Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorgelegte Arbeit mit dem Titel *Konzeption einer grafischen Nutzeroberfläche zur prozessorientierten Modellierung und Simulation* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Direkt oder indirekt übernommene Gedanken Dritter sind mit exakter Quellangabe gekennzeichnet.

Weiterhin erkläre ich, dass diese Arbeit keiner anderen Prüfungsbehörde weder in gleicher noch ähnlicher Form vorgelegt oder veröffentlicht worden ist.

Dresden den 14. November 2011

Martin Domnick

Nutzungsrecht

Hiermit erteile ich der Hochschule für Technik und Wirtschaft Dresden ein uneingeschränktes, zeitlich unbegrenztes, unwiderrufliches und übertragbares Nutzungsrecht für alle mit dieser Diplomarbeit verbunden Medien, sowohl für schutzfähige Ergebnisse als auch für urheberrechtlich schutzfähige Programme meiner Diplomarbeit.

Dresden den 14. November 2011

Martin Domnick

Inhaltsverzeichnis

Abkürzungsverzeichnis

DES	Discrete Event Simulation
FIFO	First In First Out
HTW	Hochschule für Technik und Wirtschaft
IL	Intermediate Language
LIFO	Last In First Out
SDK	Software Development Kit
WPF	Windows Presentation Foundation
XAML	Extensible Application Markup Language
XML	Extensible Markup Language

Glossar

Aktivität

Knotenpunkt im Modellnetzwerk, welcher von Entitäten durchlaufen werden kann. Durch Interaktion mit durchlaufenden Entitäten, ist eine Einflussnahme auf Simulationszustände möglich.

Ansichtsschicht

Über die Ansichtsschicht werden WPF-Benutzersteuerelemente, aus welchen sich komplexe Simulationsmodelle zusammensetzen lassen, bereitgestellt. Streng genommen handelt es sich hierbei um die Assembly *SimNetUI.DLL*. Applikationsentwickler kommen nur mit der Ansichtsschicht in Berührung.

Assembly

Die Assembly ist das Ergebnis aus der Kompilierung des C# Programmcodes und liegt als IL-Code vor. Eine Assembly kann sowohl eine DLL- oder auch eine EXE-Datei sein. Erst zur Laufzeit wird der IL-Code der Assembly durch den sogenannten Jitter in nativen Maschinencode übersetzt.[1]

Data-Binding

Eine Technologie, die in der WPF eingesetzt wird, um zwischen beliebigen Properties eine Datenbindung aufzubauen, wodurch beide Properties stets die selben Daten besitzen. Es wird zwischen einer Quelle und einem Ziel unterschieden. Änderungen an der Quelle bewirken zugleich auch immer Änderungen am Ziel. Je nach Einstellung kann dieser Prozess auch umgekehrt werden und ebenso für beide Seiten gleichzeitig geltend gemacht werden.

Deadlock

Ein Deadlock tritt auf, wenn zwei Aufgaben (Tasks) oder mehr gegenseitig voneinander abhängig sind, indem jede Aufgabe auf die Fertigstellung der jeweilig anderen Aufgabe wartet. Die Folge ist eine Blockade aller abhängigen Aufgaben.[2]

Dependency-Object

Die Klasse *DependencyObject* ist eine wichtige Basisklasse in der WPF, da sie die Verwendung von Dependency-Properties ermöglicht. Alle Klassen in der WPF erben von *DependencyObject*.[3]

[1] Vgl. [Küh10] Kapitel 1.3
[2] Vgl. [Fre10] S.
[3] Vgl. [Nat10] S. 74

Dependency-Property

Dependency-Properties werden überall in der WPF verwendet und ermöglichen die Nutzung von Styles, Data-Binding sowie Animationen. In der Praxis handelt es sich hierbei um normale .Net Properties, die unter Verwendung der WPF-API in die Infrastruktur der WPF integriert sind.[4]

Diskrete Event Simulation

Bei der (diskreten) ereignisorientierten Simulation wird ein System durch eine Folge von Ereignissen modelliert. Es werden somit nur Zeitpunkte erfasst, bei welchen sich der Zustand eines Systems verändert hat.[5]

Enhancer

Ein Programm, welches den IL-Code von Anwendungen, die auf die SimNet-Bibliothek zurückgreifen, erweitert.

Entität

Objekt, welches sich durch ein Modellnetzwerk während einer Simulation bewegt. Die grafische Darstellung von Entitäten kann variieren und wird gerne als Mittel genutzt, um unterschiedliche Zustände anzuzeigen. Das Simulationspaket Simul8 verwendet statt der Bezeichnung Entität den Begriff *Work-Item*.

Modellschicht

Bei der Programmbibliothek *SimNetUI.ModelLogic.DLL* handelt es sich um eine Softwareschicht der SimNetUI-Bibliothek, welche Applikationsentwicklern verborgen bleiben soll. Die Modellschicht besitzt ihre eigene Repräsentation des Simulationsmodells und dessen Zustände und enthält Programmcode, der für die Ausführung von Computersimulationen notwendig ist.

SimNet

SimNet ist der Name einer Programmbibliothek, welche an der HTW-Dresden im Rahmen einer früheren Diplomarbeit entwickelt worden ist. Diese Bibliothek dient als Grundlage für die aus dieser Arbeit resultierte Eigenentwicklung der Programmbibliothek SimNetUI. Die SimNet-Bibliothek integriert neue Sprachelemente in C#, die die Programmierung von ereignisorientierten Simulationen ermöglichen. Als Vorbild für eigene Spracherweiterungen dient die Programmiersprache Modsim III.

[4] Vgl. [Nat10] S. 80-81
[5] Vgl. [Rob04] S. 15

SimNetUI

Die zu dieser Diplomarbeit zugehörige Programmbibliothek für die grafische Modellierung/Entwicklung von ereignisorientierten Simulationen. Da die SimNetUI-Bibliothek aus mehreren Softwareschichten besteht, welche alle als separate Assembly realisiert worden sind, wird diese Bezeichnung als ein Überbegriff für die Zusammenfassung der Gesamtheit der hier aufgelisteten Komponenten genutzt.

- SimNetUI.DLL
- SimNetUI.ModelLogic.DLL
- SimNetUI.VisualStudio.Design.DLL

Thread

Ein Thread bezeichnet als Teil eines Prozess einen Ausführungsstrang eines Programms. Ein Prozess kann aus mehreren Threads bestehen, welche zeitgleich und unabhängig voneinander arbeiten können. Alle Threads eines Prozesses teilen sich den selben Hauptspeicher .

1 Einleitung

Heutzutage ist es häufig eine Notwendigkeit, Geschäftsprozesse mittels Computersoftware zu simulieren, da Sachverhalte und Abläufe innerhalb großer Systeme in der Industrie meist nicht ohne weiteres überschaubar sind. Unternehmen nutzen daher oft bei ihrer Entscheidungsfindung Computersimulationen, um Auswirkungen von Änderungen innerhalb komplexer Systeme besser nachvollziehen zu können. Dabei können, ohne hohe Kosten zu verursachen, verschiedene Szenarien durchexerziert werden. Eine übliche Vorgehensweise ist es, ausgehend von bekannten Größen aus der realen Umgebung ein Model für eine Computersimulation zu entwickeln. Zielstellung ist es, ein solches Model mit möglichst wenig Aufwand so nah wie möglich der Realität anzunähern. Hat man ein solches Model, ist man in der Lage, alternative Szenarien zu simulieren, womit letztendlich auch Schlussfolgerungen für die Realität vorgenommen werden können.

