

**Humboldt-Universität zu Berlin  
Institut für Informatik  
Lehrstuhl Systemanalyse**

# **Serialisierung von ODEMx Simulationsmodellen**

**Studienarbeit**

Vorgelegt von:  
Thomas Kipar

Betreuer:  
Dipl-Inf. Andreas Blunk  
Prof. Dr. Joachim Fischer

1. Dezember 2011

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>3</b>
<b>2</b>	<b>Das Problem: Verändern von Simulationszuständen</b>	<b>4</b>
2.1	Die Simulationsbibliothek ODEMX . . . . .	5
2.2	Ein Beispiel: Simulation einer Autofähre . . . . .	6
2.3	Bestandteile eines ODEMX-Modells . . . . .	7
2.4	Allgemeines Vorgehen und Einschränkungen . . . . .	8
<b>3</b>	<b>Direkte Serialisierung mit Boost::Serialization</b>	<b>10</b>
3.1	Serialisierung mit Boost . . . . .	10
3.2	Anwendung auf das Problem . . . . .	11
3.3	Möglichkeiten und Grenzen . . . . .	12
3.4	Serialisierung der Autofährensimation . . . . .	13
<b>4</b>	<b>Metamodellbasierte Serialisierung mit dem Eclipse Modeling Framework</b>	<b>16</b>
4.1	Das Ecore-Metamodell . . . . .	16
4.2	emf4cpp . . . . .	17
4.3	Das partielle ODEMX Metamodell . . . . .	18
4.4	Ecore-Metamodelle als ODEMX Simulationen . . . . .	19
4.5	Serialisierung von Simulationsmodellen . . . . .	20
4.6	Möglichkeiten und Grenzen . . . . .	21
4.7	Beispiel einer komplexen Zustandsänderung . . . . .	23
<b>5</b>	<b>Zusammenfassung</b>	<b>26</b>
	<b>Literaturverzeichnis</b>	<b>27</b>

# 1 Einleitung

Simulationen werden eingesetzt, um Phänomene zu untersuchen. Die erstellten Simulationen können helfen, ein tieferes Verständnis über das Phänomen zu erlangen, Zusammenhänge zu erkennen oder Vorhersagen über die Zukunft zu treffen. Ebenfalls können sie eingesetzt werden, um Teile von Systemen testen zu können.

Es ist dabei nötig, die Simulation mehrmals auszuführen und Ergebnisse auf Basis von Parametern, die verändert werden, zu analysieren. Dabei kann die Situation eintreten, dass gewisse Effekte nicht untersucht werden können, da die Simulation nicht entsprechend parametrisierbar ist. Es wäre in solchen Fällen also erforderlich, dass der Simulation zugrunde liegende Modell entsprechend anzupassen. Dies ist sowohl aufwändig als auch fehleranfällig.

Ein anderer, besserer Weg ist es, einen Modellzustand, der während eines Simulationslaufes erreicht wird, direkt zu verändern. So könnte die Modellkonfiguration und somit die weitere Ausführung beeinflusst werden, ohne das Modell verändern zu müssen. Bei einem Modellzustand könnten etwa einzelne Parameter verändert werden. Auch weitergehende, strukturändernde Eingriffe, wie etwa das Einfügen von neuen oder Unterbrechen bestehender Prozesse, sind denkbar.

Im Allgemeinen liegt der Modellzustand bei einer Simulation in einer bestimmten Form im Arbeitsspeicher des ausführenden Rechners vor. Damit eine Änderung am Zustand möglich ist, muss er jedoch in einer anderen Form dargestellt werden. Die Abbildung von Objekten in eine sequentielle Darstellungsform wird dabei als Serialisierung bezeichnet. Die sequentiellen Daten können anschließend etwa in eine Datei gespeichert werden. Gelingt dies in einem menschenlesbaren Format, wie XML, kann der Benutzer anschließend den serialisierten Zustand verändern. Der veränderte Zustand soll danach wieder geladen werden können und in die ursprüngliche Darstellung zurück umgewandelt werden. Dies wird Deserialisierung genannt.

Das Simulationsmodell soll dann von dem veränderten und geladenen Zustand aus weiter ausgeführt werden können. Auf dieser Art kann eine Änderung im Ablauf der Simulation untersucht werden, ohne das Modell selbst anpassen zu müssen. Dieses würde Domänenexperten ermöglichen, selbst die Ausführung der Simulation beeinflussen zu können, ohne das Expertenwissen zu besitzen, dass die Modellierung bzw. Programmierung der Simulation erfordert.

In dieser Arbeit wird untersucht, wie Simulationszustände serialisiert werden können. Daran vorgenommene Manipulationen sollen danach wieder geladen werden können. Die Simulation soll anschließend die Zustandsänderung berücksichtigen. Die Funktionalität soll hierbei für die Simulationsbibliothek ODEmx implementiert werden. Dabei soll darauf geachtet werden, dass die dazu notwendigen Verfahren möglichst generisch für alle sämtliche Simulationsmodelle eingesetzt werden kann.

## 2 Das Problem: Verändern von Simulationszuständen

Bei der Untersuchung eines realen oder gedachten Phänomens ist es oft hilfreich, eine computergestützte Simulation von oder auf Basis dieser Erscheinung durchzuführen. Dazu ist es zunächst notwendig, ein Modell des zu untersuchenden Gegenstandes anzufertigen. Die Modellierung muss dabei in besonderem Maße das Untersuchungsziel berücksichtigen.

Hat man ein Modell des Phänomens, so ist es unter Verwendung eines Simulationsframeworks möglich, eine ausführbare Simulation des Modells zu erzeugen. Das verwendete Framework soll hierbei Mittel zur Auswertung von Simulationsläufen bereitstellen, etwa die statistische Messung von veränderlichen Größen, welche im Modell verwendet werden. Auf diese Weise kann durch die Ausführung des Modells Wissen gewonnen werden, wodurch Rückschlüsse auf das originale Phänomen gezogen werden können. Dies setzt allerdings eine geeignete Modellierung voraus.

Ein einzelner Simulationslauf wird hierbei in der Regel nicht genug Informationen liefern. Es wird z. B. notwendig sein, die Simulation öfters auszuführen und dabei bestimmte Modellparameter zu variieren. So kann durch wiederholte Simulationsläufe die Kenntnis über das Phänomen vergrößert werden. Dies wird in vielen Fällen auch dazu führen, dass komplexere Änderungen an dem Modell vorgenommen werden müssen, etwa um dem Modell weitere Details hinzuzufügen oder Fehler in ihm zu beheben.

Bei der Untersuchung eines Simulationslaufes können auch bestimmte Zustände von besonderem Interesse sein. Deshalb ist hilfreich, die Simulation bei bestimmten Konfigurationen anzuhalten und den Zustand näher zu betrachten. Nützlich ist es, solche Zustände zu speichern und dem Experimentator in geeigneter Form zur Verfügung zu stellen.

Dabei kann es sein, dass die Simulation von einem solchen Zustand nun anders als im Simulationsmodell vorgesehen weitergeführt werden soll. Dies kann an Fehlern im Simulationsmodell liegen oder an äußeren Einflüssen wie Störungen, auf die das Modell hin untersucht werden soll. Das normale Vorgehen wäre hier, das Simulationsmodell entsprechend anzupassen. Dies könnte beispielsweise das Hinzufügen einer Störung zu einem Zeitpunkt sein. Anschließend würde das veränderte Modell erneut ausgeführt werden. Dieses Vorgehen ist jedoch mit hohem Aufwand verbunden. Für den Experimentator komfortabler wäre es, einen Modellzustand verändern zu dürfen und die Simulation von diesem veränderten Zustand aus weiter ausführen zu können, statt das Modell anpassen zu müssen.

