstrahiert, um den Zustandsraum des Modells gering zu halten. Da für die modellierte Kommunikationsschicht der Inhalt von Nachrichten nicht relevant ist, wird dieser nicht modelliert. Auch Nachrichten zum Lesen und Schreiben von Daten müssen im Modell nicht unterschieden werden, da das Verhalten in beiden Fällen gleich ist. Die einzigen Lese- und Schreibnachrichten, die separat modelliert sind, sind die Nachrichten im Line Connection Prozess. Damit wird der Line Connection Prozess zwischen UCES und VMC separat synchronisiert.

— Vertraulicher Inhalt wurde aus dieser Version entfernt —

Behandlung von Nachrichten Nachrichten, die wie oben beschrieben übertragen werden, müssen gemäß der Spezifikation und mit Rücksicht auf auftretende Fehler behandelt werden. Äußere Einflüsse, die zu Fehlern bei der Nachrichtenübertragung führen, müssen modelliert werden.

Für die physikalische Verbindungsleitung (der Kommunikationskanal) wird kein eigener Automat erstellt; das Verhalten des Kanals muss dennoch abgebildet werden. Durch den Kanal können Nachrichten korrekt oder fehlerhaft übertragen werden oder verloren gehen. Wahrscheinlichkeiten können in Uppaal nicht abgebildet werden und alle Ereignisse werden als gleich wahrscheinlich angenommen. Das Verhalten des Kanals wird in der globalen Funktion `GenMessage(...)` implementiert, welche jeweils beim Empfänger evaluiert wird. Damit hat der Sender keinen Einfluss darauf, ob seine Nachricht ankommt. Durch die Verwendung von Channels zur Modellierung der Nachrichtenübertragung, ändert der Empfänger jedoch auch bei einer verlorenen Nachricht seinen Zustand. Wird eine verloren gegangene Nachricht auf diese Weise empfangen, wird dies als Timeout behandelt.

Eine empfangene Nachricht wird beim Empfänger durch die Funktion `GenMessage( int rnd, Message &msg)` generiert (Abbildung 5.2). Der Inhalt einer Nachricht

wird nicht interpretiert, der Typ einer Nachricht ist im Automaten durch den verwendeten Channel beschrieben.

Wie und ob eine Nachricht übermittelt wurde, wird durch das `Message` Struct abgebildet (Abbildung 5.1). Eine Nachricht kann nach der Übertragung als korrekt (`Correct`), fehlerhaft (`Error`) oder verloren (`Lost`) gelten. Über TCTL-Formeln (A.2 und A.3) wird sichergestellt, dass zu jeder Zeit nur ein Feld des Structs `true` ist. Informationen über den Inhalt der Nachrichten haben keinen Einfluss auf die Protokollschicht und werden daher nicht modelliert.

Die globale Funktion `GenMessage(int rnd, Message &msg)` generiert eine zufällige Nachricht beim Empfänger, wobei der Parameter `rnd` ein zufälliger Wert zwischen 0 und 2 sein muss und `&msg` eine Referenz auf die zu generierende Nachricht ist. Ein Zufallswert kann in Uppaal mittels `Select` Statement generiert werden (siehe Kapitel 2). Mit der globalen Funktion `ResetMsg(Message &msg)` kann die Nachricht `&msg` auf den initialen Wert gesetzt werden. Jeder Automat hat eine Variable `msg` vom Typ `Message`, welche die zuletzt empfangene Nachricht modelliert.

```
typedef struct {
bool Error;   // True = "CheckSum" und "Length" korrekt
bool Lost;    // Nachricht verloren
bool Correct; // Nachricht Korrekt
} Message;
```

Abbildung 5.1.: Message Struct

Einsatz von Uhren An einigen Stellen in der Kommunikation spielt Zeit eine Rolle. So muss z. B. reagiert werden, wenn nach Ablauf eines gewissen Timeouts keine Nachricht empfangen wird. Zeit ist in Uppaal ein diskreter Wert. Für das Modell wird die kleinste Zeiteinheit als eine Millisekunde definiert.

— Vertraulicher Inhalt wurde aus dieser Version entfernt —

Die Verbindung ist erst nach dem Line Connection Prozess aufgebaut und kann dann zum Senden und Empfangen von Anwendungsdaten genutzt werden. Der Verbindungszustand steht für jeden Automaten in der lokalen Variable `bool lineEst`, die nach einem erfolgreichen Line Connection Prozesses auf `true` und im Fehlerfall auf `false` gesetzt wird.

```
void GenMessage(int rnd, Message &msg)
{
    msg.Error = false;
    msg.Lost = false;
    msg.Correct = false;

    if (rnd == 0)
    msg.Correct = true;

    if (rnd == 1)
    msg.Error = true;

    if (rnd == 2)
    msg.Lost = true;
}
```

Abbildung 5.2.: Funktion zum Generieren einer empfangenen Nachricht

— Vertraulicher Inhalt wurde aus dieser Version entfernt —

(a) UCES

— Vertraulicher Inhalt wurde aus dieser Version entfernt —

(b) VMC

Abbildung 5.3.: Modell für Read, Write und ChangeNotice

5.1.2. Automat der UCES

— Vertraulicher Inhalt wurde aus dieser Version entfernt —

5.1.3. Automat des VMC

— Vertraulicher Inhalt wurde aus dieser Version entfernt —

5.2. Formale Spezifikation des Protokolls

Eines der Ziele dieser Arbeit ist es, die dem Protokoll zugrundeliegende Spezifikation und deren Implementierung zu verifizieren. Zunächst wird nur die Verifizierung der Spezifikation betrachtet. Der Grundbegriff der Verifikation wird bereits in Kapitel 1 eingeführt. Demnach werden Aussagen benötigt, welche als Grundlage für ein Modell der Spezifikation genutzt werden können. Die Aussagen werden zunächst in natürlicher Sprache aus der Spezifikation extrahiert und anschließend in TCTL-Formeln übersetzt und damit formalisiert. Die Aussagen stammen aus der Protokollspezifikation von Olympus. Bei der Erstellung der Formeln wird darauf geachtet, keine geschachtelten Ausdrücke zu verwenden. Dass hat großen Einfluss auf die Zeit- und Speicherressourcen, die zur Verifikation benötigt werden. Alle erstellten Formeln werden mit Uppaal (siehe Kapitel 2) verifiziert. Eine Aussage kann durch eine oder mehrere Formeln beschrieben werden. Ebenso kann eine Formel zu mehreren Aussagen gehören. Eine Auflistung aller aus der Spezifikation extrahierten Aussagen mit zugehörigen Formeln findet sich im Anhang A.

Insgesamt werden 25 Aussagen aus der Spezifikation extrahiert und drei weitere Aussagen werden auf Basis der aktuellen Implementierung und des Automaten erstellt. Diese Aussagen decken den gesamten Teil der Spezifikation von Olympus ab, der in dieser Arbeit verifiziert wird. Die Spezifikation von Olympus besteht aus drei Teilen, dabei wird nur der erste Teil, die Protokollschicht, verifiziert. Diese ist informell in Kapitel 4.1 beschrieben. Die Aussagen werden durch 53 TCTL-Formeln in Anhang A modelliert. Fünf Spezifikationen lassen sich nicht durch TCTL-Formeln beschreiben.

Die Aussagen der Spezifikation sind in drei Gruppen geteilt: Die Aussagen, die sich mit dem Lesen und Schreiben von Nachrichten beschäftigen, finden sich in Anhang A.1. Aussagen über den Line Connection Prozess stehen in Anhang A.2. Alle Formeln zum Thema Fehlerbehandlung stehen in Anhang A.3. Neben den direkt aus der Spezifikation abgeleiteten Formeln gibt es weitere Formeln zur Plausibilitätsüberprüfung der Automaten in Anhang A.4. Die Formeln zur Plausibilitätsüberprüfung stellen sicher, dass Konzepte im Automaten richtig verwendet werden. Darunter fallen alle Formeln, die nicht direkt auf eine Spezifikation bezogen werden können, aber die für die Argumentation zur Korrektheit und Vollständigkeit des Modells notwendig sind. So darf zum Beispiel zu jeder Zeit nur ein Feld des `Message` Structs „true“ sein.

Da es keinen formalen Nachweis für die Vollständigkeit der Formeln geben kann, soll durch systematisches Vorgehen eine komplette Abdeckung der Spezifikation

durch die Formeln erreicht werden. Zur Erstellung der Formeln wird zunächst die ursprüngliche Protokollspezifikation analysiert. Dabei werden alle Aussagen der Spezifikation extrahiert und als eigenständige Sätze in natürlicher Sprache formuliert. Es wird versucht, für jede Aussage TCTL-Formeln, basierend auf dem erstellten Modell, zu formulieren.

— Vertraulicher Inhalt wurde aus dieser Version entfernt —

Dass nicht alle Aussagen als TCTL-Formeln formuliert werden können, bedeutet für die Gesamtaussage, dass man sich nicht auf ein einfaches Ja- /Nein-Ergebnis der Verifikation verlassen kann, sondern im Einzelfall die Argumentation der einzelnen Spezifikationen berücksichtigen muss.

Die Vollständigkeit der Formeln und eine korrekte Modellierung des Modells, auf das sich die Formeln beziehen, kann nicht formal bewiesen werden. Welche Folgen mögliche Fehler bei der Modellierung und Verifikation haben, wird im folgenden Kapitel diskutiert.

5.3. Verifikation der Spezifikation

In den vorangegangenen Kapiteln ist das Modell beschrieben, das die Spezifikation abbildet und es sind Eigenschaften definiert, die auf dem Modell verifiziert werden können. In diesem Kapitel werden die Ergebnisse der Verifikation zusammengefasst und diskutiert.

Mit Uppaal wird für jede Formel der Spezifikation aus Kapitel 5.2 überprüft, ob sie für das Modell in Kapitel 5.1 gilt. Für das Modell sind damit alle Eigenschaften, die durch die Formeln ausgedrückt werden, verifiziert. Da das Modell die Spezifikation abbildet, können diese Aussagen auch auf die informelle Spezifikation übertragen werden. Das verifizierte Modell kann im weiteren Verlauf als Unterstützung bei der Reimplementierung des Protokolls eingesetzt werden. Bei der Verifikation können verschiedene bekannte Probleme auftreten, auf die im Folgenden eingegangen wird.