In diesen Bereich der Simulationen gibt es eine Vielzahl von unterschiedlichen Ansätzen. Der Fokus dieser Diplomarbeit liegt allerdings auf ereignisorientierter Simulation (engl. Diskrete Event Simulation). Dies ist ein Gebiet, welches bereits auf einige Jahrzehnte Forschung zurückblicken kann. Seit den 1960er Jahren stehen spezialisierte Programmiersprachen für die Erstellung von Computersimulation zur Verfügung. Einige bekannte Vertreter darunter sind u.a. GPSS (ab 1961)[1] und Modsim. Durch die Kombination von bewährten Programmierkonzepten (wie z.B. der objektorientierten Programmierung) mit der DES sind diese Programmiersprachen dank ihrer Mächtigkeit und Simplizität hochgradig geeignet für die Modellierung komplexer Systeme, wie sie in der Realität vorkommen.

Mit dem Einzug grafischer Bedienkonzepte in Betriebssysteme ab ca. Mitte der 1980er Jahre hat sich auch in der Welt der Computersimulationen einiges verändert. Auch auf dem Gebiet der Discrete Event Simulation (DES) haben sich insbesondere in den letzten zwei Jahrzehnten grafische Systeme zunehmend durchsetzen können. Ein entscheidender Vorteil solcher Systeme ist deren einfache Zugänglichkeit, womit auch Nicht-Informatiker ein leistungsstarkes Werkzeug an die Hand bekommen. Da mit grafischen Mitteln allein komplexe Sachverhalte nicht immer korrekt abgebildet werden können, handelt es sich bei diesen Softwaresystemen tatsächlich aber häufig um Hybridsysteme, welche mit Skriptsprachen ausgestattet es erlauben, komplexe Sachverhalte mittels Logikbausteinen auszudrücken. Mit Hilfe dieser Softwaresysteme lassen sich überdies besonders einfach Sachverhalte grafisch in Form von Animationen virtualisieren.

[1] Vgl. [Gas05] S. 131

Ein großer Nachteil vieler am Markt vorhandenen Lösungen für grafische DES-Systeme ist deren proprietäre Geschlossenheit zu anderen Systemen. Obwohl es sehr oft Möglichkeiten gibt, Daten von externen Quellen zu importieren bzw. zu exportieren, ist dies meist nicht ausreichend, um wie in der Informatik üblich eine Kompatibilität zu möglichst vielen verschiedenen Komponenten zu erreichen. Solche Einschränkungen sind aber in aller Regel, wenn es um die Einbettung in komplexe IT-Systeme geht, nicht hinnehmbar.

Um eine größere Flexibilität in dieser Angelegenheit zu erreichen, und um weitestgehend unabhängig von teuren proprietären System zu werden, ist es Zielstellung dieser Diplomarbeit einen ersten minimalen Prototyp eines grafischen DES-Systems als eingebettetes System für die Entwicklungsumgebung Visual Studio zu entwerfen. Dabei wird eine möglichst enge Verzahnung mit der Windows Presentation Foundation (WPF) angestrebt, welche die derzeit modernste Technologie von Microsoft darstellt, um grafische Oberflächen für das Betriebssystem Microsoft Windows zu entwickeln.

Ausgehend von der Diplomarbeit *Untersuchung zur Einbettung von Sprachelementen der prozessorientierten Simulation in C# unter .Net* aus dem Jahre 2005 des ehemaligen Studenten Torsten Wondrak, ist an der Hochschule für Technik und Wirtschaft (HTW)-Dresden ein Erweiterung der Programmiersprache C# entstanden, welche seither kontinuierlich weiterentwickelt wird. Grundlegende Eigenschaften, die für ein DES-System fundamental wichtig sind, wurden in dieser Arbeit bereits realisiert. Daher erscheint es sinnvoll, diese Vorarbeit als Grundlage für dieses Projekt zu verwenden.

Eigenschaft	Beschreibung
Carbon	Der Treibhauseffekt wird durch die Festlegung der erzeugten CO2 Emissionen berücksichtigt. Es kann ein fester Wert für jede Einheit einer Ressource festgelegt werden. Außerdem kann die Erzeugung von CO2 Emissionen je Zeiteinheit, für die eine Ressourceneinheit belegt ist, definiert werden.
Results	Es werden Statistiken über die Auslastung und Reisezeiten für alle Einheiten der Resource erfasst.

Tabelle 2.6.: Eigenschaften einer *Ressource* in Simul8

2.3.2.2 Bewertung

Die Anzahl der Aktivitäten im Simulationspaket Simul8 ist recht überschaubar. Insgesamt gibt es lediglich sechs verschiedene Bausteine, wovon die fünf wichtigsten in dem vorangegangen Abschnitt betrachtet worden sind. Dies hat zur Folge, dass es kaum Elemente gibt, auf die man für ein Simulationsmodell verzichten wird. Ein Vorteil ist sicherlich die schnellere und unkompliziertere Erlernbarkeit der Anwendung. Allerdings muss auch berücksichtigt werden, dass evtl. umfangreiche Simulationen intensiver kommentiert werden müssen. Denn die Wirkungsweise einzelner Komponenten ist nicht in jedem Fall selbsterklärend, da vieles was in anderen Simulationspaketen wie z.B.: Simprocess über zusätzliche Aktivitäten (u.a. Routing, Assembling) realisiert wird, in einzelne Simulationsbausteine von Simul8 direkt integriert ist. Abgesehen von Quellen und Senken ist ein Simulationsmodell in Simul8 somit hauptsächlich aus Warteschlangen und *Work Center* aufgebaut. Ein weiterer Nachteil, der aus dieser Herangehensweise resultiert, ist, dass nicht jede Aktivität mit jeder anderen Aktivität gleich gut zusammenarbeitet. So sind viele Optionen des *Work Centers* für die Zusammenarbeit mit Warteschlangen ausgelegt. Einige dieser Optionen machen im Zusammenhang mit anderen Komponenten allerdings keinen Sinn und können sogar für Deadlocks sorgen. Es ist anzumerken, dass aus dieser Betrachtung einige Erkenntnisse für die angestrebte Eigenentwicklung gewonnen werden konnten. Viele der hier vorgestellten Funktionalitäten, die in Simul8 existieren, werden auch ihren Weg in die aus dieser Arbeit hervorgehende Softwarelösung finden. Allerdings erscheint es mit Bezug auf die eigene Implementation als ein konsistenterer Ansatz, Funktionalitäten auf mehrere Simulationsbausteine aufzuteilen. Dies erleichtert insbesondere auch die Programmierung der Kommunikation der verschiedenen Komponenten untereinander, womit die Endlösung weniger fehleranfällig sein sollte.

2.3.3 Simprocess

In Simprocess werden Token, die sich durch das Simulationsmodell bewegen als Entitäten (engl. Entity) bezeichnet.

Viele Aktivitäten besitzen ähnliche Eigenschaften, sodass die allgemeinen Besonderheiten vor der Betrachtung der einzelnen Aktivitäten in diesem Abschnitt vorweggenommen werden sollen.

Für die Kommunikation untereinander besitzen Aktivitäten sogenannte Pads. Dabei kann eine Aktivität mehrere Input-Pads sowie Output-Pads mit unterschiedlichen Aufgaben besitzen. Input-Pads erlauben eine unbegrenzte Anzahl an eingehenden Verbindungen, während die meisten Output-Pads hingegen nur auf eine ausgehende Verbindung beschränkt sind. Es gibt allerdings auch Ausnahmen, wie beispielsweise bei der Split-Aktivität, die in Kapitel 2.3.3.1.9 auf Seite 23 näher beschrieben wird.