Wird die Simulation begleitend zu einem realen Prozess eingesetzt, um etwa die nähere Zukunft eines realen Systems nachzuahmen, ist das Verändern von Simulationszuständen von zentraler Bedeutung. Simulationen setzen häufig stochastische Zufallsgrößen ein, um komplexe Prozesse vereinfacht nachzubilden. Nachdem das reale Phänomen abgelaufen ist, sind die Werte der Größen aber fest. Die Zustände während der Simulationsausführung müssen demzufolge ständig entsprechend angepasst werden, damit sich die Simulationsergebnisse mit der Wirklichkeit decken. Noch wichtiger, da in der Abweichung gravierender, sind Zustandsänderungen bei unvorhersehbaren Verzögerungen während der realen Prozessausführung. Diese müssen in der Simulation berücksichtigt

werden.

Ein Beispiel für eine derartige begleitende Simulation ist die des Elektrostahlwerks Gröditz, für das eine Simulation implementiert wurde, die zur Planung von Abläufen innerhalb des Stahlwerks eingesetzt wird [2]. Hierbei ist es wünschenswert, den existierenden Simulator so zu erweitern, dass dieser stets vom momentanen, realen Zustand im Stahlwerk aus gestartet werden kann. So könnten Betriebsabläufe innerhalb des Stahlwerkes leichter auf Probleme oder an neue Aufträge angepasst werden.

Diese Arbeit wird sich damit beschäftigen, wie Modellzustände gespeichert und verändert werden können. Es soll dabei möglich sein, die Simulation von einem veränderten Zustand aus weiterzuführen. Dabei wird untersucht, auf welche Art dies gelingen kann und welche Einschränkungen dafür vorgenommen werden müssen. Die Funktionalität soll für die Bibliothek ODEmx bereitgestellt werden, Änderungen an bestehender Funktionen sollen dabei vermieden werden.

## 2.1 Die Simulationsbibliothek ODEmx

Die Bibliothek ODEmx ist eine ereignis- und prozessorientierte Simulationsbibliothek für C++. Sie stellt zahlreiche Klassen bereit, die für die Entwicklung von Simulationen genutzt werden können. Eine zentrale Klasse ist hierbei `Process`. In ODEmx werden Prozesse als aktive Klassen modelliert, die den Ablauf der Simulation bestimmen. Alle Prozesse, die in einem Simulationsmodell vorkommen, müssen durch eine C++-Klasse beschrieben werden, die von der ODEmx-Klasse `Process` erbt.

Dabei definiert `Process` die abstrakte Methode `main`, welche von ererbenden Klassen implementiert werden muss. Diese Methode beschreibt den Lebenslauf des Prozesses. Es können beliebige Aktionen innerhalb dieser Methode durchgeführt werden. Verbraucht eine Aktion Modellzeit, so bietet die Klasse `Process` weitere Methoden, die genutzt werden können, um einen Modellzeitverbrauch zu simulieren. Diese Methoden sind u. a. `holdFor` und `activateAt`. Pausiert ein bestimmter Prozess wegen Modellzeitverbrauch, so werden andere Prozesse oder Ereignisse ausgeführt, bis die Modellzeit erreicht ist, zu der der Prozess weiter ausgeführt werden soll. Dann fährt er in genau dem Zustand fort, in dem er sich zuvor befunden hat. Listing 1 zeigt, wie Prozesse mit ODEmx implementiert werden.

Das Listing zeigt eine C++-Klasse `MyProcess`, die von `Process` erbt. Die Klasse `MyProcess` implementiert die `main`-Methode, in der beliebige Aktionen ausgeführt werden können. In Zeile sechs wird die Methode `holdFor` aufgerufen. Dieser Aufruf führt zu einem Verbrauch von Modellzeit.

In ODEmx wird der Wechsel des aktiven Prozesses mit Hilfe von Koroutinen umgesetzt. Die `main`-Methode jedes Prozesses ist hierbei eine Koroutine, welche unterbrochen wird, wenn der zugehörige Prozess Modellzeit verbraucht. Nach der Unterbrechung wird eine andere Koroutine und somit der damit verbundene Prozess weiter ausgeführt. Die Koroutinen werden dabei in dem gleichen Zustand wieder aktiviert, in dem sie unterbrochen worden sind. Die Auswahl der Koroutine geschieht in ODEmx anhand eines intern verwalteten Terminkalenders.

Da Koroutinen in der Sprache C++ nicht enthalten sind, müssen diese in der ODEmx

Listing 1: Implementierung eines Prozesses mit ODEmX

```
1 class MyProcess : public Process {  
2     /* Attribute ... */  
3 public:  
4     virtual void main() {  
5         /* Aktionen... */  
6         holdFor(5); // Modellzeitverbrauch und Wechsel der Koroutine  
7         /* mehr Aktionen... */  
8     }  
9 };
```

Bibliothek selbst implementiert und verwaltet werden. Bei Unterbrechung einer Koroutine lagert ODEmX dazu den aktiven Laufzeitstack ein. Wird die Koroutine wieder aktiviert, so wird dieser gespeicherte Stack wiederhergestellt. Auf diese Weise befindet sich die Koroutine bei Reaktivierung im gleichen Zustand wie vor der Unterbrechung.

Der Einsatz von Koroutinen ermöglicht es dem Benutzer, Prozessbeschreibungen auf eine einfache Art und Weise vorzunehmen. Die Serialisierung von Simulationszuständen, deren Veränderung und Fortführung wird dadurch allerdings erheblich erschwert. Die resultierenden Probleme und mögliche Lösungen werden später vorgestellt.

Weitere Details über ODEmX können z. B. in [6] und [7] gefunden werden.

## 2.2 Ein Beispiel: Simulation einer Autofähre

In diesen Abschnitt wird eine beispielhafte Simulation vorgestellt, die mit ODEmX implementiert wurde. Dabei handelt es sich um eine einfache Simulation einer Autofähre. Dazu wird eine Fähre modelliert, die zwischen dem Festland und einer Insel verkehrt und zwischen diesen beiden Seiten Autos transportiert. Mögliche Untersuchungsziele wären hierbei die durchschnittliche Wartezeit von Autos oder die Auslastung der Fähre. Abbildung 1 verdeutlicht das Problem.

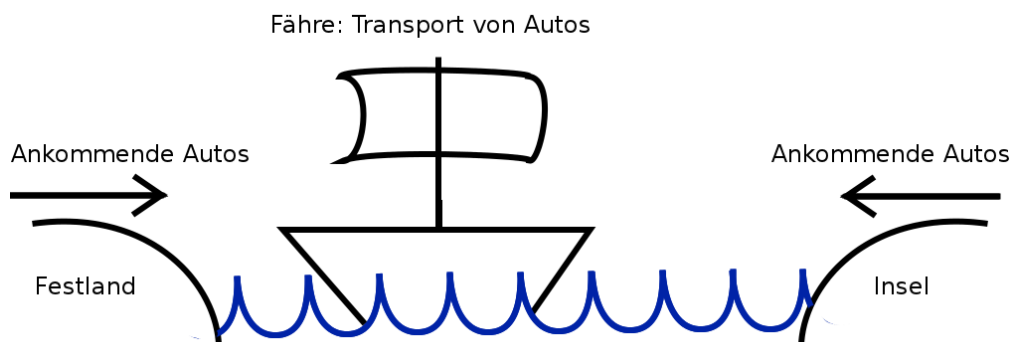


Abbildung 1: Autofähre

Die Autos erreichen die Häfen auf dem Festland und der Insel und warten darauf, von der Fähre auf das andere Ufer transportiert zu werden. Die Fähre kann dabei höchstens zehn Autos gleichzeitig laden. Die Fahrzeit zwischen beiden Ufern beträgt etwa zehn Minuten. Fährt ein Auto auf die Fähre bzw. von der Fähre, wird dafür Zeit verbraucht. Dafür benötigen verschiedene Fahrer verschieden viel Zeit. Für das Auffahren auf und Abfahren von der Fähre wird von einem Fahrer jeweils gleich viel Zeit benötigt. Erreicht die Fähre einen Hafen, so entlädt sie zunächst alle Autos und lädt anschließend die wartenden Autos. Warten keine Autos mehr oder ist die Fähre voll, so fährt sie wieder auf die andere Seite. Mindestens wartet die Fähre allerdings fünf Minuten an jedem Ufer. Insgesamt verkehrt die Fähre von 7:00 bis 21:00 Uhr. Für alle aufgeführten Zeiten werden Zufallsgrößen verwendet, um die Vorgänge vereinfacht zu beschreiben.