Bekannte Probleme bei der Verifikation Trotz aller Bemühungen, das Protokoll korrekt zu modellieren, besteht eine Plausibilitätslücke zwischen Modell und Realität, die mit formalen Mitteln nicht geschlossen werden kann. Durch systematisches Vorgehen und Begründung von Designentscheidungen soll diese Lücke so klein wie möglich gehalten werden. Die Modellbeschreibung in Kapitel

5.1 begründet die Designentscheidungen bei der Modellierung und beugt einer falschen Interpretation des Modells vor. Die Spezifikation wurde auch im Kontext der vorhandenen Implementierung betrachtet, um alle Aussagen richtig zu interpretieren.

Eine häufige Herausforderung bei der Verifikation ist das Problem der Zustandsexplosion. Dabei bilden Modelle sehr große Zustandsräume ab, die bei der Verifikation untersucht werden müssen. Das führt zu einem hohen Zeit- und Speicherbedarf bei der Verifizierung und kann im schlimmsten Fall dazu führen, dass ein Modell nicht verifiziert werden kann. Zur Verifikation des erstellen Modells in dieser Arbeit wurden maximal 144 Zustände durchlaufen. Das ist eine vergleichsweise geringe Anzahl und führt dazu, dass die Verifikation aller Formeln auf heute üblichen Rechnern in wenigen Sekunden abgeschlossen ist.

Die geringe Anzahl an Zuständen begründet sich im Design des Kommunikationsprotokolls. Das Protokoll wurde so spezifiziert, dass jede Abweichung vom vorgesehenen Kommunikationsablauf zu einem Fehler führt und jeder Fehler die Kommunikation auf den Startzustand zurücksetzt. Weiterhin hilft die einfache Betrachtung von fehlerhaften Nachrichten bei der Minimierung der möglichen Zustände. Der Nachrichteninhalt muss dafür nicht explizit modelliert werden, was andernfalls den Zustandsraum enorm vergrößern würde. Eine Nachricht bestehend aus einem Byte hat bereits 256 mögliche Zustände, also mehr als das gesamte Kommunikationsprotokoll ohne Modellierung des Nachrichteninhalts.

Ein anderes mögliches Problem von Echtzeitautomaten sind Zeno Runs [HMR02], bei denen eine Folge von Zuständen beliebig oft ausgeführt wird, ohne dass Zeit vergeht. Ein Zeno Run kann Einfluss auf die Verifizierbarkeit von Formeln haben. Ein Zeno Run wurde bereits in Kapitel 5.1.3 im Absatz über die Kommunikation beschrieben und durch einführen von Zeitverzögerungen gelöst. Dass keine weiteren Zeno Runs auftreten wird nicht formal verifiziert, aber die durchgeführte Verifizierung hat keine Hinweise auf weitere Zeno Runs gegeben.

Diskussion zur Verifizierung Obwohl die meisten Aussagen der Spezifikation auf dem Modell verifiziert werden können, müssen einige Spezifikationen genauer betrachtet werden. Einige Spezifikationen widersprechen sich oder sind unklar formuliert. Im Folgenden werden entsprechende Spezifikationen diskutiert.

— Vertraulicher Inhalt wurde aus dieser Version entfernt —

Da sich Unklarheiten in einer Spezifikation, für die bereits eine funktionsfähige Implementierung existiert, oft nicht auf theoretischer Basis lösen lassen, muss

neben der informellen Spezifikation, dem Modell und Erfahrungswerten auch die vorhandene Implementierung genutzt werden, um zu einer korrekten Interpretation der Spezifikation zu gelangen. Am Ende soll die Verifikation helfen, eine Implementierung für den VMC zu erstellen, die mit der aktuellen UCES Implementierung kommunizieren kann. Dies wird mit Hilfe von systematischen Integrationstests an der UCES zusätzlich belegt.

— Vertraulicher Inhalt wurde aus dieser Version entfernt —

6. Reimplementierung des Protokolls

Die UCES wird in Japan entwickelt und Olympus W&I entwickelt den VMC in Hamburg. Der VMC beinhaltet neben der Kommunikationsschnittstelle zur UCES noch weitere Softwarekomponenten. Im Rahmen dieser Arbeit wird nur die Kommunikation mit der UCES im VMC neu implementiert.

Die vorhandene Protokollimplementierung des VMC wird von Olympus in einer neuen Softwareversion reimplementiert. Die Reimplementierung muss der Projektstruktur der neuen Software genügen und Rücksicht auf geplante und mögliche Erweiterungen des Protokolls nehmen. Gründe für die Reimplementierung sind eine bessere Wartbarkeit, Austauschbarkeit von Komponenten und eine Verbesserung der Test- und Verifizierbarkeit. In der vorhandenen Protokollimplementierung sind die Teile der Software wie Parser, Protokollschicht und Anwendungsschicht eng miteinander verwoben, was Erweiterungen der Software teuer macht.

Die Wartbarkeit und Austauschbarkeit der Protokollimplementierung wird im neuen Design vor allem durch die klare Trennung von Benutzerschnittelle, engl. User Interface (UI) und der Protokollimplementierung erreicht. Die UI kennt keine Protokolldetails, sondern kommuniziert über einen Service mit der UCES. Der Service stellt die UCES Funktionalität in einer für die UI günstigen Form zur Verfügung. Details über die Implementierung sind somit erst innerhalb des Service notwendig, welcher die Implementierung von der UI trennt. Das Protokoll ist als Provider für den Service implementiert. Durch die Aufteilung in Service und Provider ist eine Austauschbarkeit der Implementierung, also des Providers möglich, ohne die öffentliche Schnittstelle des Service zu ändern. Abbildung 6.1 zeigt den Zusammenhang der Komponenten als UML Diagramm. Die Protokollimplementierung lehnt sich an einer serviceorientierten Architektur an [Dai11].

Im folgenden Kapitel 6.1 wird die Reimplementierung beschrieben und auf die Komponenten in Abbildung 6.1 eingegangen. Kapitel 6.2 beschreibt, wie die Reimplementierung getestet wird, und Kapitel 6.3 geht auf die Verifizierbarkeit der

Reimplementierung ein.

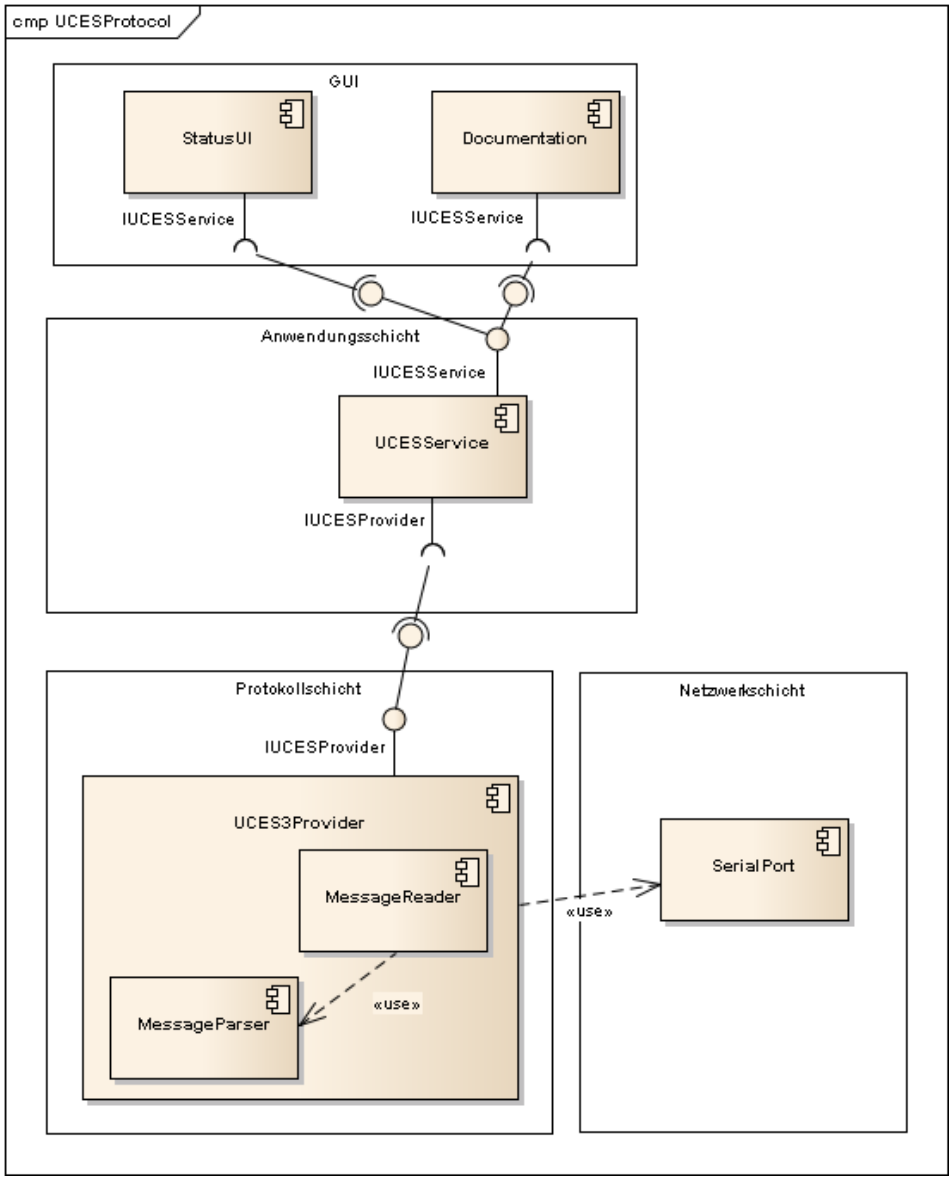


Abbildung 6.1.: Komponentendiagramm der Reimplementierung des UCES Protokolls im VMC

6.1. Beschreibung der Reimplementierung

Das Protokoll ist als Komponente der VMC Software in C# implementiert. Die Implementierung besteht aus drei Teilen: einem Service, der die Funktionalität der UCES bereitstellt, dem Provider, der die eigentliche Kommunikation implementiert, und der seriellen Kommunikation mit der UCES. Zusätzlich wird eine Status UI erstellt, die den Service nutzt und alle Daten der UCES visualisiert sowie Zugriff auf alle vom Protokoll unterstützten Funktionen bietet.