Eine elementare Eigenschaft der meisten Aktivitäten ist es, Simulationszeit verbrauchen zu können, indem die Dauer eines Prozesses, der durch die Aktivität repräsentiert wird, mittels einer Wahrscheinlichkeitsverteilung beschrieben wird. Eine Besonderheit in Simprocess ist, dass es keinen extra Baustein für Warteschlangen gibt. Stattdessen ist in allen Aktivitäten, die Simulationszeit verbrauchen können, eine Warteschlange integriert. Generatoren stellen in diesem Zusammenhang eine Ausnahme dar, da diese als Quelle keine Input-Pads besitzen.

Auch in Simprocess spielen Ressourcen eine wichtige Rolle. Aktivitäten können von Ressourcen abhängig sein und ihre Arbeit blockieren, bis ihnen die nötigen Ressourcen zur Verfügung stehen. Zudem können für Ressourcen ebenfalls Kosten definiert werden, womit diese auch als ein Werkzeug für die Kalkulation von Lohn und Betriebskosten eingesetzt werden können. Mittels eines sogenannten *Schedules* lässt sich überdies die Verfügbarkeit von Ressourcen einschränken, um beispielsweise Arbeitszeiten zu simulieren.

Simprocess erlaubt es für Entitäten, Aktivitäten und Ressourcen eine Vielzahl von Statistiken zu sammeln. Ausgewählte Statistiken können nach einem Simulationslauf in Form eines Protokolls in einem Texteditor betrachtet werden.

Die folgende Tabelle 2.7 fasst die Einzelheiten für die in dieser Arbeit betrachteten Aktivitäten aus der Simulationssoftware Simprocess zusammen.

2.3.4.2 Bewertung

Micro Saint Sharp bietet einen allgemeineren Ansatz beim Modellaufbau als die bisher vorgestellten Softwarepakete. Prinzipiell kann durch den gewählten Ansatz beim Modellaufbau und der Programmiersprache C# jeder Sachverhalt in Form eines Task-Netzwerkes abgebildet werden. Für die Realisierung vieler Anwendungsfälle steigt aber auch der Aufwand beim Aufbau des Modells. Denn die allermeisten Dinge müssen letztlich über Code realisiert werden, womit der eigentliche Task-Graph nur begrenzt lesbar ist. Für die Virtualisierung durch eine Animation stellt Micro Saint Sharp dafür aber spezialisierte Werkzeuge zur Verfügung, welche jedoch bei der Betrachtung des eigentlichen Modellaufbaus und deren Bewertung an dieser Stelle keine Rolle spielen. Für die Eigenimplementierung wird hingegen ein deklarativerer Ansatz mittels mehrerer unterschiedlicher Modellbausteine angestrebt. Allein die Integration in das Entwicklungsumfeld von Visual Studio erfordert aber von Entwicklern Kenntnisse in der Programmierung. Insofern werden auch einige Konzepte aus der Software Micro Saint Sharp übernommen, wenn es darum geht, zusätzlichen Programmcode über Modellbausteine auszuführen. Ebenso wie bei der Simulationssoftware Simprocess wurde der Ansatz einer integrierten Warteschlange auch für die Tasks in Micro Saint Sharp gewählt. Im Vergleich zu Simul8 tritt damit auch nicht das Problem auf, dass Entitäten verloren gehen können, weil keine Warteschlangen verwendet werden, die Entitäten (in diesem Fall *Work Items* genannt) auffangen und für deren weitere Verarbeitung bereithalten.

3 Elementare Konzepte des Modellentwurfs

In einem modernen Softwareentwicklungsprozess folgt die Konzeption meist direkt auf die Analyse. Denn ohne ein fundiertes Konzept sind die meisten komplizierten IT-Projekte zum Scheitern verurteilt. Hierbei geht es darum, die Erkenntnisse aus der Analyse auszuwerten und im Kontext der geplanten Entwicklung sinnvoll auf das zu erstellende System umzusetzen.

Bei der von dieser Arbeit ausgehenden Software handelt es sich um einen Prototyp, der selbstverständlich nur einen Bruchteil der Funktionalitäten abbilden kann, die für einen professionellen Einsatz von Bedeutung sind. Dies ist auch leicht nachzuvollziehen, wenn man bedenkt, dass an den derzeit etablierten Lösungen am Markt ganze Gruppen von Informatikern über Jahre hinweg gearbeitet haben. Was zugleich aber auch ein beachtlicher Vorteil ist, denn auf diese Weise lassen sich Erkenntnisse für die Eigenentwicklung übernehmen oder gar verbessern.

Es wäre naiv davon auszugehen, dass sich Ansichten über konkrete Konzepte über die Zeit hinweg nicht verändern können. Dies ist zum einen der Tatsache geschuldet, dass sich Erkenntnisse im laufe der Zeit ganz natürlich weiterentwickeln und somit alte Ideen in ein anderes Licht rücken lassen. Zudem ist nicht zu vergessen, dass ein Softwaresystem sich meist in eine komplexere IT-Struktur, welche als Unterbau dient, eingliedern muss und somit teilweise auch technische Gegebenheiten dazu führen können, dass Konzepte überarbeitet werden müssen. Insofern sind die Konzepte, die in diesem Kapitel besprochen werden, Ergebnisse dieses ganz natürlichen Prozesses. Es sei also angemerkt, dass im Rahmen der sich stetig veränderten Anforderungen an ein Softwaresystem, regelmäßige Restrukturierungsarbeiten eine Notwendigkeit darstellen. Da es sich in diesem Fall um einen Prototypen handelt und nicht um eine Software, die vom ersten Tag an für die kommerzielle Softwareentwicklung eingesetzt wird, ergibt sich der Luxus, dass Refaktorierungsarbeiten ohne Rücksicht auf Client-Code ausgeführt werden können.

An dieser Stelle erscheint es als äußerst sinnvoll, eine Unterscheidung zwischen einer Konzeption, die die Elemente die zur Beschreibung eines Simulationsmodells notwendig sind zum Gegenstand hat und einer rein technischen Konzeption zu treffen. Ersteres soll Gegenstand dieses Kapitels werden. Somit werden in diesem Kapitel lediglich diese Basiskonzepte, die zur Verwendung der aus dieser Arbeit hervorgehenden Bibliothek notwendig sind, vermittelt. Die technische Konzeption, die dann zur technischen Realisierung geführt hat, wird im nachfolgenden Kapitel erörtert.

Im Kapitel 2.3.1 ist bereits der grundlegende Aufbau eines Simulationsmodells beschrieben worden. Dieser hat sich über die Jahre hinweg bewährt und wird in beinah allen Simulationsprogrammen auf die eine oder andere Art umgesetzt. Folgerichtig und somit am sinnvollsten ist es von diesem Aufbau nicht weiter abzuweichen. Dessen ungeachtet ist es nicht verkehrt über Erweiterungskonzepte nachzudenken. Die vorgestellten Grundkonzepte stellen daher auch für die Eigenentwicklung, die aus dieser Diplomarbeit hervorgeht, das theoretische Fundament dar.

3.1 Modellcontainer

Mit der WPF hat Microsoft ein sehr mächtiges Werkzeug geschaffen, mit welchem eine produktive Entwicklung von grafischen Nutzeroberflächen für das Betriebssystem Windows erleichtert wird.