Für diese Zufallsgrößen können ODEMX-Klassen benutzt werden, die die benötigte Funktionalität bereitstellen. Die Insel und das Festland werden als Prozesse modelliert. Sie erzeugen in bestimmten Zeitabständen passive Auto-Objekte. Ebenfalls setzen sie nach Erstellung die Be- und Entladezeit für jedes Auto fest. Die Fähre wird ebenfalls als Prozess modelliert. Sie erhält neue Auto-Objekte von dem Insel- bzw. Festlandobjekt und entfernt diese aus dem Modell, wenn sie auf die andere Seite transportiert worden sind.

Als zentrales ODEMX-Element wird noch eine Simulationsklasse benötigt, welche die Simulation verwaltet.

## 2.3 Bestandteile eines ODEMX-Modells

Erstellt man ein Modell auf Basis von ODEMX, so ist es prinzipiell möglich, darauf aufbauend selbst einen Serialisierungssmechanismus zu implementieren. Wenn man dies für jedes einzelne Modell macht, so erlaubt dies die größtmögliche Flexibilität. Auf der anderen Seite erfordert es jedoch einen hohen zusätzlichen Aufwand beim Entwickeln von ODEMX-Modellen. Dies vergrößert die Wahrscheinlichkeit von Fehlern darin. Je nach Implementierung der Serialisierung könnten auch immer wieder Anpassungen daran erforderlich sein, wenn Veränderungen am Simulationsmodell durchgeführt werden.

Deshalb ist es kein guter Ansatz, Speicher- und Laderoutinen manuell für jedes Modell hinzuzufügen. Stattdessen ist ein generischer Ansatz wünschenswert. So soll es eine Serialisierungs-klasse geben, die alle ODEMX-Modelle verarbeiten kann. Es sollen möglichst keine oder nur wenige und einfache Anpassungen am Modell notwendig sein, damit die Serialisierung genutzt werden kann.

Um einen solchen generischen Ansatz entwickeln zu können, ist es zunächst einmal notwendig, zu untersuchen, auf welche Art und Weise Modellzustände in ODEMX beschrieben werden können. Da es sich bei ODEMX-Modellen um C++-Programme handelt, werden dem Modellierer bzw. Programmierer viele Freiheiten eingeräumt. Aus diesem Grund gibt es zahlreiche Möglichkeiten, Modellzustände darzustellen.

So kann der Modellzustand mit globalen Variablen, Objektattributen und Referenzen beschrieben werden. Des Weiteren können statische Variablen und statische Klassenattribute eingesetzt werden. Möglich ist auch die Verwendung von ganz anderen Methoden, etwa das Auslagern von Daten in Datenbanken oder Dateien, die Zustandsinformationen

beinhalten.

Neben den vom Modellierer explizit festgelegten Größen wird der Modellzustand noch durch ODEMX intern verwalteten Informationen beschrieben. Dies sind etwa die Modellzeit, der Terminkalender oder Größen zur Beschreibung von Zufallsverteilungen. Solche Informationen speichert ODEMX stets in Objektattributen. Zuletzt wird ein Zustand noch durch die eingelagerten Stacks, die zu den Koroutinen der verschiedenen Prozessobjekte gehören, beschrieben. Auf diesen Stacks befinden sich z. B. die Werte der lokalen Variablen, die in den `main`-Methoden der Prozessklassen benutzt werden.

## 2.4 Allgemeines Vorgehen und Einschränkungen

Aufgrund dieser vielen verschiedenen Arten, mit der ein Modellzustand beschrieben werden kann, ist es nicht ohne Einschränkungen möglich, den gesamten Zustand generisch zu erfassen und daran Veränderungen zuzulassen. So gibt es etwa keinen generischen Weg, sämtliche Werte von globalen oder statischen Variablen und Klassenattributen zu erhalten. Um globale Variablen und statische Klassenattribute zu serialisieren, wäre es deshalb notwendig, eine Funktion bereitzustellen, an welche all diese Variablen bzw. Attribute übergeben werden müssen, wenn das Modell gespeichert bzw. geladen wird. Für statisch Variablen, die in Funktionen genutzt werden, funktioniert dies nicht. Es gibt keine Stelle in einem C++-Programm, an der auf alle statischen Variablen zugegriffen werden kann. Mit weiteren externen Zustandsspeichern wie Datenbanken könnte jedoch ähnlich verfahren werden wie mit globalen Variablen.

Das manuelle Übergeben der globalen Variablen und Klassenattribute an eine solche Funktion erschwert die Modellierung, da für die Serialisierung modellspezifischer Quellcode geschrieben werden muss. Dieser muss bei Veränderungen am ODEMX-Modell gegebenenfalls angepasst werden. Ein besserer Weg ist es daher, auf globale Variablen und Klassenattribute zu verzichten. Ersetzt werden können diese durch Objektattribute, die in einer zentralen Modellklasse definiert werden. Bei ODEMX-Modellen bietet sich hierbei die notwendige Simulationsklasse an. Von dieser existiert bei einem Simulationslauf immer genau eine Instanz. Darüber hinaus müssen die meisten ODEMX-Klassen (insbesondere Prozesse) bereits eine Referenz auf diese Instanz haben. Es wird deshalb davon ausgegangen, dass zu serialisierende ODEMX-Modelle keine globalen oder statischen Variablen sowie Klassenattribute enthalten. Auch sollen keine zustandsrelevanten Werte extern in Datenbanken oder Dateien abgelegt werden.

Ein anderes Problem bei der generischen Serialisierung von ODEMX-Modellzuständen findet sich bei den eingelagerten Laufzeitstacks, die zu den Koroutinen der Prozessinstanzen gehören. Die Stacks könnten zwar gespeichert werden, aber es ist nicht möglich, serialisierte Stacks wiederherzustellen, damit ein bestimmtes Modell von einem Zustand aus weiter ausgeführt werden kann. Dies liegt u. a. daran, dass ODEMX-Simulationen auf verschiedenen Systemen wie Windows, Unix und Solaris lauffähig sind. Der Aufbau der Stacks unterscheidet sich zwischen verschiedenen Systemen, es kann also ein bereits bestehender Stack nicht auf ein anderes System portiert werden. Ein serialisierter Modellzustand könnte also höchstens auf dem selben System weiter ausgeführt werden.

Jedoch ist selbst dies im Allgemeinen nicht möglich. Ein Grund ist, dass bei den



genannten Systemen beim Neustart eines Programms die (virtuelle) Startadresse des Stacks jeweils unterschiedlich ist. Deshalb wären bei Wiederverwendung von eingelagerten Stacks (aus einem vorherigen Programmablauf) die darauf gespeicherten Zeiger (Framepointer und Stackpointer) nicht mehr gültig. Darüber hinaus können sich auf dem Stack wiederum Zeiger auf Objekte befinden, die sich im Heap des Speichers befinden. Wird das Programm neu gestartet, so ist es nicht möglich, den Heap genauso wiederherzustellen, dass er vorliegt wie vor einer Zustandsspeicherung. Zeiger, die auf dem gespeicherten Stack liegen, würden daher ungültig werden.

Deshalb ist es nicht möglich, die von ODEmx verwendeten Laufzeitstacks bei Neustart des Programms wiederherzustellen. Dementsprechend kann nicht der gesamte, sondern nur ein partieller Modellzustand serialisiert werden. Um einen solchen partiellen Zustand weiter ausführen zu können, muss eine wichtige Anforderung an Simulationen ausgenutzt werden: die Reproduzierbarkeit. Es ist notwendig, Simulationen so zu entwerfen, dass Ergebnisse reproduzierbar sind, d. h. zweimaliges Ausführen der gleichen Simulation (mit gleichen Parametern) muss zu den exakt gleichen Ergebnissen führen. Nur so ist es möglich, sinnvolle Auswertungen mit Hilfe von Simulationen anzufertigen und interessante Effekte von einem einzelnen Lauf analysieren zu können.