In Kapitel 4.2 wird auf die Teilung des Protokolls in mehrere Schichten eingegangen. Diese Schichten spiegeln sich auch in der Reimplementierung wider. Die unterste Schicht bildet die Netzwerkschicht, welche durch einen Wrapper um die `SerialPort` Klassen aus dem .Net Framework implementiert ist. Die darüberliegende Protokollschicht bildet den größten Teil der Software und beinhaltet Logik zur Detektierung des Verbindungszustands, den Parser und ein Interface zum Senden und Empfangen von Nachrichten. Die Protokollschicht wird durch den Provider implementiert und stellt das Interface für die Anwendungsschicht bereit.
Die Anwendungsschicht wird durch den Service implementiert, welcher die Funktionalität der UCES über den Provider anwendbar macht. Über den Service können andere Komponenten wie die Status UI auf Funktionalitäten der UCES zugreifen, ohne die Protokollimplementierung zu kennen. Abbildung 6.1 zeigt die Beziehung zwischen den einzelnen Komponenten. Im Folgenden wird genauer auf die Teile der Implementierung eingegangen.

UCESService Der `UCESService` implementiert die Funktionen, die Clients mit der UCES durchführen wollen, mit Hilfe des `UCESProvider`. Während der Provider implementierungsabhängig ist, ist die öffentliche Schnittelle des Service darauf ausgelegt, von der UI oder anderen Services genutzt zu werden und die dortigen Anforderungen zu implementieren. Innerhalb des Services werden Anfragen an das öffentliche Interface verarbeitet und notwendige Aufrufe beim Provider ausgeführt.

UCESProvider Der Provider implementiert die Protokollschicht und stellt die UCES-Funktionalitäten als Interface für den Service bereit. Der Provider implementiert Funktionen für Lese- und Schreibanfragen der UCES, eine Funktion zum Versenden von Change Notifications und verwaltet Informationen über den Zustand der Verbindung.

Zur Implementierung der Kommunikation mit der UCES nutzt der Provider den

`SerialPort`, welcher über die serielle Schnittstelle empfangene Daten bereitstellt und Daten an die UCES senden kann. Die vom `SerialPort` empfangenen Bytes werden vom `MessageReader` in Nachrichtenstrukturen übersetzt.

— Vertraulicher Inhalt wurde aus dieser Version entfernt —

6.2. Tests der Reimplementierung

Um die Funktionalität der erstellten Protokollimplementierung sicherzustellen, muss diese getestet werden. Neben der Protokollimplementierung wird zu diesem Zweck eine grafische Oberfläche, engl. Graphical User Interface (GUI) erstellt, die alle Funktionen des VMC, die eine Kommunikation mit der UCES erfordern, zugänglich macht. Dabei können 29 verschiedene Werte gesetzt oder gelesen werden, die mit der UCES synchronisiert werden. Anhand der Test-GUI werden systematische Integrationstests mit einer UCES durchgeführt.

Die Tests zeigen, dass die erstellte Protokollimplementierung alle Funktionen des VMC unterstützt, die eine Kommunikation mit der UCES erfordern. Alle von Olympus spezifizierten Anwendungsfälle wie Loginprozeduren, Benachrichtigungen an die UCES über Zustandsänderungen etc. können mit der Test-GUI ausgeführt werden.

Die so durchgeführten Tests sind Blackboxtests und decken somit nicht gezielt die inneren Strukturen der Implementierung ab und sind nicht automatisiert. Zu diesem Zweck werden in der Praxis meistens Unittests angewendet. Unittests testen einzelne Komponenten der Software. Da es aber auch mit Unittests nur schwer möglich ist, eine vollständige Testabdeckung des Quellcodes zu erreichen, wird im nächsten Kapitel die Verifizierbarkeit der Reimplementierung diskutiert.

6.3. Verifizierbarkeit der Reimplementierung

Software zu testen, ist oft mit hohem Aufwand verbunden, und eine Verifizierung von Software ist nur schwer möglich, wenn die Implementierung nicht darauf ausgelegt ist. Ein Beispiel soll dies verdeutlichen: In dem Beispiel wird eine Nachricht als Bytefolge empfangen. Es soll verifiziert werden, dass auf das Empfangen der Nachricht die richtige Antwort folgt. Es werden zwei relevante Funktionen betrachtet: in der ersten Funktion wird die Nachricht geparst und ein Ereignis

ausgelöst, welches die Nachricht enthält. Das Programm verarbeitet die Nachricht und versendet die Antwort durch den Aufruf der zweiten Funktion.

Die Funktion `Read(byte[] data)` in Abbildung 6.2 formatiert eine empfangene Folge von Bytes in eine Nachricht. Nach jeder empfangenen Nachricht wird eine Antwort erstellt und mit der Funktion `SendMessage(UCESMessage message)` in Abbildung 6.3 an die UCES gesendet.

```
internal void Read(byte[] data)
{
m_Message.Parse(data);
  if (m_Message.Finished)
  {
    if (MessageReceived != null)
      {
          MessageReceived.Invoke(
            this, new MessageReaderEventArgs(m_Message)
          );
      }
    }
}
```

Abbildung 6.2.: Lesen einer Nachricht

```
private void SendMessage(UCESMessage message)
{
    if (m_SerialPort == null)
    {
         m_Logger.Error("Can't send Message to UCES");
         return;
    }

    byte[] data = message.Compile();
    m_SerialPort.Write(data, 0, data.Length);
}
```

Abbildung 6.3.: Senden einer Antwort

Um nachzuweisen, dass nach jeder empfangenen Nachricht auch eine Antwort gesen-

det wird, muss der Programmfluss vom Empfangen des Events `MessageReceived` bis zum Aufruf der Funktion `SendMessage` analysiert werden. Hier soll das Beispiel überschaubar gehalten werden und es wird nur die Funktion `SendMessage` betrachtet. Es fällt auf, dass ein Versenden einer Nachricht nicht immer ausgeführt wird. Wenn `m_SerialPort == null` ist, wird die Antwort nicht versendet und ein Fehler erzeugt. Um den Quellcode zu verifizieren, müssen für dieses Beispiel programmweit alle Fälle geprüft werden, die `m_SerialPort` auf `null` setzen. Neben dem kontrollierten Fehlerfall, auf den explizit im Programm eingegangen wird, gibt es noch weitere, weniger offensichtliche Fehler. So kann auch der Funktionsparameter `message` als `null` übergeben werden oder `message.Compile()` `null` zurückgeben.

Statische Quellcodeanalysen sind durch große Zustandsräume im Programm oft nur mit großem Speicher- und Zeitaufwand durchführbar. Selbst die Verifizierung von abstrakten Modellen benötigt häufig viel Speicher und Zeit. Bei automatisierten Tests muss der Programmierer alle wichtigen Testfälle manuell finden. Hier können ebenfalls Fehler auftreten. Die üblichen Normen der Testabdeckung (z. B. Codeabdeckung) sagen nichts über die Qualität oder Richtigkeit der Testfälle aus.

Die Beispiele verdeutlichen, dass eine vollständige Verifikation der Reimplementierung in diesem Fall nicht durchführbar ist. Übliche Testmethoden können keinen direkten Bezug zum erstellten Modell herstellen. Daher wird im folgenden Kapitel ein Framework entwickelt, das auf Basis der bereits verifizierten Spezifikation zum Testen der Reimplementierung eingesetzt werden kann.

7. Modellbasiertes Testen der Reimplementierung

Bei der Reimplementierung des Protokolls stellt sich die Frage, wie sehr sich die Implementierung an die formale Spezifikation hält. Die formale Spezifikation liegt in Form des erarbeiteten Modells vor. In der Praxis werden Modell und Implementierung nicht zwingend von derselben Person entwickelt. Zudem besteht keine formale Beziehung zwischen Modell und Implementierung.

Im Grundlagenkapitel 3 werden bereits verschiedene Methoden zur modellgestützten Softwareverifikation vorgestellt. Betrachtet man die Möglichkeiten zur Softwareverifikation, wird schnell klar, dass eine vollständige Verifikation von Software heute nicht für alle Programme möglich ist. Ein Ansatz besteht daher darin, dass sich verifizierte Software in ihrer Architektur nach den verwendeten Verifikationsmethoden richtet. Das kann soweit gehen, dass Implementierungen aus Modellen erzeugt werden oder die Software für die Verifizierbarkeit schleifenfrei sein muss [PNW11].

Die Reimplementierung des Protokolls kann sich nicht nach den Verifikationsmethoden richten, da das Protokoll nur ein Teil der VMC Software ist und gewisse Anforderungen erfüllen müssen. Die VMC Software wird nach medizinischen Standards entwickelt und richtet sich nach einer serviceorientierten Architektur. Um die Wartbarkeit und Erweiterbarkeit der Software zu gewährleisten, dürfen keine Abhängigkeiten zu neuen Tools (z. B. zur Quellcodegenerierung) entstehen und die Implementierung muss wie die VMC Software in C# erfolgen. Spätere Änderungen werden von Olympus direkt am Quellcode vorgenommen, weshalb dieser gut lesbar und nicht automatisch generiert sein muss.

Nach Betrachtung der verschiedenen Möglichkeiten zur Verifikation wird ein Ansatz gewählt, der die Implementierung nicht verifiziert, aber basierend auf dem verifizierten Modell testet. Hierfür wird ein Framework entwickelt, das eine Beziehung zwischen Implementierung und dem Uppaalmodell herstellt und die genannten Anforderungen an die Protokollimplementierung erfüllt. Beim Ausführen des Programmcodes z. B. durch automatisierte Tests können mit Hilfe des Frameworks

Abweichungen zwischen Quellcode und Modell gefunden werden. Im Gegensatz zu einer statischen Analyse werden also nur Fehler entdeckt, die sich in ausgeführtem Code befinden. Verbunden damit muss messbar sein, wie vollständig das entwickelte Framework in die Implementierung eingebunden ist.

In Kapitel 7.1 wird beschrieben, wie die Beziehung zwischen Modell und Programmcode hergestellt wird. Kapitel 7.2 geht dann auf die Anwendung des Frameworks ein und zeigt beispielhaft die Integration im Quellcode. Kapitel 7.3 beschreibt die Implementierung des Frameworks und wie die Anbindung an Uppaal funktioniert. Kapitel 7.4 beschreibt die praktische Anwendung des Frameworks und wie das Framework getestet wird.

7.1. Beziehung zwischen Modell und Implementierung

Das entwickelte Testframework hat das Ziel, Abweichungen zwischen einer Implementierung und dem zugrundeliegenden verifizierten Modell zu erkennen. Das Modell liefert die Rahmenbedingungen für die Ausführungsreihenfolge des Quellcodes. Die Analyse findet zur Laufzeit des Programms statt.