Die aus dieser Arbeit hervorgehende *SimNetUI*-Bibliothek ergänzt die WPF um Komponenten, die eine grafische Entwicklung eines Simulationsmodells ermöglichen. Um eine Separierung von den sonstigen Komponenten einer Nutzeroberfläche zu erreichen, sieht das Konzept für den Modellaufbau ein eigenes Container-Control vor, welches ein komplettes Simulationsmodell beinhaltet. Ein entscheidender Vorteil, der mit diesem Ansatz einhergeht, ist, dass auf diese Weise relativ einfach zentrale Einstellungen für ein Simulationsmodell über den Container vorgenommen werden können. Beispielsweise lassen sich auf diese Weise Aktionen, wie das Starten und Beenden einer Simulation, bequem über Kommandos bereitstellen. Das Container-Control wird von dem aus der WPF bekannten Canvas-Control abgeleitet und erbt damit all dessen Eigenschaften zur Positionierung von WPF Steuerelementen. Diese Flexibilität kann zum Beispiel genutzt werden, um das Simulationsmodell zu dokumentieren bzw. grafisch mit WPF Bordmitteln aufzuwerten.

kann. Weiterhin wird damit das Problem umgangen, dass Warteschlangen häufig nur im Kontext mit bestimmten Aktivitäten überhaupt sinnvoll eingesetzt werden können. Ein Missbrauch eines solchen Bausteins ist somit bereits ausgeschlossen.

Die für die SimNetUI-Bibliothek realisierten Sortierungsvarianten für Warteschlangen können der Tabelle 3.6 entnommen werden.

Variante	**Beschreibung**
FIFO	*First In First Out* stellt die gängigste Sortierung dar. Entitäten verlassen die Warteschlange in der Reihenfolge, wie sie diese betreten haben.
LIFO	Die Einstellung *Last In First Out* sorgt dafür, dass Entitäten in der umgekehrten Reihenfolge ihres eintreffens die Warteschlange verlassen.
PRIORITY_FIFO	Bei Wahl dieser Variante werden Entitäten zunächst *absteigend* nach ihrer Priorität sortiert. Entitäten mit der gleichen Priorität werden nach dem FIFO Prinzip innerhalb ihrer Gruppe sortiert.
PRIORITY_LIFO	Entitäten werden zuerst *absteigend* nach ihrer Priorität sortiert. Befinden sich mehrere Entitäten mit der gleichen Priorität in der Warteschlange, werden diese nach dem LIFO Prinzip innerhalb ihrer Gruppe angeordnet.
PRIORITY_DESC_FIFO	Bei Wahl dieser Variante werden Entitäten zunächst *aufsteigend* nach ihrer Priorität sortiert. Entitäten mit der gleichen Priorität werden nach dem FIFO Prinzip innerhalb ihrer Gruppe sortiert.
PRIORITY_DESC_LIFO	Entitäten werden zuerst nach *aufsteigender* Priorität sortiert. Entitäten mit gleicher Priorität werden nach dem LIFO-Prinzip sortiert.

Tabelle 3.1.: Sortierungsarten von Warteschlangen

3.2.1.2 Routing

Eine weitere markante Eigenschaft, die beinah jede Aktivität auszeichnet, mit Ausnahme der *Exit*-Aktivität, ist die Möglichkeit, Entitäten an Aktivitäten gezielt weiterzuleiten. In Form eines Events kann ein Entwickler eines Simulationsmodells genau

festlegen, welche von mehreren angeschlossenen Aktivitäten unter welchen Umständen das Ziel für eine ausgehende Entität darstellt. Ein Vorteil, der sich aus dem Verzicht einer zusätzlichen Aktivität für das Routing ergibt, ist , dass das Modell durch diese Maßnahme tatsächlich aufgeräumter erscheint, da wesentlich an Platz gespart wird. Weiterhin bleibt der Dokumentationseffekt gewahrt, denn es ist leicht zu erkennen, dass bei mehreren ausgehenden Verbindungen ein Routing erfolgen muss.

3.2.2 Verbindungen

Wie schon im Kapitel 2.3.1 erwähnt, repräsentieren Kanten jeweils eine Route zwischen verbundenen Aktivitäten.

Ein hierfür notwendiges Basiskonzept stellen die Verbindungsstücke dar, welche ähnlich wie die *Pads* aus Simprocess funktionieren. Theoretisch könnte eine Aktivität mehrere solcher Verbindungsstücke besitzen. Jedoch besitzen die realisierten Aktivitäten in der *SimNetUI*-Bibliothek derzeit maximal einen Eingang und einen Ausgang. Die Anzahl der ausgehenden oder eingehenden Verbindungen eines Verbindungsstücks können Beschränkungen unterliegen. Bei den derzeit implementierten Aktivitäten wird von dieser Möglichkeit allerdings noch kein Gebrauch gemacht.

Die Koordinaten für die Stützpunkte der Verbindungslinien werden direkt mittels XAML-Code beschrieben. Dies ist die flexibelste Lösung, da es damit dem Programmierer völlig freigestellt wird, wie dieser die Verbindungslinien anordnet. Lediglich Anfangs- und Endpunkt sind unveränderlich.

Da es sehr mühsam ist, den XAML-Code für solche Veränderungen selbst zu schreiben, hat es sich angeboten intensiven Gebrauch von den Erweiterungsmöglichkeiten des Visual Studio Designers zu machen[2]. Dank dieser Erweiterungen lassen sich Verbindungen relativ einfach mittels grafischer Werkzeuge erstellen. Diese Verbindungen können zudem in Aussehen und Form über ein Kontextmenü verändert werden. Für erweiterte Szenarien kann die hierfür wichtige Zeichenkette im XAML-Code aber weiterhin direkt angepasst werden.

XAML schreibt als Derivat des Extensible Markup Language (XML)-Standards eine hierarchische Dokumentstruktur vor. Damit eine Zuordnung von Verbindungen zu Aktivitäten möglich ist, werden diese sinnvollerweise der Aktivität zugeordnet, von welcher die Verbindung ausgeht. Im nachfolgenden Quelltext ist dies klar zu erkennen.

[2]Eine Abhandlung bezüglich der Verwendung der Erweiterungen für den Visual Studio Designer findet sich im Kapitel 3.8.

Eigenschaft	Beschreibung
Schedule	Der Schedule stellt einen Plan zur Erzeugung von neuen Entitäten dar. Es kann ein globaler Anfangs- sowie Endzeitpunkt definiert werden. Der eigentliche Plan besteht aus einer Liste von statistischen Verteilungsfunktionen, die mit einer fest definierten Dauer versehen werden können. Der Generator verwendet die in dieser Liste definierten Verteilungsfunktionen nacheinander, bis zum jeweiligen Ablauf der festgelegten Dauer. Falls der für den gesamten Schedule bestimmte Endzeitpunkt erreicht wird, bevor der Plan komplett abgearbeitet worden ist, wird die Erzeugung von weiteren Entitäten eingestellt.
Entity	Für die vom Generator zu erzeugenden Entitäten können deren Attribute angepasst werden. Dies kann dynamisch im Code erfolgen, nachdem das EntityLeft Ereignis ausgelöst worden ist oder direkt in der Modellbeschreibung im XAML-Code angepasst werden.

Tabelle 3.2.: Eigenschaften eines *Generators* aus der SimNetUI-Bibliothek

3.2.5.2 Exit

Die *Exit*-Aktivität wird in einem Simulationsmodell als Senke verwendet. Ankommende Entitäten verlassen das Simulationsmodell über solch einen Knotenpunkt.

Eigenschaft	Beschreibung
Entitäten Schranke	Für die *Exit*-Aktivität kann eine Schranke definiert werden, deren Erreichen zum unweigerlichen Ende der Simulation führt.

Tabelle 3.3.: Eigenschaften der *Exit*-Aktivität aus der SimNetUI-Bibliothek

3.2.5.3 Wait

Die *Wait*-Aktivität wird zum Simulieren von Prozessen verwendet, welche Simulationszeit verbrauchen. Diese Aktivität besitzt zudem eine integrierte Warteschlange, die Entitäten auffängt, die zur Zeit nicht zur Verarbeitung vorgesehen sind.