Die Reproduzierbarkeit kann für die Serialisierung der Zustände ausgenutzt werden. Man führt die Simulation bis zu einem gewissen Punkt aus und speichert den aktuellen partiellen Modellzustand. Dieser Zustand kann nun verändert werden, sei es durch das Ändern eines einzelnen Attributes oder durch anlegen komplett neuer Objekte.

Soll nun die Simulation von diesem veränderten partiellen Zustand aus weiter ausgeführt werden, so startet man die Simulation zunächst komplett neu und führt sie bis zu der Stelle aus, an dem der Zustand zuvor gespeichert worden ist. Jetzt müssen die Änderungen, die im gespeicherten Zustand vorgenommen wurden, in das neu ausgeführte Modell übernommen werden. Dadurch sind die Zustandsänderungen in einem ausführbaren Modell präsent. Die Prozessobjekte und die damit verbundene Stacks bleiben unverändert bestehen. Wird mit der Simulation des veränderten Modells nun fortgefahren, so werden die bestehenden Stacks verwendet, die weiterhin gültig sind. Da jedoch Attribute oder sogar die Objektstruktur verändert wurde, ist eine Änderung des Zustands erfolgt.

Aufgrund der notwendigen, vorgenommenen Einschränkungen darf der Modellzustand nur durch Objekte und seine Attribute beschrieben werden. Ferner wird davon ausgegangen, dass sämtliche Objekte vom benötigten Simulationsobjekt (transitiv) über Referenzen erreicht werden können. Da die eingelagerten Laufzeitstacks nicht gespeichert werden, sollen lokale Variablen, die in den `main`-Methoden der Prozesse verwendet werden, nur temporäre, nicht für den Zustand relevante Werte speichern.

In den folgenden zwei Kapiteln 3 und 4 werden zwei verschiedene, generische Ansätze vorgestellt, mit denen Zustände von ODEmx-Simulationen verändert werden können.

## 3 Direkte Serialisierung mit Boost::Serialization

Boost ist eine weit verbreitete, freie C++ Bibliothek, die eine Vielzahl von generischen Algorithmen und Strukturen bereitstellt. Darunter zählen u. a. Container, Funktoren, Methoden für Multithreading, Parser oder auch Netzwerkunterstützung. Dabei wird in vielen Fällen massiver Gebrauch von Templates gemacht. Dies ermöglicht einen flexiblen Einsatz der von Boost implementierten Funktionen. Einige Teile der Bibliothek, wie Smart Pointer, sind bereits von der Boost Bibliothek in den C++ Standard übergegangen.

### 3.1 Serialisierung mit Boost

Ein Bestandteil von Boost ist die Serialisierungsbibliothek. Mittels dieser können Objekte gespeichert und anschließend wieder geladen werden. Boost nutzt dazu ein sogenanntes Archiv. Wird ein Objekt gespeichert, so werden dazu sämtliche Attribute in dieses Archiv geschrieben. Beim Laden stellt Boost das Archiv wieder her, so dass daraus die Attribute gelesen werden können. Dazu muss jede Klasse, deren Objekte serialisiert werden sollen, die Methode `serialize` bereitstellen. In dieser werden dann alle zu serialisierenden Attribute per `&`-Operator an das Archiv zu übergeben. Das Archiv wird der Methode als Parameter übergeben.

In Listing 2 ist ein einfaches Beispiel aufgeführt. Es zeigt die Definition der Klasse `MyClass`, welche die drei Attribute von verschiedenen Typen enthält (`myInt`, `myString` und `myPointer`, Zeile 11-13). Damit Instanzen der Klasse mit Boost serialisiert werden können, enthält die Klasse die `serialize`-Methode (Zeile 5-9). In dieser werden die drei Attribute dem Archiv übergeben. Das Implementieren dieser Methode ist bereits ausreichend, um Instanzen von `MyClass` mittels Boost speichern und laden zu können.

Listing 2: Serialisierung mit Boost

---

```
1 class MyClass {
2 private:
3     friend class boost::serialization::access;
4     template<class Archive>
5     void serialize(Archive & ar, const unsigned int version) {
6         ar & myInt;
7         ar & myString;
8         ar & myPointer;
9     }
10
11     int myInt;
12     ::std::string myString;
13     MyClass* myPointer;
14 public:
15     /* und mehr... */
16 };
```

---

An das Archiv können primitive Datentypen, Zeiger, Objekte, Referenzen und Arrays übergeben werden. Darüber hinaus unterstützt Boost auch die Serialisierung von Containern aus der Standard Template Library (STL), etwa `list` und `map`. Die Klassen der Objekte, die an das Archiv übergeben werden, müssen ihrerseits auch die `serialize`-Methode besitzen. Die `serialize`-Methode wird sowohl für das Laden als auch für das Speichern verwendet. Bei Bedarf können jedoch auch zwei unterschiedliche Methoden implementiert werden.

Außer dem Archiv erhält die `serialize`-Methode einen Eingabeparameter für eine Version. Dieser zweite Parameter kann genutzt werden, um Archive, die von älteren Programmversionen erstellt worden sind, auch in neueren weiterhin verarbeiten zu können. Archive von älteren Versionen könnten einige andere Attribute enthalten als die der aktuellen Programmversion. Die Boost Archive selbst dienen dazu, die übergebenen Daten zu speichern, z. B. in eine Datei, bzw. diese von dort aus wieder auszulesen. Boost selbst beinhaltet zwei verschiedene Archivtypen, die beide Daten in eine Datei schreiben: ein Binärarchiv und ein XML-Archiv, das lesbare XML-Dateien erzeugt und wieder einliest.

Es gibt auch die Möglichkeit, eigene Archive zu implementieren und so Dateien in anderen Formaten zu erzeugen. Denkbar ist auch die Implementierung von Archiven, die keine Dateien erzeugen sondern die Daten in anderer Form ablegen. Vorstellbar sind hier u. a. Datenbanken. Es gibt auch weitere Wege, um die Serialisierung an seine Bedürfnisse anzupassen. Ein Beispiel ist die Objekterzeugung, die bei dem Laden von Archiven durchgeführt wird. Standardmäßig erwartet Boost, dass alle Klassen der zu ladende Objekte einen parameterlosen Konstruktor besitzen. Ist dies nicht der Fall, hat der Benutzer die Möglichkeit, die Methode `load_construct_data` mit passender Template-Spezialisierung zu schreiben, um dem Konstruktor der Klasse geeignete Parameter zur Verfügung zu stellen.

Speichert man mit der Bibliothek Boost eine Menge von Objekten, so erhält jedes Objekt eine eindeutige Nummer zur Identifikation. Beim Laden eines Archivs werden alle zuvor gespeicherten Objekte neu erstellt. Die Attribute werden anschließend auf die Werte gesetzt, wie sie im Archiv vorliegen. Referenzen und Zeiger können mit Hilfe der beim Speichern generierten IDs anschließend korrekt aufgelöst werden. So erhält der Benutzer nach dem Ladevorgang neue Objekte, die mit den zuvor gespeicherten identisch sind.

## 3.2 Anwendung auf das Problem

Möchte man eine ODEMX-Simulation laden, so ist es nicht möglich, neue Objekte nach dem Ladevorgang zu benutzen. Wie zuvor in Kapitel 2.4 dargestellt, können die in ODEMX für die Koroutinen verwendeten Laufzeitstacks nicht gespeichert werden. Stattdessen müssen diese während der gesamten Laufzeit der Simulation beibehalten werden - und somit auch deren zugehörige Prozessobjekte. Dies kann erreicht werden, in dem die Simulation bis zu der Zustandsänderung erneut ausgeführt wird. Anschließend sollen die Veränderungen geladen werden. Ziel ist hierbei, unter Verwendung der Boost-Bibliothek keine neuen Objekte zu erzeugen, sondern die Bestehenden weiter zu verwenden.