Um mit Hilfe des Frameworks Fehler im Programmcode bezüglich des Modells zu finden, müssen Teile des Programmcodes mit Zuständen des Modells in Beziehung gebracht werden. Der Programmcode wird hierfür als Kontrollflussgraph, engl. Control Flow Graph (CFG) auf Statementlevel betrachtet. Dabei hat jedes Statement im Programm einen Knoten im CFG und jede Kante im CFG stellt eine Transition zwischen zwei Zuständen des Programms dar. Damit liegt der Programmcode wie das Modell auch als Zustandsautomat vor. Den Zusammenhang zwischen Modellen und CFG zeigen auch Studien, in denen Softwarekomponenten verifiziert werden, indem ein Modell aus dem CFG extrahiert wird [CCG03]. Das Testframework basiert auf der Beziehung zwischen Zuständen im Programm und Zuständen im Modell. Die Frage, ob ein Modell mit einer Implementierung übereinstimmt, ist allgemein unentscheidbar. Um dennoch Abweichungen der Implementierung vom Modell finden zu können, basiert der hier beschriebene Ansatz auf manuell hergestellten Beziehungen zwischen Modell und Programmcode. Es liegt am Entwickler, über die hergestellten Beziehungen zu argumentieren.

Vergleicht man den kompletten CFG eines Programms mit einem Modell, können sich beide in der Anzahl und Benennung der Variablen unterscheiden. Auch

benannte Zustände im Modell können nicht ohne weiteres im CFG wiedergefunden werden. Das Programm hat üblicherweise mehr Variablen und Zustände als ein zugrundeliegendes Modell. Betrachtet man nur die Variablen im Programmcode, die eine Entsprechung im Modell haben, können mehrere Zustände des CFG zusammengefasst werden und durch einen einzelnen Zustand im Modell abgebildet werden. Ein Beispiel ist der Nachrichtenparser der Protokollimplementierung. Der Parser macht einen großen Teil der Software aus. Im Modell (siehe Kapitel 5.1) wird das Parsen von Nachrichten aber nicht explizit modelliert, sondern das Senden und Empfangen von Nachrichten stark abstrahiert. Bevor also eine Beziehung zwischen Modell und Quellcode hergestellt werden kann, müssen die Zustände des Automaten im CFG, also im Programmcode, gefunden werden. Es liegt am Entwickler, im Fehlerfall das Modell oder die Implementierung zu korrigieren und bei jeder Anpassung des Modells die TCTL-Formeln erneut zu verifizieren.

Um Zustände im Programm mit Zuständen im Automaten vergleichen zu können, muss zunächst die Gleichheit von Zuständen definiert werden. Jeder Variablen im Modell kann eine Variable im Programm zugeordnet werden. Ein Zustand im Programm ist gleich einem Zustand im Modell, wenn alle zugeordneten Variablen in Modell und Programm denselben Wert haben.

Neben den Zuständen sind auch die Zustandsübergänge relevant. Es wird die Menge von Zuständen in der Implementierung betrachtet, die eine Entsprechung in einem als vollständig angenommenen Modell haben. Zwischen diesen Zuständen der Implementierung dürfen nur Zustandsübergänge existieren, die auch zwischen den entsprechenden Zuständen im Modell existieren. Das Framework kann Zustandsübergänge im Programm erkennen, die keine Entsprechung im Modell haben.

Ein Beispiel soll die Beziehung zwischen einem CFG und einem Modell zeigen. Abbildung 7.1 zeigt den CFG einer For-Schleife, die von 0 bis 2 läuft. Der zugehörige Quellcode ist mit abgebildet. Der Quellcode implementiert das Modell aus Abbildung 7.2a. Obwohl sich die beiden Automaten im Verhalten nicht unterscheiden, haben sie unterschiedlich viele Zustände und auch die Anzahl und Benennung der Variablen muss nicht zwingend übereinstimmen. So hat das Modell die Zustände $P_m = \{\texttt{Start}, \texttt{Loop}, \texttt{Iterate}, \texttt{Exit}\}$ und der CFG hat die Zustände $P_{cfg} = \{\texttt{Begin}, \texttt{i=0}, \texttt{i<3}, \texttt{DoSomething()}, \texttt{i++}, \texttt{Exit}\}$. Somit werden Konzepte benötigt, um Zustände inklusive Variablen von Modell und CFG in Beziehung zu setzten.

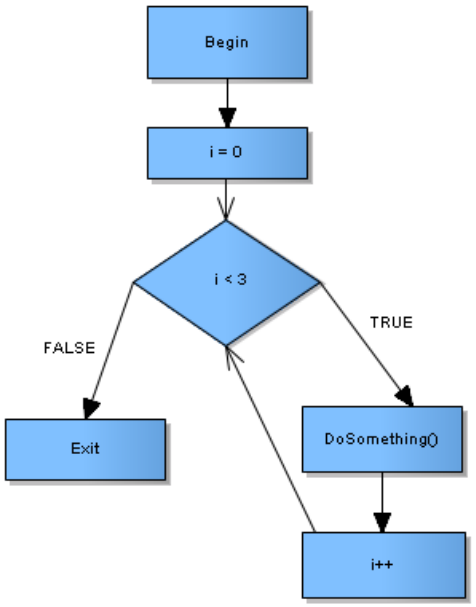
Für die Zustände wird jeder Zustand im Programm mit einem Zustand im Modell in Beziehung gesetzt. Abbildung 7.2b zeigt die Relation grafisch für dieses Beispiel. Es entstehen Tupel (L_{cfg}, L_m) mit Zuständen des Programms L_{cfg} und Zuständen

des Modells L_m.

$$R = \{(\texttt{Begin}, \texttt{Start}), (\texttt{i=0}, \texttt{Loop}), (\texttt{i<3}, \texttt{Loop}), \\ (\texttt{DoSomething(), Iterate}), (\texttt{i++}, \texttt{Loop}), (\texttt{Exit}, \texttt{Exit})\} \tag{7.1}$$

Immer wenn während der Ausführung der Zustand des Programms wechselt, muss der Automat in den Zustand wechseln, der in R festgelegt ist. Da Modelle aus mehreren Automaten bestehen können, kann eine Position im Modell auch ein Vektor mit den Positionen aller Automaten sein, wie im Uppaal Tutorial beschrieben [GB06].

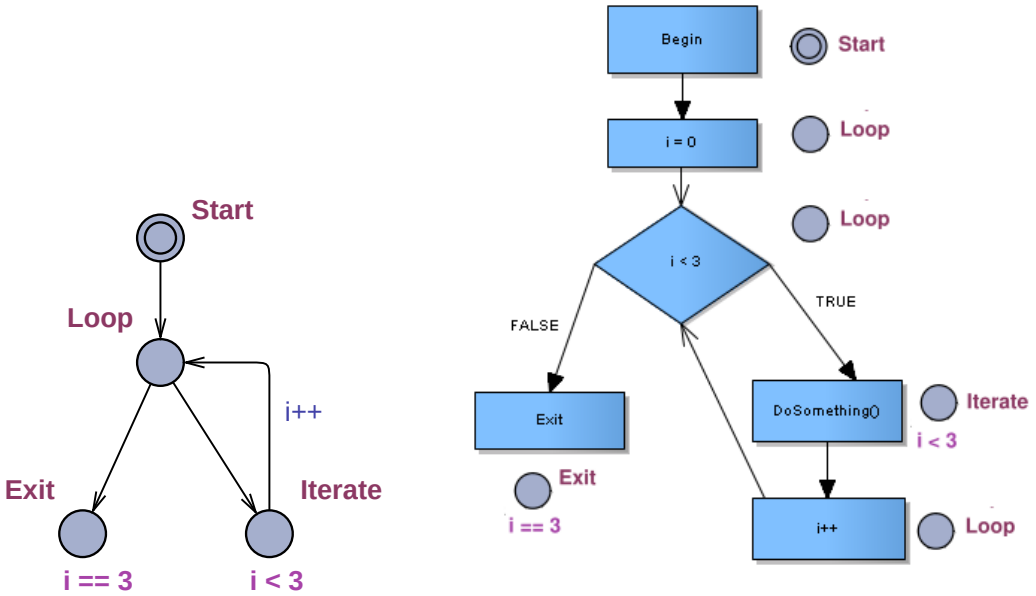
Neben den Positionen müssen auch die Variablen überprüft werden. Hierfür werden sogenannte Variablenbindungen eingeführt. Eine Variablenbindung bindet eine Variable aus dem Programm mit einer Variablen des Automaten. In diesem Beispiel wird nur eine Bindung zwischen der Variable `i` des Automaten und der Variable `i` des Modells benötigt. Bei jedem Zustandswechsel müssen die Werte aller gebundenen Variablen übereinstimmen.




```
for(int i = 0; i < 3; i++)
{
   DoSomething();
}
```

Abbildung 7.1.: For-Schleife mit CFG

Um die Vollständigkeit der Beziehung zwischen Programm und Modell zu messen, wird eine Norm benötigt. Das schwächste Kriterium ist eine Abdeckung aller



(a) Uppaalmodell einer For-Schleife (b) CFG mit zugehörigen Zuständen des Modells

Abbildung 7.2.: CFG einer For-Schleife mit zugehörigen Zuständen des Modells

Positionen des Modells. Die in Abbildung 7.2b dargestellte Relation erfüllt dieses Kriterium. Relation R (siehe Formel 7.1) zeigt, dass alle Positionen aus P_{cfg} mit mindestens einer Position aus P_m in Beziehung stehen. Wird das untersuchte Programm vollständig ausgeführt, wird für jede Transition im CFG sichergestellt, dass eine entsprechende Transition im Modell existiert.

Die vorgestellte Norm bezieht aber nicht die Variablen im Modell ein. Im Beispiel hängt die einzige Variable direkt vom Kontrollfluss des Programms ab. Das ist im allgemeinen nicht der Fall. Eine vollständige Relation muss nicht nur die Positionen des Modells an den CFG binden, sondern auch alle Variablen. Ist das der Fall, kann von einer vollständigen Zustandsabdeckung gesprochen werden.

Es ist denkbar, neben der Zustandsabdeckung auch die Kantenabdeckung zu prüfen. Das heißt, dass jede mögliche Transition zwischen zwei Zuständen im Automaten eine entsprechende Kante zwischen zwei gebundenen Zuständen im Programm hat. Damit können schon vor Ausführung des Programms Abweichungen zwischen dem CFG und dem Modell erkannt werden. Die Implementierung eines Tools zur Überprüfung der Kantenabdeckung ist aber weit komplexer als die Überprüfung, ob alle Positionen des Modells im Programm gebunden sind. Die Entwicklung eines Tools zur Messung der Modellabdeckung im Programmcode steht noch aus und ist

nicht Teil dieser Arbeit.