Eigenschaft	Beschreibung
Verarbeitungs-kapazität	Eine *Wait*-Aktivität ist imstande, mehrere Entitäten für einen Zeitraum zu blockieren. Die normale Kapazität ist auf eine Entität festgelegt. Für gewisse Anwendungsszenarien macht es aber durchaus Sinn, diese Grenze zu erhöhen
Distribution	Eine *Wait*-Aktivität ist imstande, mehrere Entitäten für einen Zeitraum festzuhalten. Die normale Kapazität ist auf eine Entität festgelegt. Für gewisse Anwendungsszenarien macht es aber durchaus Sinn, diese Grenze zu erhöhen.
Resourcen	Eine *Wait*-Aktivität kann von Ressourcen abhängig sein. Erst wenn alle notwendigen Ressourcen zur Verfügung stehen, nimmt diese Aktivität ihre Funktion wieder wahr. Eintreffende Entitäten werden bis dahin in der Warteschlange aufgehalten.

Tabelle 3.4.: Eigenschaften der *Wait*-Aktivität aus der SimNetUI-Bibliothek

3.2.5.4 Assign Resource

Mit der Aktivität *Assign Resource* wird es Entitäten ermöglicht, Ressourcen zu belegen. Sofern nicht alle definierten Ressourcen zur Verfügung stehen, werden eintreffende Entitäten in der Warteschlange solange aufgehalten, bis sich dieser Sachverhalt verändert hat.

3.2.5.5 Release Resource

Das Gegenstück zur Aktivität *Assign Resource* stellt die *Release Resource*-Aktivität dar. Diese Aktivität löst die Verbindung von Entitäten zu Ressourcen wieder auf. Es ist unbedingt zu empfehlen, dass für jede Ressource, die über eine *Assing Resource*-Aktivität Entitäten zugewiesen wird, eine *Release Resource*-Aktivität mit in das Modell übernommen wird. Andernfalls können sehr schnell Deadlocks im Simulationsmodell entstehen, da Ressourcen nicht mehr freigegeben werden.

3.3 Entitäten

Der Begriff Entität bezeichnet die Elemente, die sich während einer Simulation im Modell zwischen verbundenen Aktivitäten bewegen. Die konkrete Bedeutung, die einer

Softwareschicht das Fundament der SimNetUI-Bibliothek dar. Die SimNet-Bibliothek stellt Sprachelemente zur Verfügung, die die Entwicklung eines Simulationsmodells mit C# erleichtern. Durch Verwendung dieser Bibliothek müssen elementare Eigenschaften von Simulationssystemen für die SimNetUI-Bibliothek nicht selbst implementiert werden. Die SimNet-Bibliothek implementiert einen Scheduler, welcher sich selbständig um die Planung von Ereignissen kümmert. Zur Synchronisation von Ereignissen stehen Elemente wie Ressourcen, Trigger oder Interrupts zur Verfügung. Durch die Separierung beider Bibliotheken kann die Weiterentwicklung an unabhängig voneinander arbeitende Programmierer delegiert werden.

Den eigentlichen Kern der SimNetUI-Bibliothek stellen die beiden Komponenten *SimNetUI.ModelLogic* sowie *SimNetUI* dar. Entwickler, die auf Basis der SimNetUI-Bibliothek Anwendungen entwickeln, werden ausschließlich mit der Ansichtskomponente (*SimNetUI*) in Berührung kommen. Über diese Komponente werden Benutzersteuerelemente für die Entwicklung mit der WPF bereitgestellt. Die Modellschicht stellt eine interne Repräsentation des Simulationsmodells dar. Der eigentliche Programmcode, der für die Ausführung einer Simulation benötigt wird, ist somit in der Modellschicht zu finden. Eine Anwendung, die die SimNet-Bibliothek verwendet, muss auch stets durch den sogenannten Enhancer erweitert werden. Der Enhancer ist ein ergänzender Bestandteil zur SimNet-Bibliothek und wird in diesem Fall verwendet, um den Intermediate Language (IL)-Code der Modellschicht zu erweitern.

Ein interessanter Aspekt für die Entwicklung von Benutzersteuerelementen für die WPF ist die einfache Erweiterbarkeit des Visual Studio Designers. Um Zugriff auf diese Erweiterungsmöglichkeiten zu erhalten, war es notwendig, zur eigentlichen Bibliothek ein weiteres Projekt zu realisieren.[1]

4.1.1 Begründung für Trennung in eine Modell- und Ansichtsschicht

Die Unterteilung in eine *Modell-* und eine *Ansichtskomponente* bedarf weiterer Aufklärung.

Ein offensichtlicher Vorteil dieser Trennung ist zum einen eine Unterscheidung von verschiedenen Aufgabenbereichen. Während die Ansichtskomponente als Schnittstelle für Simulationsentwickler dient, kann sich die Modellkomponente um die internen Implementationsdetails kümmern. Letztlich ist es Zielstellung, einen Nutzer der SimNetUI-Bibliothek nicht mit unnötigen Interna zu belasten. Daher ist sehr viel Beachtung der

[1] Die Nutzung der Erweiterbarkeitsmechanismen der WPF ist Thema des Kapitels 4.3

Gestaltung der Zugriffsregeln erwiesen worden. Durch die Auslagerung von Code in eine eigenständige *Assembly* wird dieser Prozess erheblich vereinfacht.

Durch die Verwendung der SimNet-Bibliothek ergibt sich leider auch ein Nachteil. Denn aufgrund der Erweiterung des IL-Codes durch den Enhancer, ist es im Anschluss nicht mehr möglich, den C# Programmcode der Modellschichtkomponente zu debuggen. Gäbe es diese Separierung nicht, würde sich dieser Nachteil ebenso auf einen großen Teil des Programmcodes auswirken, der für die Bereitstellung der WPF-Komponenten der SimNetUI-Bibliothek zuständig ist. Weiterhin ist es zur Entwicklungszeit sehr nützlich, wenn Kompilierungszeiten möglichst kurz sind. Der Code der Modellschicht ist wesentlich kompakter als der Code für die Ansichtsschicht und kann daher vom *Enhancer* bedeutend schneller verarbeitet werden.

Der wichtigste Grund für die Gliederung des Codes ist die Notwendigkeit einer *multithreaded* Anwendung. Ereignisse im Kontext einer DES-Simulation sind stets zeitlich voneinander getrennt. Dass zwei Ereignisse zum selben Zeitpunkt ausgeführt werden, stellt daher eine extreme Ausnahme dar. Eine Parallelisierung des Simulationscodes wäre zudem viel besser, in der SimNet-Bibliothek unterzubringen. Die Aufteilung in unterschiedliche Threads hat vielmehr rein technische Gründe, die sich aus dem Zusammenspiel mit der WPF-Technologie ergeben. Zum einen wird eine Anwendung, die in der Lage ist, auf Eingaben auch während der Ausführungszeit des Simulationscodes zu reagieren, als sehr viel Nutzerfreundlicher empfunden.