Die Serialisierungsbibliothek von Boost bietet, wie im vorherigen Abschnitt 3.1 be-

schrieben, einige Möglichkeiten, das Verhalten der Serialisierung anzupassen. Die Objekterzeugung während des Ladens und die anschließende notwendige Wiederherstellung von Referenzen und Zeigern sind jedoch strikt gekapselt. Schließlich ist es das erwartete Verhalten, dass beim Laden abgesehen vom Archiv keine weiteren Daten benötigt werden und abschließend dem Benutzer neue Objekte zur Verfügung gestellt werden. Um das für ODEMX gewünschte Verhalten zu realisieren, nämlich die existierenden Objekte wiederzuverwenden, ist es deshalb notwendig, Anpassungen an der Boost Bibliothek durchzuführen und sie anschließend neu zu kompilieren.

Die Serialisierungsbibliothek muss dabei so verändert werden, dass zu keinem Zeitpunkt neue Objekte konstruiert werden, sondern stattdessen stets die bestehenden Objekte manipuliert werden. Die Objekterzeugung und Referenzauflösung wird von Boost in den Klassen `pointer_iserialize` und `basic_iarchive_impl` implementiert. Diese müssen demzufolge angepasst werden. Zusätzlich muss noch die bereits beschriebene Methode `load_construct_data` implementiert werden, so dass keine neuen Objekte konstruiert werden.

Da diese Veränderungen innerhalb der Boost Bibliothek erfolgen müssen, ist es nun nicht mehr möglich, mit der veränderten Version die Serialisierung wie zuvor zu nutzen. Es werden in keinem Fall neuen Objekte erstellt, wenn Archive geladen werden. Da dieses Verhalten nicht das Erwartete noch das Dokumentierte ist, führen diese Änderungen dazu, dass die Bibliothek nun nicht mehr als Boost angesehen werden kann. Es wäre also notwendig, die benötigte Funktionalität von Boost in eine eigene neue, für die ODEMX-Serialisierung benötigte Bibliothek, auszulagern.

Möchte man nun ein ODEMX-Simulationsmodell erweitern, so dass dieses serialisiert werden kann, muss der Benutzer für alle verwendeten Klassen des Modells, wie Prozesse oder passive Klassen, die `serialize` Methode implementieren (analog zu Listing 2). Da diese Methode in der Regel nur alle Attribute der Klasse dem Archiv übergibt, wäre hierfür eine automatische Generierung vorstellbar. Realisierbar wäre es z. B. mit einer statischen Codeanalyse für C++ Klassen, die alle definierten Attribute sammelt und daraus den Code für die `serialize`-Methode generiert. Eclipse CDT bietet bereits Unterstützung für die automatische Erstellung von `get`- und `set`-Methoden für sämtliche deklarierte Attribute. Analog könnte die Codegenerierung für die `serialize` Routine funktionieren.

### 3.3 Möglichkeiten und Grenzen

Verändert man ODEMX Modelle mit Hilfe des beschriebenen Ansatzes unter Verwendung von Boost, so können sämtliche primitive Attribute, die in einem ODEMX Simulationsmodell vorkommen, abgeändert werden. Dies gilt prinzipiell auch für die primitiven Attribute, die in internen ODEMX-Klassen verwendet werden. Dazu müsste die Bibliothek jedoch entsprechend erweitert werden, so dass allen ODEMX-Klassen die zur Serialisierung benötigte `serialize`-Methode hinzugefügt wird.

Von einem auf diese Weise veränderten Modellzustand aus kann die Simulation anschließend fortgesetzt werden. Dies erlaubt bereits recht umfangreiche Eingriffe in den Simulationsablauf. Vorstellbar ist es etwa, einen bestimmten Prozess zeitweise zu deaktivieren.

tivieren. Im Autofahrenmodell könnte man beispielsweise zeitweisen einen Ausfall der Fähre untersuchen, indem man ein neues Attribut einfügt, dass angibt, ob die Fähre aktiv ist oder nicht. Weitere ähnliche Phänomene sind auf diese Art und Weise abbildbar, wie der Weg eines Bauteils durch eine Fabrik oder eine Verzögerung während eines Vorganges.

Es ist jedoch nicht möglich, darüber hinaus Änderungen an einem Zustand vorzunehmen. Dazu gehören das Erstellen von neuen und das Entfernen von bestehenden Objekten aus dem Modell. Im Beispiel kann also keine weitere Fähre hinzugefügt oder Autos von der Fähre beseitigt werden. Auch Referenzen können nicht geändert werden. Deshalb ist es beispielsweise unmöglich, das Ziel der Fähre zu ändern.

Diese Einschränkung hängt mit den notwendigen Änderungen innerhalb der Boost Bibliothek zusammen. Da bei einer Zustandsänderung die Prozessobjekte (und die mit ihnen verbundenen Stacks) weiterverwendet werden müssen, wurde die Serialisierungsbibliothek so angepasst, dass beim Laden die bestehenden alten Objekte weiterverwendet werden. Es werden keine neuen Objekte erstellt. Stattdessen wird beim Laden nur den bestehenden Referenzen gefolgt, um nacheinander alle Objekte zu erreichen und deren Attribute anzupassen. Aus diesem Grund ist es auch nicht möglich, Referenzen zu ändern: die primitiven Attribute würden dann nicht oder nicht in der erwarteten Reihenfolge beim Laden erreicht werden.

Um dennoch Referenzen auf Objekte zu ändern, könnte man diese Information durch ein primitives Attribut modellieren. So könnte im Autofahrenmodell eine Zahl angeben, welches Ziel die Fähre hat, statt einen Zeiger auf das entsprechende Objekt zu verwenden. Dieses Vorgehen würde jedoch die Verständlichkeit des Modells erheblich verschlechtern. Komplexe Modelle wären dadurch nicht mehr handhabbar. Deshalb sollte auf solche Maßnahmen verzichtet werden.

### 3.4 Serialisierung der Autofährensimulation

In diesem Abschnitt soll anhand eines Beispiels gezeigt werden, wie eine Zustandsänderung in einem ODEmx-Modell mit dem beschriebenen Verfahren durchgeführt werden kann. Dazu wird das vorgestellte Fahrenmodell verwendet. Damit die Serialisierung genutzt werden kann, muss für alle Modellklassen die `serialize`-Methode implementieren werden (analog zu Listing 2).

Damit kann die Simulation des Modells zu einer beliebigen Zeit unterbrochen und der vorliegende Zustand gespeichert werden. Nutzt man zur Speicherung ein XML-Archiv, so sieht die XML-Repräsentation eines Zustands der Autofährensimulation aus wie in Listing 3 abgebildet.

Zu erkennen sind in den Zeilen fünf bis 13 die XML-Darstellung des Festland-Prozesses. Dieser enthält u. a. ein `creationTime` Objekt, welches eine Instanz `Normal` ist. Diese ODEmx-Klasse wird genutzt, um normal verteilte Zufallszahlen zu erzeugen. Das Fahrenobjekt wird von den Zeilen 15 bis 23 beschrieben. Das Attribut `nextHarbour` beschreibt das nächste Ziel der Fähre. Es enthält dabei eine Objekt-ID des Ziels (in diesem Fall die von `mainland`). In dem gespeicherten Zustand ist die Fähre also auf dem Weg zum Festland. Darüber hinaus erkennt man die einzelnen Objekte, die die Autos

Listing 3: Das serialisierte Fahrenmodell

---

```

1 <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2 <!DOCTYPE boost_serialization>
3 <boost_serialization signature="serialization::archive" version="9">
4 <sim class_id="0" tracking_level="0" version="0">
5   <mainLand class_id="1" tracking_level="1" version="0"
6     object_id="_0">
7     <creationTime class_id="2" tracking_level="1" version="0"
8       object_id="_1">
9       <mean_>5</mean_>
10      <deviation_>0.200000000000000001</deviation_>
11    </creationTime>
12    <!-- mehr ... -->
13  </mainLand>
14  <!-- mehr ... -->
15  <ferry class_id="4" tracking_level="1" version="0" object_id="_10">
16    <nextHarbour class_id_reference="1" object_id_reference="_0">
17    </nextHarbour>
18    <car class_id_reference="3" object_id="_12">
19      <parkTime>0.031908145169092489</parkTime>
20      <creationTime>161.42868352859978</creationTime>
21    </car>
22    <!-- mehr ... -->
23  </ferry>
24 </sim>

```

---

darstellen, welche sich auf der sich Fähre befinden.