Das entwickelte Framework stellt Funktionen bereit, um die Relation zwischen dem Programmcode und dem Modell herzustellen. Die Funktionen stellen mehrere Eigenschaften des Programmcodes sicher: dass der Automat in einen bestimmten Zustand wechselt, dass der Automat sich in einem bestimmten Zustand befindet und dass Variablen des Programms die gleichen Werte wie gebundene Variablen des Modells haben. Zudem hat das Framework die Aufgabe, zur Laufzeit festzustellen, ob vom Programm ausgelöste Zustandsänderungen auf dem Automaten gültig sind und alle Variablenbindungen eingehalten werden.

7.2. Funktionsweise des Testframeworks

Das Framework stellt eine Beziehung zwischen Programmcode und Uppaalmodellen her. In diesem Kapitel wird ein praktischer Ansatz beschrieben, wie die Zustände des Modells an den Programmcode gebunden werden können.

Die Beziehungen werden über Annotationen im Quellcode beschrieben. Das Framework evaluiert die Annotationen zur Laufzeit und meldet dem Benutzer Abweichungen des Programms vom Modell.

Das Framework besteht aus zwei Teilen. Es simuliert während der Ausführung eines Programmes die Ausführung eines Uppaalmodells. Hierzu braucht es auf der einen Seite einen Simulator, der Information über den aktuellen Zustand des Modells und mögliche Folgezustände bereitstellt. Auf der anderen Seite muss dieser Simulator auf die Annotationen im zu verifizierenden Programm reagieren. Im Folgenden wird auf beide Teile eingegangen.

Simulation des Modells Die Simulation des Modells wird vom Uppaalserver ausgeführt (siehe Kapitel 2.1). Der Server stellt einen Simulator bereit, der zu jeder Zeit weiß, in welchem Zustand sich das Modell befindet. Zu jedem Zustand sind Informationen über alle möglichen Transitionen vorhanden. Somit kann sich der Simulator frei im Zustandsraum des Modells bewegen. Der Server kann über ein Netzwerk oder lokal betrieben werden. Durch die von Uppaal verwendete Server-Client Architektur ist es einfach möglich, einen eigenen Client für den Server zu schreiben, der den Simulator steuert.

Annotationen Um ein zu testendes Programm mit dem zugrundeliegenden Modell zu verbinden, werden im Programmcode Annotationen eingebunden. Dabei soll der Aufwand für den Entwickler möglichst gering gehalten werden. Die Annotationen bestehen aus Sprachkonstrukten der verwendeten Programmiersprache (in diesem Fall C#) und werden zur Laufzeit evaluiert. Die Evaluierung kann mit einem Compiler Flag deaktiviert werden, um die Tests (z. B. in Releaseversionen) zu deaktivieren. Im Folgenden werden die verschiedenen Annotationen vorgestellt.

Es gibt zwei Typen von Annotationen: zum einen müssen Stellen im Quellcode mit Positionen im Modell synchronisiert werden, zum anderen müssen die Variablenbindungen zwischen Modellvariablen und Variablen im Programm hergestellt werden. Zur Synchronisation der Positionen gibt es zwei verschiedene Annotationen: `RequireState` und `EnsureState`. Variablen können gebunden und wieder entbunden werden. Dafür stehen die Annotationen `Bind` und `Unbind` zur Verfügung. Eine Übersicht der Annotationen findet sich in Tabelle 7.1. Im Folgenden wird die Anwendung und Funktionsweise der verschiedenen Annotationen beschrieben.

`RequireState(Location loc)` markiert einen Zustandswechsel. Beim Ausführen einer `RequireState` Annotation, wechselt das Modell in einen Zustand mit der angegebenen Position `loc`. Für den Zustand, in den gewechselt wird, werden neben der Position auch alle gebundenen Variablen überprüft. Ist eine Transition im Modell vom aktuellen Zustand zu dem von der Annotation verlangten Zustand nicht möglich, wird eine Fehlermeldung (Exception) erzeugt, die Aussagen über die fehlgeschlagene Transition liefert.

`EnsureState(Location loc)` stellt sicher, dass sich der Automat an der annotierten Stelle im Programm in einem gegebenen Zustand befindet. Dabei werden neben der Position des Modells auch alle gebundenen Variablen überprüft. Befindet sich der Automat nicht im geforderten Zustand, wird eine Fehlermeldung erzeugt, die Aussagen über den aktuellen und den erwarteten Zustand liefert.

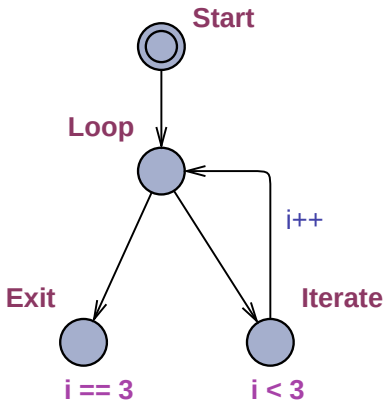
Für die Variablenbindung werden spezielle Uppaalvariablen (`UppaalVariable`) im Programm verwendet. Uppaal unterstützt nur Ganzzahlen und Boolsche Variablen, diese Limitierung überträgt sich auch auf die im Programm zur Verfügung stehenden Uppaalvariablen. Eine Uppaalvariable kann im Quellcode mit der Annotation `Bind(UppaalVariable var)` gebunden werden. Sobald eine Variable gebunden ist, wird der Wert der Variablen bei jeder `RequireState` Annotation nach dem Zustandswechsel und bei jeder `EnsureState` Annotation überprüft.

Annotation	Beschreibung
`RequireState(Location loc)`	Zustandswechsel
`EnsureState(Location loc)`	Überprüfung des aktuellen Zustands
`Bind(UppaalVariable var)`	Binden einer Variable
`UnBind(UppaalVariable var)`	Lösen einer Variablenbindung

Tabelle 7.1.: Annotationen

Beispiel An dem Beispiel aus Kapitel 7.1 soll gezeigt werden, wie Quellcode mit einem Modell in Beziehung gesetzt werden kann. Bei einer korrekten Implementierung des Modells treten keine Fehler auf. Später im Beispiel wird gezeigt, was passiert, wenn der Quellcode nicht dem Modell entspricht. Das Framework hilft dann, die Fehler in der Implementierung zu finden.

Um die Beziehung zum Modell herzustellen, werden die Annotationen in Form von Funktionsaufrufen in den Programmcode eingebettet (siehe Tabelle 7.1). Abbildung 7.3 zeigt ein Modell (links), welches durch eine For-Schleife (rechts) implementiert ist. Das Modell gibt vor, dass der Zustand `Iterate` dreimal erreicht wird, bevor die Schleife über den Zustand `Exit` verlassen wird.




```
for(int i = 0; i < 3; i++)
{
   DoSomething();
}
```

Abbildung 7.3.: Beispiel Modell mit annotiertem Code

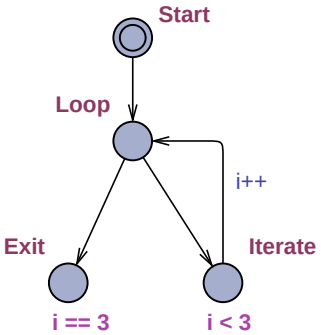
In Kapitel 7.1 wird anhand von Abbildung 7.2 gezeigt, welche Teile des Programms mit welchen Zuständen des Modells in Beziehung stehen. Das Wissen wird nun genutzt, um die Annotationen in den Quellcode einzuarbeiten. Der annotierte Code steht in Abbildung 7.4.

Vor der Ausführung des Beispielcodes befindet sich der Automat im Startzustand `Start`. Noch bevor die erste Programmzeile ausgeführt ist, muss der Automat in den

Zustand `Loop` wechseln. Dafür wird die Annotation `RequireState(...)` in Zeile 1 eingefügt. Die Funktion `DoSomething()` wird im Zustand `Iterate` ausgeführt. Um das sicher zu stellen, wird vor dem Funktionsaufruf die Annotation in Zeile 4 eingefügt. Anschließend wird in Zeile 6 sichergestellt, dass sich der Automat vor dem Inkrementieren von `i` wieder im Zustand `Loop` befindet. Nach dem Durchlaufen der For-Schleife muss der Automat sich im Zustand `Exit` befinden. Eine entsprechende Annotation ist am Ende des Beispielcodes in Zeile 8 zu finden.

Damit das Framework mögliche Fehler erkennen kann, muss das Programm ausgeführt werden. In diesem Beispiel gibt es keine Parameter, von denen der Kontrollfluss abhängt und somit nur einen möglichen Programmablauf. Das Beispielprogramm muss also nur einmal ausgeführt werden, um eine Abdeckung aller Zustände und Zustandsübergänge zu erreichen. Ein einfacher Durchlauf des Beispielprogramms führt alle eingebetteten Annotationen in der einzig möglichen Reihenfolge aus und ist somit in der Lage, alle Abweichungen des annotierten Programms vom Modell zu finden.

Angenommen, der Code ist fehlerhaft und iteriert viermal wie in Abbildung 7.5. Dann ist ist der Automat nach der dritten Iteration im Zustand `Loop` und die Zustandsvariable `i` hat im Automaten den Wert 3. Das Programm führt nun eine weitere Iteration aus und versucht, in den Zustand `Iterate` zu wechseln. Dies wird auf Grund der Invarianten `i < 3` fehlschlagen und einen entsprechenden Fehler erzeugen. In diesem Beispiel genügt schon ein einfaches Ausführen des Codes ohne weitere Parameter, um einen strukturellen Fehler im Programm feststellen zu können. In der Praxis müssen zu testende Programme meist öfter ausgeführt werden und eine Testabdeckung aller möglichen Ausführungsreihenfolgen ist nur schwer zu erreichen.




```
1 Uppaal.RequireState(UppaalState.Loop);
2 for(int i = 0; i < 3; i++)
3 {
4   Uppaal.RequireState(UppaalState.Iterate);
5   DoSomething();
6   Uppaal.RequireState(UppaalState.Loop);
7 }
8 Uppaal.RequireState(UppaalState.Exit);
```

Abbildung 7.4.: Beispiel Modell mit annotiertem Code

Das Beispiel zeigt, dass durch das Überprüfen von Beziehungen zwischen Programmcode und Modell Fehler entdeckt werden können. Dabei wird noch keine

```
Uppaal.RequireState(UppaalState.Loop);
for(int i = 0; i < 4; i++)
{
  Uppaal.RequireState(UppaalState.Iterate);
  DoSomething();
  Uppaal.RequireState(UppaalState.Loop);
}
Uppaal.RequireState(UppaalState.Exit);
```

Abbildung 7.5.: Fehlerhafter annotierter Code

Variablenbindung genutzt. Um die Beziehung zwischen allen Zuständen des Programms mit allen Zuständen des Automaten herzustellen, ist es nötig, auch die Variable `i` im Programm an den Automaten zu binden. Abbildung 7.6 zeigt das Programm aus dem vorherigen Beispiel, erweitert um Annotationen, welche die Schleifenvariable `i` an die Variable `i` des Modells binden. Weiterhin wird die Funktion `DoSomething` so erweitert, dass sie eine Referenz auf die Variable `i` übergeben bekommt. Nach der For-Schleife hat die Variable `i` im Programm keine Gültigkeit mehr und die Bindung wird mit `UnBind` wieder aufgehoben.