Die eigentliche Hürde stellt allerdings die Art und Weise dar, wie Animationen in der WPF realisiert sind. Für Anwendungen, die auf Basis der WPF laufen, kann zwischen einem UI-Tread und einen Render-Thread unterschieden werden. Nutzercode wird in aller Regel im UI-Thread ausgeführt. Darüber hinaus können weitere Threads vom Anwendungsentwickler realisiert werden. Animationen werden über den Render-Thread der WPF ausgeführt, können aber im UI-Thread initialisiert werden. Zwischen dem UI-Thread und dem Render-Thread besteht eine Synchronisation, sodass immer dann, wenn Nutzercode ausgeführt wird, der nicht Bestandteil der WPF ist, der Render-Thread blockiert ist. Eine folgerichtige Konsequenz, die sich hieraus ergibt, ist, dass es für Anwendungsentwickler nicht möglich ist, auf das Ende der Ausführung einer Animation im UI-Thread zu warten, um im Anschluss weiteren Code auszuführen. Einen Ausweg aus dieser Situation schafft ein Ereignis, dass nach Beenden der Animation im UI-Thread ausgelöst wird. Das alleine genügt allerdings nicht, um das Problem vollständig zu lösen. Den je komplexer der Aufruf-Stack ist, um so schwieriger wird es den UI-Thread an gegebener Stelle wieder zu verlassen, um später an geeigneter Stelle wieder einzuspringen. Darum erscheint ein zusätzlicher Thread, der für die Zeit der Ausführung einer Animation blockiert wird, als sehr nützlich. Somit wird Pro-

5 Zusammenfassung und Ausblick

5.1 Ziellstellung

Mit dieser Diplomarbeit sollte untersucht werden, wie sich ein Programmierwerkzeug für die grafische Modellierung ereignisorientierter Systeme und der Simulation dieser Systeme in das Entwicklungsumfeld von Visual Studio 2010 optimal integrieren lässt.

5.2 Vorgehensweise

Da es auch für diesen speziell abgegrenzten Bereich der Computersimulation für eine Reihe von Aspekten unterschiedliche Konzepte gibt, ist eine umfassende Analyse notwendig gewesen. Hierzu sind einige am Markt etablierte Softwarelösungen betrachtet worden. Die Ergebnisse dieser Analyse sind ausgewertet worden, woraufhin eigene Konzepte für die Realisierung des Programmierwerkzeugs unter Berücksichtigung der Besonderheiten des gewählten Entwicklungsumfeldes entwickelt worden sind.

5.3 Ergebnis

Das Resultat, was aufgrund dieser Vorbereitungen erarbeitet worden ist, ist eine .Net-Bibliothek, die auf Basis der WPF-Technologie von Microsoft, Funktionalitäten bereitstellt, die eine einfache und unkomplizierte Entwicklung von ereignisorientierten Simulationen mit dem Entwicklungswerkzeug Visual Studio 2010 ermöglichen. Eine Stärke dieser Lösung ist unter anderem die ausdrucksstarke Präsentation von Modellzuständen über den Verlauf einer Simulation. Computersimulationen können ohne Bereitstellung der Entwicklungssoftware als Windows Anwendungen verbreitetet werden. Eine Möglichkeit, die die Eigenentwicklung von den untersuchten Simulationspaketen unterscheidet.

Der Text dieser Diplomarbeit soll als Grundlage für das Studium des erarbeiteten Resultates dienen.

5.4 Ansätze für künftige Erweiterungen

In der derzeitigen Form der Programmbibliothek, die als Resultat aus dieser Diplomarbeit hervorgegangen ist, ist diese nur für eine eingeschränkte Anzahl von Anwendungsfällen sinnvoll einsetzbar. Mit dieser Arbeit wurde daher lediglich eine Grundlage geschaffen. Auf diesem bereiteten Fundament kann aber eine Fortentwicklung stattfinden, um die Software vielseitiger einsetzbar zu gestalten. In diesem Zusammenhang ist zu erwähnen, dass es eine Anzahl von Anknüpfungspunkten gibt, die bestehende Softwarelösung zu erweitern.

Einige Bereiche in denen Erweiterungen oder Verbesserungen möglich sind, sollen nun aufgezählt werden.

5.4.1 Integration einer aktualisierten Version der SimNet-Bibliothek

Zeitgleich zur Entwicklung der SimNetUI-Bibliothek ist im Rahmen einer weiteren Diplomarbeit an der HTW-Dresden die SimNet-Bibliothek erneuert worden. Dies erklärt den Umstand, warum für die SimNetUI-Bibliothek bei Abschluss des Projektes nicht die neueste Version der SimNet-Bibliothek als Grundlage dient. Daher sollte für eine künftige Weiterentwicklung der SimNetUI-Bibliothek mit oberster Priorität die Integration der jeweiligen aktuellsten Version der SimNet-Bibliothek angestrebt werden.

Neben wichtigen Restrukturierungsmaßnahmen ist die SimNet-Bibliothek, im Rahmen der erwähnten Diplomarbeit, um Fähigkeiten ergänzt worden, von welchen insbesondere die SimNetUI-Bibliothek profitieren sollte. Beispielsweise ist die Möglichkeit geschaffen worden, Simulationen zu unterbrechen, um diese zu einen späteren Zeitpunkt wieder fortzusetzen. Ein großer Teil der Refaktorierungsmaßnahmen zielte auch auf die Beseitigung von Programmfehlern in der alten Version der SimNet-Bibliothek ab. Hier muss erwähnt werden, dass die Ursprüngliche Version der SimNet-Bibliothek bereits im Jahre 2005 entwickelt worden ist. Mittlerweile ist die Entwicklung der Programmiersprache C# weiter vorangeschritten, sodass Schwierigkeiten bei der Verwendung von neueren Programmiertechniken, welche 2005 noch nicht präsent waren, aufgetreten sind.

Weil einige Programmfehler in der alten SimNet-Version den Einsatz teilweise unmöglich gemacht haben, wurde die für dieses Projekt verwendete ursprüngliche SimNet-Version von einem Teil kritischer Programmfehler befreit. Da es einen regen Austausch zwischen dem Studenten, der die SimNet-Bibliothek weiterentwickelt hat, gab, sind all

diese Änderungen auch in der überarbeiten Version der SimNet-Bibliothek mit eingeflossen.

Zur Ausgangssituation kann also zusammenfassend gesagt werden, dass zwei grundverschiedene Versionen der SimNet-Bibliothek existieren. Eine Integration der überarbeiteten Version wird somit zwangsläufig Änderungen des Programmcodes der SimNetUI-Bibliothek insbesondere in der Modellschicht nach sich ziehen. Weiterhin wird es ebenso eine Notwendigkeit sein, dass die zu integrierende Version der SimNet-Bibliothek angepasst werden muss, um ein besseres Zusammenspiel mit der SimNetUI-Bibliothek zu erlauben. Generell wäre es eine Überlegung Wert, das *INotifyPropertyChanged*-Interface für Klassen der SimNet-Bibliothek zu implementieren, welche nach außen Properties zur Verfügung stellen. Für die Klasse *RessourceObj* aus der derzeitig verwendeten Version ist das *INotifyPropertyChanged*-Interface implementiert worden. Diese Änderung müsste in jedem Fall auch in die überarbeitete SimNet-Bibliothek übernommen werden.

5.4.2 Subnetzwerke

Computermodelle für ereignisorientierte Simulationen können im professionellen Umfeld schnell sehr große Dimensionen annehmen. In der Programmierung ist es von je her eine gewöhnliche Vorgehensweise, Probleme in Teilbereiche mit unterschiedlichen Aufgabenstellungen zu gliedern. Denn nach diesem Prinzip wird es sehr viel einfacher, komplexe Sachverhalte zu verwalten. Ist erst einmal eine Abgrenzung in verschiedene Teilaufgaben erfolgt, ist es außerdem möglich Redundanzen zu vermeiden, wenn an verschiedenen Stellen im Modell die selben Abläufe beschrieben werden.

Eine Implementation von Subnetzwerken in die bestehende Softwarelösung ist eine umfangreiche Aufgabe und erfordert ein teilweise Refaktorierung des gegenwärtigen Programmcodes, welcher der erstellten Software zu Grunde liegt.