Es ist nun möglich, den Zustand durch das Editieren der XML-Datei abzuändern. Damit können zuvor nicht berücksichtigte Effekte anhand der bestehenden Simulation untersucht werden. Beispielsweise könnte diese Besonderheit ein erhöhtes Fahrzeugaufkommen am späten Vormittag sein. Zwischen zehn und zwölf Uhr soll die Anzahl der Autos, die an den beiden Häfen ankommen, höher sein als sonst. Um dieses zu untersuchen, führt man die Simulation bis zehn Uhr aus und speichert den Zustand. In dem gespeicherten Zustand ändert man nun den Erwartungswert für die Autoerzeugung (z. B. von fünf auf zwei, in Listing 3 ist dies das Attribut `mean_` in Zeile neun; das gleiche Attribut existiert für das hier nicht dargestellte Inselobjekt).

Nach der Änderung wird die Simulation erneut gestartet und dabei um zehn Uhr der veränderte Zustand geladen. Um zwölf Uhr wird ein weiteres Mal der Modellzustand gespeichert. In diesem werden nun die ursprünglichen Erwartungswerte wiederhergestellt. Abschließend wird die Simulation ein letztes mal gestartet, wobei zu den jeweiligen Zeiten beide Zustandsänderungen geladen werden. Auf diese Weise wurde ein zweistündiges, erhöhtes Autovorkommen simuliert.

In der Abbildung 3.4 werden diese Änderungen statistisch erfasst. Dabei wird gezeigt, zu welcher Uhrzeit wie viele Autos von der Fähre transportiert worden sind. Das erste

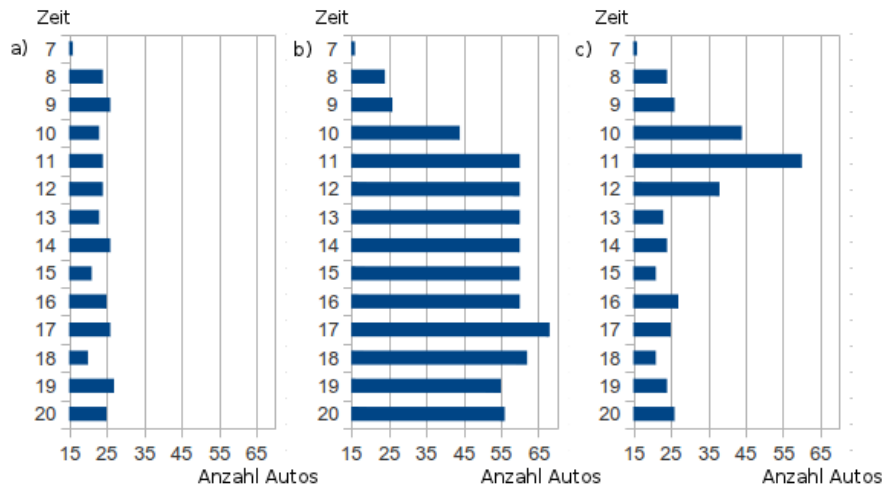


Abbildung 2: Änderungen an der Autofährensimulation

Diagramm a) zeigt dabei das Ergebnis der unveränderten Simulation. Bei dem Simulationslauf, der b) erzeugte, wurde die erste Änderung, die Verringerung des Erwartungswertes bei der Autoerzeugung, vorgenommen. Es ist ein deutlicher Anstieg der Anzahl der transportierten Fahrzeuge ab zehn Uhr zu erkennen. Das letzte Balkendiagramm c) wurde von der Simulation erzeugt, bei der beide Zustandsänderungen vorgenommen worden sind. Wie erwartet ist hier nun eine Häufung von Transporten zwischen zehn und zwölf Uhr zu erkennen.

## 4 Metamodellbasierte Serialisierung mit dem Eclipse Modeling Framework

Um strukturverändernde Modellmodifikationen vornehmen zu können, bietet es sich an, die Simulationsmodelle durch ein Metamodell zu beschreiben. Dadurch können auch neu erstellte oder gelöschte Objekte erkannt und verarbeitet werden. Ebenfalls können Referenzen auf diese Weise geändert werden. In diesem Kapitel wird ein metamodellbasierter Ansatz auf Basis des Eclipse Modeling Frameworks vorgestellt, mit dem umfangreiche Eingriffe in einen Modellzustand möglich sind.

Das Eclipse Modeling Framework (EMF) [13] ist ein Java-Framework zur automatischen Codegenerierung auf Grundlage von Metamodellen. Der Javacode, den EMF aus einem Metamodell erzeugt, kann in einem Programm eingesetzt werden, um z. B. zur Laufzeit Instanzen des Metamodells zu erzeugen. Auf solchen Modellinstanzen können dann Abfragen ausgeführt (Model Queries), Veränderungen vorgenommen oder auch Prüfungen auf Validität durchgeführt werden. Weiter stellt EMF Methoden zur Serialisierung von den Instanzen bereit. Metamodelle (und ihre Instanzen) können dabei u. a. im XML Metadata Interchange (XMI) [11] Format gespeichert werden.

Auf Basis der Metamodelle können mit EMF auch Editoren generiert werden, die sich in Eclipse Rich Client Platform (RCP) Anwendungen einbetten lassen. Die Editoren erlauben dem Benutzer dabei, Modellinstanzen in einem Baumeditor zu erzeugen bzw. zu ändern. Der für den Editor generierte Javacode kann hierbei manuell erweitert werden, um ihm weitere Funktionen hinzuzufügen oder sein Verhalten anzupassen.

Es gibt zahlreiche weitere Projekte, die weitere Tools aufbauend auf EMF bereitstellen. Bekannte Vertreter sind hier das Graphical Modeling Framework (GMF) oder Graphiti [3], mit denen graphische Editoren für EMF Metamodelle erstellt werden können. Verbreitet ist auch das Xtext Framework [5]. Mit diesem können domänenspezifische Sprachen (DSL) definiert werden um anschließend dafür entsprechende Texteditoren zu generieren. Die Menge an bestehenden und neu entstehenden Projekten, die auf EMF aufbauen, zeigt die hohe Bedeutung des Frameworks in der Java- und Modellierungswelt.

### 4.1 Das Ecore-Metametamodell

Sämtliche Funktionen, die EMF bereitstellt, basieren auf Modellen. EMF erlaubt dabei verschiedene Formen, um Metamodelle zu definieren. So kann das Metamodell durch annotierte Java Interfaces beschrieben werden, vergleichbar wie dies beim Datenbankframework Hibernate geschehen kann. Erlaubt ist auch die Definition durch UML-Modelle oder XML-Schemata (XSD) [14]. Es sind auch textuelle Beschreibungen möglich, etwa mit XCore [8] oder dem OCLinEcore [4] Editor.

Intern benutzt EMF ein einheitliches Metametamodell zur Beschreibung der verschiedenen Metamodelleigenschaften, das sogenannte Ecore-Modell. Es folgt dabei dem Essential Meta-Object Facility (EMOF) Standard der Object Management Group (OMG) [10]. Das Ecore Modell beschreibt dabei u. a. Klassen, Relationen, Attribute und die Vererbungshierarchie eines Metamodells. Dies erlaubt es es, zur Laufzeit eines Programms ein auf Ecore basierendes Metamodell zu erstellen oder zu untersuchen.



Auf Ecore basierte Metamodelle können entweder aus oben genannten Modellen durch EMF erzeugt werden oder aber auch direkt durch einen bereitgestellten Baumeditor erstellt werden.

## 4.2 emf4cpp

Das Eclipse Modeling Framework und darauf aufbauende Projekte stellen dabei nur Generatoren bereit, die Java Code erzeugen. Senac, Sevilla und Martínez [12] haben die Kernfunktionalität des Eclipse Modeling Frameworks nach C++ portiert. Die daraus entstandene Bibliothek heißt emf4cpp. Ihr Ziel war es dabei, das erprobte Metamodellierungsframework EMF auch in C++ Programmen nutzen zu können, um so eine Grundlage für die modellgetriebene Entwicklung in C++ zu schaffen. Gleichzeitig wollten sie durch die Benutzung der effizienten Sprache C++ Performancesteigerungen (sowohl in Ausführungszeit als auch Speicherverbrauch) gegenüber der Javaimplementierung erreichen.