Mit der Variablenbindung kann nun zusätzlich sichergestellt werden, dass die Funktion `DoSomething` keine Änderungen an der Variablen `i` vornimmt. Bei jedem Ausführen einer `RequireState`-Annotation wird überprüft, ob die Variable `i` im Programm den gleichen Wert hat wie im Modell. Eine Abweichung der Werte führt also direkt nach dem Ausführen von `DoSomething` zu einem Fehler.

```
Uppaal.RequireState(UppaalState.Loop);
UppaalVariable<int> i = new UppaalVariable<int>(UppaalVar.i);
Uppaal.Bind(i);
for(i = 0; i < 3; i++)
{
  Uppaal.RequireState(UppaalState.Iterate);
  DoSomething(ref i);
  Uppaal.RequireState(UppaalState.Loop);
}
Uppaal.RequireState(UppaalState.Exit);
Uppaal.UnBind(i);
```

Abbildung 7.6.: Code mit Variablenbindung

Mit dem vorgestellten Framework können auf Basis der Annotationen Abweichungen des Kontrollflusses vom Modell entdeckt werden. Dabei werden Zustände berücksichtigt, die eine Entsprechung im Modell haben. Weitere Zustände des Programms, die keine Beziehung zum Modell haben, können ohne vorherige Erweiterung des Modells nicht überprüft werden. Fehler werden nur bei Zustandsänderung des Programms erkannt. Im Hinblick auf das Halteproblem kann weder überprüft werden, ob das Programm anhält, noch können Fehler entdeckt werden, die nicht im Modell dargestellt werden können. Alle Tests basieren auf manuell erstellten Annotationen. Fehler in den Annotationen haben direkten Einfluss auf die Korrektheit der Tests.

7.3. Implementierung des Frameworks

In den vorherigen Kapiteln wird ein Framework vorgestellt, welches mit Hilfe von Annotationen im Quellcode eine Beziehung zu einem Modell herstellt. Zur Laufzeit des Programms wird das Modell entsprechend der Annotationen simuliert. Die Implementierung des Frameworks muss auf der einen Seite eine Anbindung an die vom zu testenden Programm verwendete Programmiersprache haben. Auf der anderen Seite muss ein in Uppaal erstelltes Modell mit Hilfe der Uppaalserveranwendung simuliert werden. In dieser Arbeit wird eine Implementierung erstellt, die Annotationen für C# bereitstellt. Die Anbindung an den Uppaalserver wird in Java realisiert, da eine Javabibliothek zur Kommunikation mit dem Server bereits existiert.

Das Framework besteht aus zwei Teilen: einer C#-Bibliothek mit Implementierung für die Annotationen (`ModelDriver`) und einer Anbindung an den Simulator von Uppaal in Java (`ModelSimJava`). Das UML-Diagramm in Abbildung 7.7 zeigt den Zusammenhang der verschiedenen Komponenten. Die Protokollimplementierung ist das Programm, das gegen ein vom Uppaalserver geladenes Modell getestet werden soll.

Der `ModelDriver` implementiert die Annotationen als C# Funktionen und Klassen. Das zu testende Programm, hier die Protokollimplementierung, referenziert den als Bibliothek bereitgestellten `ModelDriver` und hat so Zugriff auf die Annotationen. Die Simulation des Modells findet im Uppaalserver statt, der über die Standard I/O oder Ethernet angesprochen werden kann. Uppaal stellt zur Nutzung des Servers eine Javabibliothek bereit. Um die Verbindung zwischen der .Net-Umgebung für die Annotationen und der Javaumgebung für den Simulator herzustellen, wird ein SOAP-WebService genutzt (`ModelSimService`). Im Folgenden wird zunächst die

Javaumgebung beschrieben und anschließend die .Net-Umgebung.

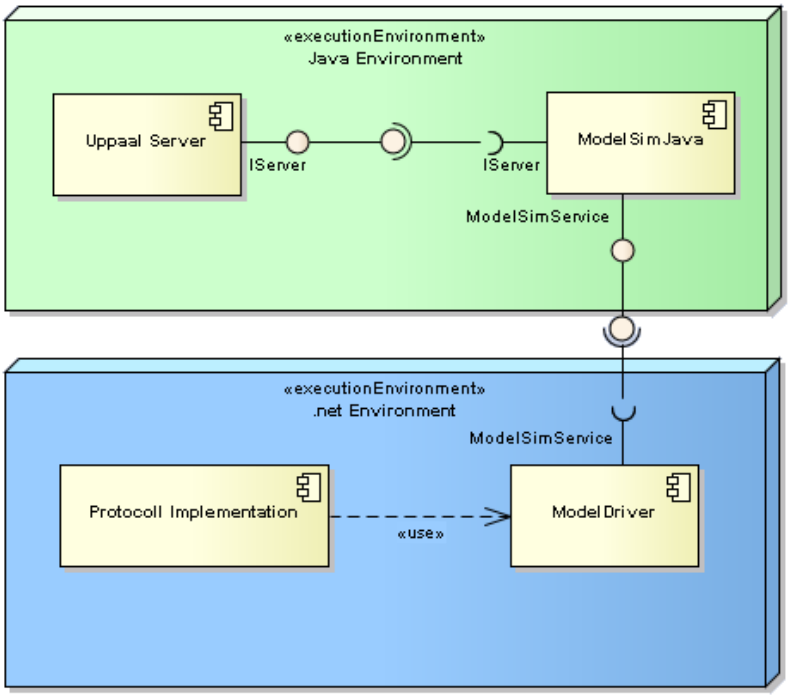


Abbildung 7.7.: Komponenten des Frameworks

Javaumgebung In der Javaumgebung ist die Anwendung `ModelSimJava` implementiert. `ModelSimJava` kommuniziert mit dem Uppaalserver, um Modelle zu laden und zu simulieren. `ModelSimJava` stellt mehrere Funktionen zu Verfügung: das Laden eines Modells im Simulator, die Bereitstellung von Informationen über alle Variablen und Positionen des aktuellen Zustands, das Wechseln in einen möglichen Folgezustand des Modells, das Zurücksetzen des Modells auf den Startzustand und das Setzen, Löschen und Abfragen von Bedingungen für Variablen im Modell. Die Bedingungen für Variablen werden nach jedem Zustandswechsel überprüft und stellen sicher, dass Variablen im Modell einen bestimmten Wert haben. `ModelSimJava` benachrichtigt den Aufrufer beim Wechsel in einen nicht existierenden Folgezustand. Die Funktionalität wird über einen Webservice bereitgestellt, so dass die Funktionen von anderen Komponenten, wie hier dem `ModelDriver`, genutzt werden können. Mit `ModelSimJava` kann nur ein Modell zur Zeit geladen werden. Dies befindet sich nach dem Laden im initialen Zustand.

Ein Zustandswechsel geht immer vom aktuellen Zustand aus. Die Entscheidung, in welchen Folgezustand gewechselt wird, wird auf Basis einer Bedingung an die Transition entschieden. Erfüllt eine Transition die Bedingung, kann sie gefeuert werden. Ist die Bedingung nicht erfüllt, kann nicht in den entsprechenden Folgezustand gewechselt werden. Wenn eine Bedingung mehrere Transitionen ermöglicht, wird eine mögliche Transition zufällig ausgewählt und gefeuert. Ist keine gültige Transaktion vorhanden, wird eine Exception geworfen. Hierbei ist zu bedenken, dass durch das zufällige Auswählen der Transition die Ausführung nicht reproduzierbar ist. Eine bessere Alternative kann daher sein, die gewählte Transition zu protokollieren oder eine Fehlermeldung zu erzeugen, wenn mehrere Transitionen möglich sind.

Wie die Bedingungen funktionieren, zeigt das folgende Beispiel. Die Implementierung der Methode `RequireState(string locationName)` ist in Abbildung 7.8 gegeben. Die Funktion nimmt den Namen einer Position im Modell als Eingabe und teilt dem Simulator über den Aufruf der Methode `MoveNext(cond)` in Zeile 13 mit, dass in den nächsten Zustand gewechselt werden soll. Für den Folgezustand muss die übergebene Bedingung `cond` gelten. Dabei wird die Bedingung zunächst mit einer `locationCondition` für die übergebene Position initialisiert (Zeile 7). Wenn es weitere Annahmen gibt (Zeile 9), werden diese Annahmen in einer `andCondition` mit der `locationCondition` verknüpft (Zeile 11). Die Bedingung wird über den Webservice an `ModelSimJava` übergeben und dort ausgewertet. Jede Bedingung implementiert eine Funktion, die überprüft, ob die Bedingung im Modell für eine gegebene Transition zutrifft. `ModelSimJava` kann so für jede mögliche Transition in einem Zustand überprüfen, ob der Folgezustand für eine Bedingung gültig ist.

.Net-Umgebung In der .Net-Umgebung ist sowohl die zu testende Anwendung als auch der `ModelDriver` implementiert. Der `ModelDriver` ist eine C#-Bibliothek, die einen Client für den Webservice nutzt, um Zugriff auf ein Uppaalmodell zu bekommen. Die Bibliothek stellt die Annotationen für das zu verifizierende Programm bereit und wertet diese aus.