Ein Ansatz der für eine nähere Betrachtung in Frage kommt, ist die Implementation von Subnetzwerken in Form eines eigenen *UserControls*. Benutzersteuerelemente spielen in der WPF bereits eine bedeutende Rolle und sind das Mittel der Wahl, um spezialisierte Elemente für die Oberfläche einer Anwendung zu entwickeln. Um eine flüssige Entwicklung zu gewährleisten, ist in diesem Fall aber auch die Verwendung von Erweiterungsmöglichkeiten der Visual Studio IDE sinnvoll. Die optimale Lösung wäre die Bereitstellung einer Vorlage, welche als neues Element über die üblichen Dialoge der Visual Studio IDE einem Projekt hinzugefügt werden könnte.

5.4.3 Aktivitäten

5.4.3.1 Verbesserungen bestehender Aktivitäten

Die Funktionalität bestehender Aktivitäten kann unter Umständen ausgebaut werden. Als sinnvolle Erweiterung kann für Generatoren die Möglichkeit geschaffen werden, zu einem Zeitpunkt *mehrere* Entitäten in das Simulationsmodell *gleichzeitig* eintreten zu lassen. Hier bietet es sich an, die selben Verteilungsfunktionen zu verwenden, wie auch bereits für die Planung von neuen Ereignissen.

5.4.3.2 Ergänzung neuer Aktivitäten

Nur eine begrenzte Auswahl von Aktivitäten sind für die Ursprungsversion der SimNetUI-Bibliothek realisiert worden. Allerdings gibt es viel Raum, um neue Aktivitäten zu implementieren. Eine erhöhte Vielseitigkeit in diesem Zusammenhang würde sich positiv in einer Erweiterung möglicher Einsatzgebiete bemerkbar machen. Als Grundlage für Aktivitäten, die für eine Fortentwicklung der Software in Betracht gezogen werden können, bietet sich das Studium der Analyse, deren Ergebnisse in dieser Diplomarbeit schriftliche festgehalten worden sind, an.

Folgende Aktivitäten, stellen sinnvolle Erweiterungen für den bereits bestehenden Satz dar.

- ➢ Eine Assemble-Aktivität, die sich an der Lösung aus der Software Simprocess orientieren kann. Sinnvoll wäre es, wenn sich diese Aktivität von der Basisklasse *ActivityDelayBase* herleiten würde.
- ➢ Eine Aktivität für das Klonen von Entitäten.
- ➢ Eine Aktivität zum Verschachteln oder Zusammenfassen von Entitäten.

5.4.4 Statistiken

5.4.5 Erfassung neuer Statistiken

Im Kapitel 3.6 ist bereits erwähnt worden, dass zum gegenwärtigen Zeitpunkt keine Statistiken über Entitäten erfasst werden. Folgende statistischen Werte sind aber von Interesse:

- ➢ Wartezeit in Warteschlangen insgesamt
- ➢ Mittlere Verweilzeit in Warteschlangen

- ➢ Kürzeste Wartezeit in einer Warteschlange
- ➢ Längste Wartezeit in einer Warteschlange
- ➢ Bearbeitungszeit insgesamt
- ➢ Mittlere Bearbeitungszeit
- ➢ Kürzeste Bearbeitungszeit
- ➢ Längste Bearbeitungszeit

Darüber hinaus sollten alle Statistiken, die über Ressourcen in der SimNet-Bibliothek erfasst werden, auch für die Verwendung in der SimNetUI-Bibliothek zur Verfügung gestellt werden. Zur Zeit der Niederschrift dieser Diplomarbeit, werden noch nicht alle Statistiken über die SimNetUI-Bibliothek Anwendungsentwicklern zur Verfügung gestellt.

5.4.6 Werkzeuge zur Auswertung einer Simulation

Bisher ist eine abschließende Auswertung von Simulationen alleinige Aufgabe des Programmierers, der sich der SimNetUI-Bibliothek bedient. Denkbar wäre die Bereitstellung eines Report-Controls, welches über eine Datenbindung mit dem Simulationscontainer verbunden ist. Nach einem Simulationslauf kann eine Analyse des Modells erfolgen, dessen Ergebnis in übersichtlicher Art und Weise in Form eines Reports tabellarisch aufbereitet werden kann. So können auch Statistiken von Entitäten dargestellt werden, die sonst während des Simulationslaufs für Simulationsentwickler nur schwer abgreifbar wären.

Die Analyse der Statistiken muss aber keineswegs am Ende der Simulation geschehen. Wenn beispielsweise die Anforderung gestellt ist, für einen festgelegten Zeitraum und einer Auswahl von Aktivitäten eine Zusammenfassung eines ausgewählten statistischen Wertes zu berechnen, ist eine kontinuierliche Analyse oder Sammlung von Daten erforderlich. Eine flexible Lösung für diese Problemstellung zu entwickeln, ist mit Sicherheit eine Herausforderung.

A Projektmappe Übersicht

Die CD, die dieser Diplomarbeit beiliegt, enthält eine Projektmappe für Visual Studio mit Projekten, welche bei der Bearbeitung des Diplomthemas *Konzeption einer grafischen Nutzeroberfläche zur prozessorientierten Modellierung und Simulation* erstellt bzw. bearbeitet worden sind.

A.1 SimNet

Das Projekt SimNet ist eine angepasste Version der SimNet-Bibliothek welche Ursprünglich aus der Diplomarbeit *Untersuchung zur Einbettung von Sprachelementen der prozessorientierten Simulation in C# unter .Net* hervorgegangen ist und seit dem am Fachbereich Informatik der HTW-Dresden weiterentwickelt wird. Dieses Projekt wird separat von der SimNetUI-Bibliothek entwickelt. Als diese Diplomarbeit ausgearbeitet wurde, war bereits eine aktualisierte Version der SimNet-Bibliothek in Arbeit. Daher handelt es sich bei der derzeit verwendeten Version nicht um die aller neueste Version. Ziel muss es sein, für künftige Versionen der SimNetUI-Bibliothek auf eine neue angepasste Version der SimNet-Bibliothek zu setzen.

A.2 Enhancer

Der Enhancer ist ein ergänzender Bestandteil der SimNet-Bibliothek. Dieser wird verwendet um den IL-Code von Projekten zu erweitern, welche die SimNet-Bibliothek verwenden.

A.3 SimNetUI

Die Ansichtsschicht der SimNetUI-Bibliothek ist im Projekt SimNetUI implementiert. Über die Ansichtsschicht werden Benutzersteuerelemente für die WPF bereitgestellt, mit deren Hilfe sich sehr einfach Simulationsmodelle entwerfen lassen.[1]

[1] Ausführliche Informationen zur Verwendung der Ansichtsschicht für Applikationsentwickler sind im Kapitel 3 zu finden. Informationen der Implementation betreffend können im Kapitel 4.2 nachgelesen werden.

A.4 SimNetUI.ModelLogic

Bei der Modellschicht handelt es sich um eine Softwareschicht, die sich um Implementationsdetails kümmert, die für Applikationsentwickler nicht einsehbar sein sollen. Die Modellschicht besitzt ebenso wie die Ansichtsschicht zur Laufzeit ein eigene Modellrepräsentation. Der Programmcode, der sich um die Simulationsaspekte kümmert ist in der Modellschicht unter Verwendung der SimNet-Bibliothek implementiert worden. Der IL-Code der Modellschicht wird zusätzlich durch den Enhancer erweitert.[2].