Emf4cpp enthält dabei eine C++ Implementierung des Ecore-Metametamodells und einen Codegenerator, der C++ Quellcode aus Ecore-Modellen erzeugt. Dabei werden jedoch nur die Basiskonzepte unterstützt. Typparameter oder Annotationen, die in Ecore-Modellen eingesetzt werden können, werden bei der Codegenerierung nicht berücksichtigt. Des Weiteren enthält die Bibliothek die Möglichkeit, Modellinstanzen im XMI Format zu speichern und wieder zu laden. Weitere Funktionen von EMF sind in der Bibliothek leider nicht zu finden.

So vermisst man insbesondere die Hilfsfunktionen aus EcoreUtil im Umgang mit emf4cpp. Darin sind viele nützliche Funktionen etwa zum durchsuchen oder kopieren von Modellinstanzen enthalten. Diese oft benötigten Funktionen können bei Bedarf aber auch selbst implementiert werden. Neben den fehlenden Funktionen enthält die Bibliothek, die im Moment in der Version 0.0.2 vorliegt, einige Probleme bzw. Fehler. So gibt in der Laderoutine von Modellen einen Fehler, der verhindert, dass bestimmte Referenzen richtig geladen werden können.

In Ecore-Modellen definierte Referenzen werden durch zahlreiche Eigenschaften beschrieben, wie z. B. die Multiplizität. Eine weitere Eigenschaft heißt `containment` und ist vom booleschen Typ. Sie gibt an, ob ein referenziertes Objekt ein Teil von dem Objekt ist, dass die Referenz enthält. Hat die `containment`-Eigenschaft den Wert wahr, so ist das referenzierte Objekt ein Teil des referenzierenden Objektes, sonst nicht. Die Eigenschaft ist Vergleichbar mit der Aggregations-Eigenschaft von Assoziationen in UML. Die `containment` Eigenschaft wird auch für die Serialisierung von Modellen in XMI benötigt. Sie legt fest, an welcher Stelle ein Objekt abgelegt wird. Abgesehen von der Wurzel muss jedes zu speichernde Objekt in genau einer solchen Referenz (mit gesetzter `containment` Eigenschaft) enthalten sein. Ein Fehler in emf4cpp ist es, dass Referenzen, bei denen die `containment` Eigenschaft nicht gesetzt ist und die mehrere Objekte referenzieren können, nicht geladen werden.

Die emf4cpp Implementation nutzt diese `containment`-Eigenschaft ferner, um den Lebenszyklus der Objekte zu steuern. Wird ein Objekt aus einer `containment` Referenz entfernt, so wird automatisch dessen Destruktor gerufen. Damit kann das Objekt nicht

mehr verwendet werden. Dies ist jedoch im Umgang von Modellinstanzen während der Laufzeit äußerst unpraktisch. So ist das Verschieben eines Objektes von einem Teil des Modells in einen Anderen nicht möglich.

Für diese Arbeit wurde der aufgeführte Fehler in der Serialisierung behoben. Darüber hinaus wurde die automatische Zerstörung von Objekten aus der Bibliothek entfernt. Dies erlaubt komplexe Modelländerungen zur Programmlaufzeit. Allerdings muss der Modellierer nun darauf achten, die Destruktoren der erstellen Objekte selbst zu rufen, um Speicherlecks zu vermeiden. Empfehlenswert wäre hier der Einsatz von Smart-Pointern, um die Speicherfreigabe von Modellobjekten automatisch zu verwalten. Außerdem wurden den in der Bibliothek für Referenzen genutzte Listen-Klassen weitere Methoden zugefügt, um auch einzelne Objekte aus ihnen entfernen zu können. Die Ursprüngliche Version von emf4cpp unterstützt bei Referenzlisten lediglich das Hinzufügen, nicht aber das Entfernen von Objekten.

### 4.3 Das partielle ODEMX Metamodell

Ziel ist es nun, Simulationen für ODEMX mit Ecore-Modellen zu erstellen. Diese Modellinstanzen können dann im XMI Format gespeichert, verändert und wieder geladen werden. Anschließend soll die Simulation mit den getätigten Änderungen fortgesetzt werden. Dazu ist es zunächst nötig, ein Metamodell für die ODEMX-Bibliothek zu erstellen. Das Metamodell soll dabei die von ODEMX bereitgestellten Klassen in einem Ecore-Modell abbilden.

Abbildung 3 zeigt das Metamodell. Dieses bildet dabei nur einen Teil der von ODEMX angebotenen Funktionen ab. Neben den zentralen Klassen `Process` und `Simulation` wurden einige Klassen für die Synchronisation (`PortHead`, `PortTail` und `Timer`) und Zufallsverteilungen (`Normal` und `NegativeExponential`) in das Metamodell mit aufgenommen. Damit Simulationsläufe auch ausgewertet werden können, enthält das Metamodell noch die Statistikklassen `Sum` und `Histogram`. Die getroffene Auswahl enthält somit Klassen aus verschiedenen ODEMX-Modulen und deckt die Hauptfunktionalität ab. Es kann deshalb davon ausgegangen werden, dass eine Erweiterung des Metamodells um weitere in ODEMX existierende Klassen problemlos möglich ist.

Der aus dem ODEMX-Metamodell generierte C++ Quelltext wird so erweitert, dass die generierten Klassen jeweils ein weiteres `private` Attribut enthalten, nämlich eine Referenz auf ein Objekt der entsprechenden Klasse aus der ODEMX Bibliothek. Die Namen der beiden verschiedenen Klassen sind hierbei gleich. Um beide unterscheiden zu können, werden im Folgenden die aus dem Metamodell generierten Klassen als Meta- oder Wrapper-Klassen bezeichnet. Die Klassen der von den Meta-Klassen referenzierten ODEMX-Objekte werden ODEMX- oder funktionale Klassen genannt.

Die Konstruktoren der ODEMX-Klassen erwarten stets einige Parameter. Erstellt man ein Objekt einer Meta-Klasse, so müssen zunächst die Attribute gesetzt werden, die für die Konstruktion der zugehörigen ODEMX-Klasse benötigt werden. Anschließend wird das zugehörige ODEMX-Objekt instanziiert. Methodenaufrufe, die dann an einem Objekt des Metamodells getätigt werden, werden nun einfach an das referenzierte ODEMX-Objekt weitergeleitet. Die Meta-Klassen enthalten also keine eigene Funktionalität, son-