Durch die lose Kopplung zwischen der C# -Bibliothek und dem Uppaalmodell über einen Java-Webservice ist es möglich, dass mehrere Programme über ein Netzwerk auf das selbe Uppaalmodell zugreifen. Dies kann gerade bei Client/Server-Anwendungen wie Kommunikationsprotokollen nützlich sein, wenn beide Kommunikationspartner im gleichen Modell modelliert sind, aber als unabhängige Software realisiert sind. Ein weiterer Vorteil der Trennung ist die Austauschbarkeit der Komponenten. So können einfach Implementierungen in anderen Programmiersprachen erstellt werden, ohne die Anbindung an Uppaal neu zu programmieren. Auf

```
1   [Conditional(CompileFlag)]
2   public void RequireState(string locationName)
3   {
4     if (!IsSystemAvalible)
5       throw new ModelDriverException("System not Avalible");
6
7     condition cond = new locationCondition(locationName);
8
9     if (m_Expectations != null)
10    {
11      cond = new andCondition(cond, m_Expectations);
12    }
13    MoveNext(cond);
14  }
```

Abbildung 7.8.: Implementierung der Funktion `RequireState`

der anderen Seite kann auch der Server einfach ausgetauscht werden, um andere Modelle und Tools zu unterstützen.

Alle öffentlichen Funktionen im `ModelDriver` sind mit dem Conditional Attribut gekennzeichnet und werden nur kompiliert, wenn in der Zielanwendung die Compilervariable `UPPAAL` gesetzt ist. Dies hat mehrere Vorteile: zur Ausführung der Annotationen werden Abhängigkeiten wie Uppaal benötigt, die beim Kunden nicht zwingend vorhanden sind. Zudem führt die Auswertung der Annotationen zu einer Verzögerung bei der Ausführung des Programms, was in einer Produktivumgebung störend sein kann.

In den Annotationen werden alle Positionen und Variablen der Uppaalmodelle über Strings angesprochen. Dies führt leicht zu Fehlern bei der Einbindung der Annotationen. Um den Zugriff auf die notwendigen Strings zu vereinfachen, implementiert der `ModelDriver` eine zusätzliche Funktion zur Generierung einer C# -Klasse. In der generierten Klasse sind Felder mit den Strings für alle Positionen und Variablen im Uppaalmodell enthalten. Die Klasse kann im zu verifizierenden Programm genutzt werden, um typsicher auf die Strings im Modell zugreifen zu können. Ein Teil des generierten Codes ist in Abbildung B.3 im Anhang zu finden.

7.4. Anwendung und Tests des Frameworks

Das Framework wird am Beispiel der Reimplementierung des Protokolls angewendet. Dabei werden Annotationen in den Programmcode eingebettet, die eine Beziehung zum in dieser Arbeit erstellten und verifizierten Modell herstellen. Neben dem praktischen Einsatz wird die Funktionalität des Frameworks auch durch Unittests sichergestellt.

Die eingebetteten Annotationen werden in Integrationstestset mit Hilfe der erstellten Test-GUI (siehe Kapitel 6.2) getestet. Die Tests zeigen Abweichungen zwischen Modell und Implementierung, deren Gründe auf das Modell, die Reimplementierung oder die falsche Verwendung der Annotationen zurückzuführen sind. Letztere sind oft Flüchtigkeitsfehler des Entwicklers oder Fehlinterpretationen des Modells oder der Implementierung. Wird nach einem gefundenen Fehler das Modell angepasst, muss die Verifikation aller Formeln erneut durchgeführt werden. Dies kann je nach Größe und Komplexität des Modells einen erheblichen Zeitaufwand benötigen.

Beim Abbruch der Verbindung durch den VMC kann eine Abweichung zum Modell festgestellt werden. Das Programm kann in der aktuellen Reimplementierung die Verbindung abbrechen, während das Modell im Zustand `LineEstablished` ist. Hier müsste ein Abbruch der Verbindung zum Zustand `Start` im Modell führen. Diese Kante ist im Modell jedoch nicht vorhanden, da dieses Verhalten in der Spezifikation nicht vorgesehen ist. Das Framework wirft eine entsprechende Fehlermeldung. In diesem Fall muss die Implementierung angepasst werden, um dem Modell zu genügen. Dieses Beispiel zeigt, dass Abweichungen zwischen Modell und Implementierung vom Framework aufgedeckt werden können und das Framework die erwarteten Anforderungen erfüllt.

Neben der beispielhaften Anwendung des Frameworks in der Protokollimplementierung wird das Framework mit Unittests getestet. Dabei muss sowohl die C# -Bibliothek als auch der Java-Webservice getestet werden (siehe Kapitel 7.3). Die Unittests sind nur im C# -Teil implementiert, welcher aber vom Webservice abhängt. Damit werden durch die Tests alle Teile des Frameworks abgedeckt. Insgesammt decken 25 Unittests das öffentliche Interface ab, das aus zwei Klassen mit neun Funktionen besteht. Die Tests beinhalten sowohl Positiv- als auch Negativtests und basieren auf einem beigelegten Modell. Alle Unittests können erfolgreich durchgeführt werden.

Teil IV.

Zusammenfassung

8. Fazit

In dieser Diplomarbeit wird ein von Olympus entwickeltes Kommunikationsprotokoll analysiert und verifiziert. Das Protokoll dient zur Kommunikation zwischen den Systemen UCES und VMC, die von Olympus entwickelt und im Operationssaal eingesetzt werden. Ziel der Arbeit ist die Erstellung einer funktionierenden Reimplementierung des Protokolls im VMC. Zur Sicherstellung der korrekten Funktionsweise wird das Protokoll modellbasiert verifiziert. Die Verifikation basiert auf der vorhandenen Protokollspezifikation und auf der existierenden Implementierung des Protokolls. Die anschließende Reimplementierung baut auf den Ergebnissen der Verifikation auf. Es wird weiterhin sichergestellt, dass die Reimplementierung mit der vorhandenen UCES fehlerfrei kommunizieren kann und dass die Aussagen der Spezifikation für die Reimplementierung gelten. Dazu wird zum einen das reimplementierte Protokoll in einem Integrationstest mit der UCES getestet, zum anderen wird ein neuer Ansatz entwickelt, der ein Testen der Reimplementierung unter Verwendung des verifizierten Modells ermöglicht.

Das Kommunikationsprotokoll wird in einem Modell abgebildet. In der Arbeit werden Konzepte zur Modellierung des Kommunikationsprotokolls beschrieben und erfolgreich angewendet. Dabei wird das Verhalten beider Kommunikationspartner in zwei Echtzeitautomaten modelliert. Mit dem erstellten Modell wird anschließend die Protokollspezifikation verifiziert. Dafür wird die informell vorliegende Spezifikation, bestehend aus 28 Aussagen, zunächst in Timed Computational Tree Logic (TCTL) formalisiert. 23 Aussagen werden erfolgreich mit dem Tool Uppaal verifiziert. Für die verbleibenden Aussagen wird die Verifikation argumentativ durchgeführt. Eine Aussage ist nicht verifizierbar. Weiterhin wird ein Widerspruch in der Spezifikation aufgedeckt und einige unklar formulierte Aussagen werden einzeln diskutiert. Für nicht eindeutige Aussagen der informellen Spezifikation wird die vorhandene Implementierung als maßgebend angenommen, da die Reimplementierung kompatibel zum existierenden System sein muss. Die im Rahmen der Verifikation durchgeführten Tests der vorhandenen Implementierung decken zwei Abweichungen von der Spezifikation auf, die bei einer Reimplementierung ebenfalls berücksichtigt werden müssen. Zum einen wird ein Datenfeld in Nachrichten nicht wie erwartet verwendet, zum anderen haben Tests einen Deadlock in der

Protokollimplementierung gezeigt.

Auf Basis der Verifikation wird das Protokoll für den VMC reimplementiert. Die Reimplementierung erfolgt in C# und erfüllt alle Anforderungen von Olympus. Sie wird von Olympus in die nächste Version des VMC integriert. Die grundlegende Funktionsweise wird erfolgreich durch Integrationstest mit der UCES getestet. Eine grafische Testoberfläche deckt alle Funktionen des Protokolls ab.

Herausfordernd ist - trotz Vorlage eines verifizierten Modells - sicherzustellen, dass alle Aspekte des Spezifikation tatsächlich von der Implementierung abgedeckt werden. Dafür reichen die durchgeführten Integrationstests nicht aus, weil interne Zustände der Implementierung damit nicht getestet werden. Es wird daher ein neuer Ansatz entwickelt, mit dem sich die Implementierung auf Basis des verifizierten Modells testen lässt.

Für den Ansatz wird eine Beziehung zwischen Programmcode und dem verifizierten Modell hergestellt. Ein in dieser Arbeit entwickeltes Framework kann auf Basis von manuell hergestellten Beziehungen zur Laufzeit des Programms Abweichungen zwischen Modell und Implementierung feststellen. Ein Abgleich der Implementierung mit dem Modell hilft dem Entwickler schon im frühen Entwicklungsstadium, alle Zustände des Protokolls zu berücksichtigen. Mit dem Framework werden Teile des Programmcodes über Annotationen mit Zuständen im Modell in Beziehung gesetzt. Dabei unterstützt das Framework Annotationen für C# und bietet eine direkt Anbindung an den Uppaalserver, der zur Evaluation der Annotationen am Modell genutzt wird. Dem Entwickler steht mit dem Framework ein Werkzeug zur Verfügung, um eine direkte Brücke zwischen Design und Realisierung zu schlagen.

Die Funktionalität des Testframeworks wird mit Unittests und einem Integrationstests am reimplementierten Protokoll getestet. Zusätzlich wird die Anwendbarkeit des Frameworks an der Reimplementierung des Protokolls evaluiert. Die Annotationen werden in die Reimplementierung eingebettet und zeigen, dass sich die Reimplementierung während der Integrationstests wie das Modell verhält. Die Integration der Annotationen in die Reimplementierung erhebt keinen Anspruch auf Vollständigkeit, dient aber als Vorzeigebeispiel für eine erfolgreiche Anwendung des Testframeworks in der Praxis.

Im Ergebnis liefert die Arbeit eine Verifizierung des existierenden Kommunikationsprotokolls. Dabei wird ein weiteres praktisches Beispiel für die Modellierung von Kommunikationsprotokollen in Uppaal geliefert. Auf Basis der Verifikation wird das Kommunikationsprotokoll für den VMC reimplementiert. Die erstellte Reimplementierung wird von Olympus in zukünftigen Softwareversionen im Produk-

tionsbetrieb verwendet. Ein Framework zum modellbasierten Testen ist erfolgreich implementiert und hilft beim Entwickeln und Testen der Protokollimplementierung, indem Unterschiede zwischen verifiziertem Modell und Implementierung frühzeitig aufgedeckt werden.