A.5 SimNetUI.VisualStudio.Design

Das Erweiterungsprojekt zur Ansichtsschicht stellt zusätzliche Werkzeuge für den Visual Studio WPF Designer *Cider* zur Verfügung.

A.6 SimNetUI.ProjectTemplate

Um einen schnellen Einstieg in die Entwicklung mit der SimNetUI-Bibliothek zu ermöglichen, ist ein Projekttemplate erstellt worden, welches beim Anlegen eines neuen Projektes als Vorlage verwendet werden kann.

A.7 SimNetUI.ProjectTemplate.Setup

Die Installation des Projekttemplates erfolgt über eine VSIX-Paket-Datei, welche das Projekttemplate als Addon für Visual Studio installiert. Nach der Installation ist die Projektvorlage im Dialog *Neues Projekt* auswählbar.

A.8 Example.Phonecell

Das Projekt *Example.Pohnecell* demonstriert den Einsatz der SimNetUI-Bibliothek.

Simuliert wird die Belegung von 4 Telefonzellen. In einem Zeitraum von 500 Minuten treffen mit einer Gleichverteilung (*UniformDouble*) von 30 Sekunden bis 10 Minuten

[2]Umfassende Informationen zur Implementation der Modellschicht sind im Kapitel 4.2 festgehalten worden.

B Debugging

B.1 Projektmappe Debugeinstellungen

Einstellung	Beschreibung
Debug	Alle Projekte in der Projektmappe werden mit den Standard-Debugeinstellungen kompiliert. Der IL-Programmcode der Modellschicht wird nicht durch den Enhancer erweitert. Daher sind Computersimulationen bei Verwendung dieser Einstellung nicht ausführbar. Diese Einstellung eignet sich, um Fehler im Programmcode der Modellschicht zu finden, da nach der Erweiterung des IL-Codes nur noch dieser lesbar ist. Denn der IL-Programmcode ist wesentlich unlesbarer als der C#-Programmcode und eignet sich daher nicht in jedem Fall zum Debuggen.
DebugEnhance	Alle Projekte in der Projektmappe werden mit Standard-Debugeinstellungen kompiliert. Für die Modellschicht wird der Enhancer zusätzlich zur Erweiterung des IL-Codes verwendet. Computersimulationen sind mit dieser Einstellung daher ausführbar.
Release	Alle Projekte in der Projektmappe werden mit Standard-Releaseeinstellungen kompiliert. Der IL-Programmcode der Modellschicht wird nicht durch den Enhancer erweitert. Somit können Computersimulationen bei Wahl dieser Einstellung nicht ausgeführt werden.
ReleaseEnhance	Alle Projekte in der Projektmappe werden mit den Standard-Releaseeinstellungen kompiliert. Für die Modellschicht wird der Enhancer zur Erweiterung des IL-Codes verwendet. Computersimulationen sind mit dieser Einstellung daher ausführbar. Diese Einstellung ist für die endgültige Auslieferung einer neuen Version der SimNetUI-Bibliothek zu wählen.

Tabelle B.1.: Debugeinstellungen

B.2 Erweiterungsprojekt

Um das Erweiterungsprojekt *SimNetUI.VisualStudio.Desing* für die Ansichtsschicht zu debuggen, muss dieses als Startprojekt im Projektmappen-Explorer der Visual Studio IDE definiert worden sein. Zum Debuggen wird eine 2. Instanz der Visual Studio IDE gestartet. In dieser Instanz sollte nun ein Projekt geöffnet werden, welches die SimNetUI-Bibliothek verwendet.

C Einrichtung eines Arbeitsumfeldes zur Fortentwicklung der SimNetUI-Bibliothek

Die Vorgehensweise zur Installation der notwendigen Entwicklungswerkzeuge, mit deren Hilfe eine Weiterentwicklung der SimNetUI-Bibliothek möglich wird, soll hier in aller Kürze beschrieben werden.

1. Installation der Entwicklungsumgebung Visual Studio 2010. Gegebenenfalls muss noch ein Servicepack eingespielt werden.

2. Installation des Visual Studio 2010 Software Development Kit (SDK). Hierbei ist zwingend darauf zu achten, die richtige Version mit dem korrekten Service Pack zu installieren.

3. Zur Disassemblierung sowie Assemblierung benötigt der Enhancer die beiden Hilfsprogramm *ildasm.exe* und *ilasm.exe*. Der Programmpfad beider Applikationen muss dem Windows Systempfad hinzugefügt werden.

   Die Programmpfade, unter welchen beiden Anwendungen zu finden sind, können je nach Installation variieren. Bei einer Windows 7 64 Bit Installation unter Verwendung von .Net 4.0 und Visual Studio 2010 sind die Anwendungen unter folgenden Programmpfaden zu finden:

   - ildasm.exe: C:\Windows\Microsoft.NET\Framework\v4.0.30319
   - ilasm.exe: C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\Bin\NETFX 4.0 Tools\x64

   .

   Unter Windows 7 kann der Dialog zur Bearbeitung der Umgebungsvariablen auf folgendem Weg erreicht werden:

   a) Den Pfad *Systemsteuerung\System und Sicherheit\System* in die Adressbar des Windows-Explorer kopieren

   b) Den Dialog für Erweiterte Systemeinstellungen durch einen Klick auf gleichnamigen Link öffnen

   c) Den Reiter *Erweitert* anwählen.

   d) Die Schaltfläche *Umgebungsvariablen* betätigen

Abbildungsverzeichnis

Tabellenverzeichnis

Auflistungen

Literatur

[05] *Micro Saint Sharp User Guide*. Micro Analysis und Design, Inc. 2005.

[08] *User's Manual Simprocess*. CACI International Inc. 2008.

[Fre10] Adam Freeman. *Pro .NET 4 Parallel Programming in C#*. Apress, 2010. ISBN: 978-1-4302-2968-1.

[Gas05] Saul I. Gass. *An Annotated Timeline Of Operations Research. An Informal History*. Kluwer Academic Publishers, 2005. ISBN: 1-4020-8116-2.

[Küh10] Andreas Kühnel. *Visual C# 2010: Das umfassende Handbuch*. 5. Aufl. Deutschland: Galileo Computing, 2010. ISBN: 978-3-8362-1552-7.

[MSD11a] MSDN. *Microsoft.Windows.Design.Model-Namespace*. 2011. URL: *http://msdn.microsoft.com/de-de/library/microsoft.windows.design.model%28v=VS.100%29.aspx* (besucht am 20. 10. 2011).

[MSD11b] MSDN. *WPF-Designer-Erweiterbarkeit*. 2011. URL: *http://msdn.microsoft.com/de-de/library/bb546938%28v=VS.100%29.aspx* (besucht am 14. 10. 2011).

[Nat10] Adam Nathan. *WPF 4 Unleasched*. 1. Aufl. Vereinigte Staaten von Amerika: Pearson Education, 2010. ISBN: 978-0-672-33119-0.

[Rob04] Stewart Robinson. *Simulation. The Practice of Model Development and Use*. John Wiley & Sons, Ltd, 2004. ISBN: 0-470-84772-7.

[Sto06] Bea Stollnitz. *How can I propagate changes across threads?* 23. Sep. 2006. URL: *http://bea.stollnitz.com/blog/?p=34* (besucht am 16. 09. 2011).

[Wil10] Shawn Wildermuth. „Build More Responsive Apps With The Dispatcher". In: *MSDN Magazine* (Okt. 2010). URL: *http://msdn.microsoft.com/en-us/magazine/cc163328.aspx* (besucht am 15. 09. 2011).

[Won05] Torsten Wondrak. *Untersuchung zur Einbettung von Sprachelementen der prozessorientierten Simulation in C# in .Net*. 2005.