9. Ausblick

— Vertraulicher Inhalt wurde aus dieser Version entfernt —

Zum produktiven Einsatz der Reimplementierung des Protokolls fehlen noch ausführliche Unittests. Die Arbeit beschränkt sich auf Integrationstests zwischen der vorhandenen Implementierung und der Reimplementierung sowie auf die Unittests, die im Rahmen des entwickelten Frameworks durchgeführt werden. Zudem muss eine Dokumentation gemäß der Qualitätsrichtlinien bei Olympus erstellt werden.

Parallel zur Diplomarbeit wurden von Olympus Erweiterungen für das Protokoll ausgearbeitet, die in der Implementierung nachgepflegt werden müssen. Diese Änderungen können durchgeführt werden, ohne dass die Verifikation ihre Gültigkeit verliert, da sie nur ausschließlich die Anwendungsschicht betreffen.

Das entwickelte Testframework zeigt, dass es möglich ist, Teile einer Implementierung mit Zuständen eines Modells abzugleichen und so Unterschiede zwischen Kontrollfluss des Programms und Zustandsübergängen im Modell aufzudecken. Es sind weitere Evaluationen notwendig, die zeigen, wie praktikabel der Einsatz des Frameworks im Alltag der Softwareentwicklung ist. Mit dem Frameworks können bei der Entwicklung von Software viele Fehler bereits im frühen Entwicklungsstadium vermeiden werden. Ein wichtiger Schritt zur produktiven Anwendung des Frameworks ist die Entwicklung eines Tools, das die Vollständigkeit der Beziehung zwischen Programm und Modell mit den in dieser Arbeit vorgestellten Normen misst.

Es werden verschiedene Normen vorgestellt, die Auskunft über die Vollständigkeit der Einbindung des Frameworks in das zu testende Programm liefern. Aus Zeitgründen wurde bisher kein Tool entwickelt, um diese Normen in der Praxis anwenden zu können.

Denkbar ist auch eine Erweiterung des Testframeworks zu einer statischen Analyse der Annotationen. Um eine Aussage treffen zu können, ob man die Ergebnisse des Testframeworks auch nur mit statischen Analysen erzielen kann, ist weitere Forschung notwendig.

Durch die Client/Server-Architektur des Testframeworks ist es einfach möglich, neben der Anbindung an Uppaal auch andere Modellierungstools zu unterstützten. So wäre eine Anbindung an UML Tools wie Enterprise Architekt denkbar. Auch der Client, der die Annotationen für C# bereitstellt, kann durch Implementierungen in anderen Sprachen wie Java ersetzt und mit beliebigen Serverimplementierungen genutzt werden.

Teil V.

Anhang

A. Protokoll Spezifikation

In diesem Kapitel findet sich eine Auflistung aller Aussagen der Spezifikation (siehe Kapitel 5.2). Die Aussagen sind in Absätze gegliedert. Zu jeder Aussage gibt es eine oder mehrere Formeln mit kurzen Kommentaren. Gemäß den Teilen des Protokolls sind die Aussagen in die Kategorien Lesen und Schreiben (A.1), Line Connection Prozess (A.2) und Fehlerbehandlung (A.3) aufgeteilt. Der letzte Abschnitt beinhalten Formeln zur Prüfung der Plausibilität des Automaten (A.4).

Für alle gegeben Formeln ϕ gilt $M, s \models \phi$ dabei ist M das verwendete Modell und s der Startzustand. Das Modell mit seinem Startzustand ist in Kapitel 5.1 beschrieben.

Die temporalen Operatoren haben eine höhere Präzedenz als die logischen Operatoren. Somit gilt zum Beispiel: $E\Box\, \phi \wedge \psi \Leftrightarrow E\Box\, (\phi \wedge \psi)$. Die verwendete Logik ist in Kapitel 2.3 beschrieben. In Tabelle A.1 sind weitere verwendete Formelzeichen definiert.

Symbol	Bedeutung
$\phi \Rightarrow \psi$	ϕ impliziert ψ
$\phi \rightsquigarrow \psi$	ϕ führt zu ψ, entspricht $A\Box\, (\phi \Rightarrow A\Diamond\, \psi)$

Tabelle A.1.: Symbole

A.1. Lesen und Schreiben

— Vertraulicher Inhalt wurde aus dieser Version entfernt —

A.2. Line Connection Prozess

— Vertraulicher Inhalt wurde aus dieser Version entfernt —

Connection Detection

— Vertraulicher Inhalt wurde aus dieser Version entfernt —

A.3. Fehlerbehandlung

— Vertraulicher Inhalt wurde aus dieser Version entfernt —

A.4. Plausibilität des Modells

Die Spezifikationen in diesem Kapitel stammen nicht aus der Spezifikation von Olympus. Es sind Annahmen, die für das Modell getroffen werden, um sicherzustellen, dass Konzepte im Modell richtig verwendet wurden.

Spezifikation 1: Es treten keine Deadlocks auf.

$$M, s \models A\Box \texttt{not deadlock} \tag{A.1}$$

Spezifikation 2: Ist eine Nachricht nicht korrekt, ist sie verloren oder fehlerhaft. Eine Nachricht kann nur korrekt sein, wenn sie nicht verloren oder fehlerhaft ist.

$$M, s \models A\Box \texttt{not VMC.msg.Correct} \Rightarrow \texttt{VMC.msg.Lost or VMC.msg.Error} \tag{A.2}$$

$$M, s \models A\Box \texttt{VMC.msg.Correct} \Rightarrow \texttt{not VMC.msg.Lost and not VMC.msg.Error} \tag{A.3}$$

— Vertraulicher Inhalt wurde aus dieser Version entfernt —

B. Modell des Protokolls

B.1. Uppaal Automaten

— Vertraulicher Inhalt wurde aus dieser Version entfernt —

Abbildung B.1.: UCES

— Vertraulicher Inhalt wurde aus dieser Version entfernt —

Abbildung B.2.: VMC

B.2. Modelldeklarationen

— Vertraulicher Inhalt wurde aus dieser Version entfernt —

B.3. Quellcode

— Vertraulicher Inhalt wurde aus dieser Version entfernt —

Abbildung B.3.: Generierter Code zum Zugriff auf Modellelemente

Teil VI.

Verzeichnisse

Abkürzungsverzeichnis

CFG Kontrollflussgraph, engl. Control Flow Graph

CTL Computational Tree Logic

CTS Clear to Send

GUI grafische Oberfläche, engl. Graphical User Interface

LTL Linear Time Logic

RS-232 Spezifikation zur Kommunikation über eine Serielle Schnittstelle

RTS Request to Send

RxD Empfangsleitung im RS-232 Standard

TCTL Timed Computational Tree Logic

TP Touch Panel

TxD Sendeleitung im RS-232 Standard

UCES Produktname des medizinischen Controllers von Olympus

UI Benutzerschnittelle, engl. User Interface

VMC Produktname des nicht-medizinischen Controllers von Olympus

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[BFK+98] Howard Bowman, Giorgio Faconti, Joost-Pieter Katoen, Diego Latella, and Mieke Massink. Automatic verification of a lip synchronisation algorithm using Uppaal. In *Proceedings of the 3rd International Workshop on Formal Methods for Industrial Critical Systems*, pages 97--124, 1998.

[BGK+02] Johan Bengtsson, W. O. David Griffioen, Kåre J. Kristoffersen, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Automated verification of an audio control protocol using Uppaal, July-August 2002.

[BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices International Workshop*, pages 49--69, Marseille, France, March 2004. Springer.

[BM97] Johan van Benthem and G. B. Alice ter Meulen, editors. *Handbook of Logic and Language*. MIT Press, Cambridge, MA, USA, 1997.

[CAB+98] William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Francesmary Modugno, David Notkin, and Jon D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):156--166, 1998.

[CCG03] Sagar Chaki, Edmund Clarke, and Alex Groce. Modular verification of software components in C. In *IEEE Transactions on Software Engineering*, pages 385--395, 2003.

[CCI90] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13--17, January 1990.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 439--448. ACM Press, 2000.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, Anaheim, CA, USA, October 2003.

[Dai11] Robert Daigneau. *Service Design Patterns: Fundamental Design Solutions for Soap/Wsdl and Restful Web Services*. Pearson Education, 2011.

[GB06] Kim G. Larsen Gerd Behrmann, Alexandre David. *A Tutorial on UPPAAL 4.0*. Department of Computer Science, Aalborg University, Denmark, November 2006.

[GH04] Martin Giese and Rogardt Heldal. From informal to formal specifications in UML. In *Proceedings of UML2004*, volume 3273 of LNCS, pages 197--211. Springer, 2004.

[GLMR05] Guillaume Gardey, Didier Lime, Morgan Magnin, and Olivier H. Roux. Roméo: A tool for analyzing time Petri nets. In *Proceedings of Computer Aided Verification, CAV'05*, volume 3576, pages 418--423. Springer, 2005.

[Gor11] Pieter Van Gorp. Model-driven development of model transformations. In *International Conference on Model Transformation*, pages 47--61, 2011.

[Gou93] Mohamed G. Gouda. Protocol verification made simple: A tutorial. *Computer Networks and ISDN Systems*, 25(9):969--980, 1993.

[GQ11] Ganesh Gopalakrishnan and Shaz Qadeer, editors. *Computer Aided Verification, CAV'11*, volume 6806 of *Lecture Notes in Computer Science*, Snowbird, UT, USA, July 2011. Springer.

[HMR02] Michael Heymann, George Meyer, and Stefan Resmerita. Analysis of

zeno behaviors in hybrid systems. In *Proceedings of the 41st IEEE Conference on Decision and Control*, pages 2379--2384, Las Vagas, NV, 2002.

[Hol97] Gerard J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23:279--295, 1997.

[Joh05] Kristofer Johannisson. Formal and informal software specifications. Technical Report 6, Department of Computer Science and Engineering Chalmers University of Technology and Göteborg University, 2005.

[LLM07] Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159--189, June 2007.

[MNS95] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: bridging the gap between source and high-level models. *SIGSOFT Softw. Eng. Notes*, 20(4):18--28, October 1995.

[PNW11] Lee Pike, Sebastian Niller, and Nis Wegmann. Runtime verification for ultra-critical systems. In *Proceedings of the 2nd International Conference on Runtime Verification*, LNCS. Springer, September 2011.

[RH08] Linda H. Rosenberg and Lawrence E. Hyatt. Software re-engineering. Technical Report 1, Software Assurance Technology Center Unisys Federal Systems, 2008.

[Ste08] Asher Sterkin. State-oriented programming. *6th International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL)*, July 2008.

[Wan04] Farn Wang. Formal verification of timed systems: A survey and perspective. In *Proceedings of the IEEE*, volume 92, pages 1283--1305, August 2004.