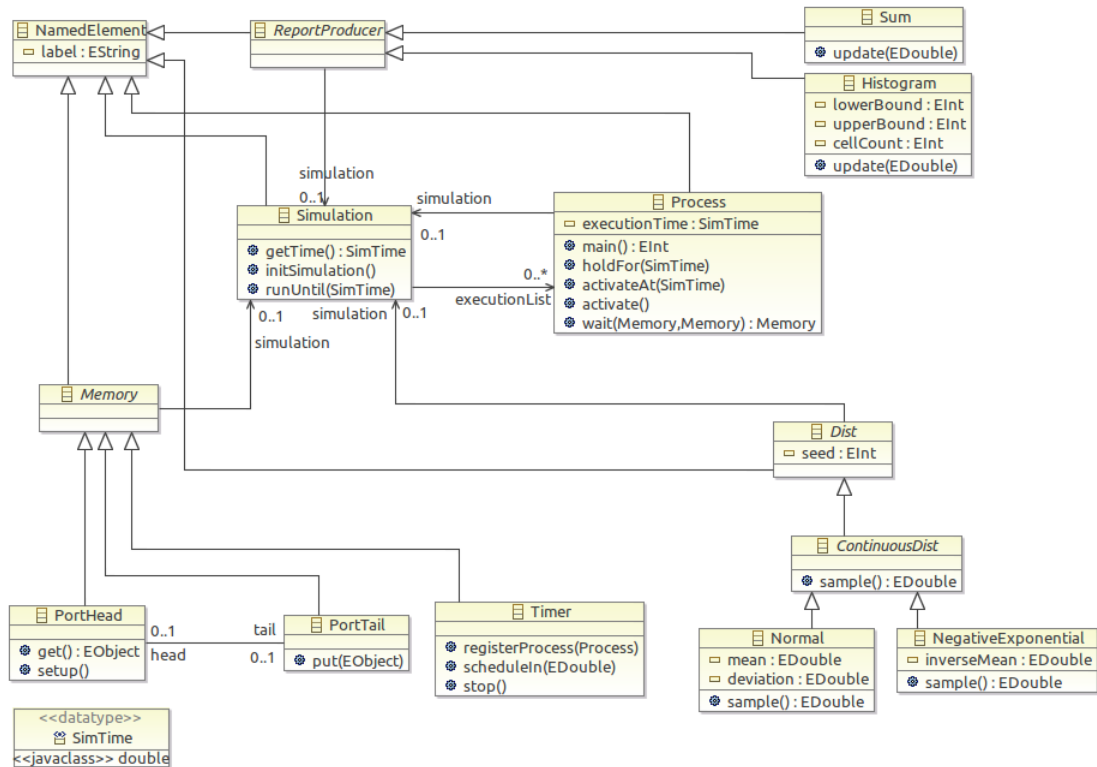


Abbildung 3: Das ODEMX Metamodell

den fundieren nur als Wrapper für die ursprünglichen ODEMX-Klassen.

#### 4.4 Ecore-Metamodelle als ODEMX Simulationen

Aktuell wird ein ODEMX-Modell mit einem C++-Programm beschrieben. Das Vorgehen, um ODEMX-Simulationen stattdessen durch Ecore-Metamodelle zu beschreiben, unterscheidet sich deshalb stark von dem bisherigen Weg. In Abbildung 4 wird die Vorgehensweise illustriert, mit der Ecore-Metamodellbasierte ODEMX-Simulationen entworfen werden können.

Der obere, gelbe Kasten zeigt die Schritte, die der Modellierer durchführen muss, um ein ausführbares Simulationsmodell zu erzeugen. Zuerst muss der Modellierer dazu ein spezifisches Ecore-Metamodell erstellen, dass sein Problem beschreibt. Dazu werden Klassen und deren Attribute sowie Referenzen definiert. Das spezifische Metamodell verwendet dabei wiederum Klassen aus dem allgemeinen ODEMX-Metamodell (Abbildung 3). So sollen etwa im spezifischen Modell definierte Prozesse von der Klasse `Process` erben, die von dem ODEMX-Metamodell bereitgestellt wird.

Als nächstes kann mit Hilfe von `emf4cpp` aus dem erstellten Ecore-Metamodell C++-Quelltext erzeugt werden. Dieser enthält die C++-Definition der modellierten Klassenstruktur. Zuletzt fehlt noch das Festlegen des Verhaltens. Dieses wird, genau wie in

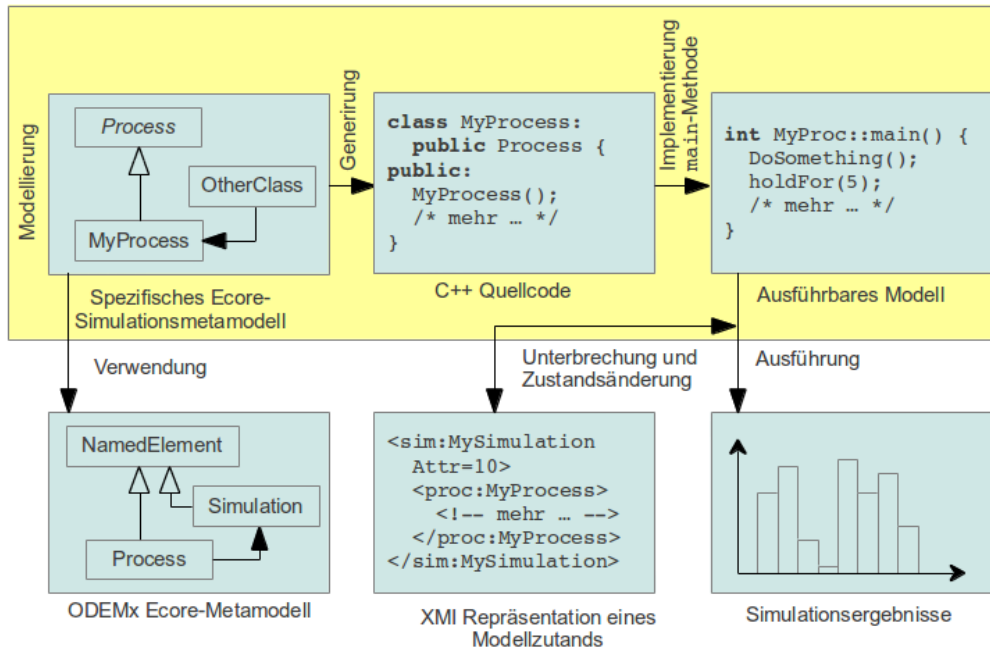


Abbildung 4: Erstellung von ODEMX Simulationen mit Ecore-Metamodellen

der ursprünglichen Variante, dass ganze Simulationsmodell in C++ zu definieren, in den main-Methoden der Prozesse beschrieben. Dazu muss der Modellierer die main-Methoden in dem generierten Code manuell implementieren.

Emf4cpp bietet hierbei, genau wie EMF, an, geschützte Regionen im Quelltext zu definieren. Dies geschieht mit Hilfe von Präprozessordirektiven, die Beginn und Ende einer solchen Region markieren. Verändert der Modellierer nun das Ecore-Metamodell, in dem er z. B. ein neues Attribut zu einer Klasse hinzufügt, bleibt der durch die Regionen geschützte Quelltext bei erneuter Codegenerierung bestehen.

Nach der Implementierung der main-Methoden ist das Modell ausführbar und die Simulation kann gestartet werden. Dazu muss lediglich eine Instanz der Simulationsklasse erstellt werden und die entsprechende Methode zur Ausführung aufgerufen werden. Reports zur Analyse von Simulationsläufen können wie bisher erstellt werden. Zusätzlich kann nun aber der Modellzustand zu einem beliebigen Zeitpunkt gespeichert werden. Dazu wurde eine neue Klasse `odemx::meta::Serializer` implementiert, die Methoden zum Speichern und zum Laden von Simulationsmodellen bereitstellt. Auf diesen Weg können die Simulationsergebnisse auf Modelländerungen hin untersucht werden. Die Arbeitsweise der Klasse `Serializer` wird im folgenden Abschnitt 4.5 vorgestellt.

## 4.5 Serialisierung von Simulationsmodellen

Definiert man ein Simulationsmodell mit Ecore, so kann das Modell zu einer beliebigen Modellzeit gespeichert werden. Dazu kann der XML-Serialisierer benutzt werden,

der in der `emf4cpp` Bibliothek enthalten ist. Die auf diese Weise erzeugte Datei kann nun verändert werden und damit auch der in ihr gespeicherte Modellzustand. Mit Hilfe von `emf4cpp` kann eine XMI-Datei auch wieder geladen werden. `Emf4cpp` erstellt beim Laden für jedes in XMI beschriebene Objekt eine neue Instanz der entsprechenden Modellklasse.

Um das geladene Modell weiter ausführen zu können, müssen aber zunächst weitere Schritte vollzogen werden. Zuerst muss die Simulation erneut bis zu der Modellzeit ausgeführt werden, zu der die Änderung vorgenommen worden ist. Dann muss den geladenen Meta-Objekten aus dem ODEMX-Metamodell, wie z. B. Instanzen von `Simulation` oder `Normal`, das zugehörige ODEMX-Objekt zugeordnet werden, dass in dem erneut ausgeführten Modell verwendet worden ist. Da dazu notwendig Zuordnung gelingt anhand der eindeutigen Namen, die jedes Objekt des ODEMX-Metamodells besitzt.

Jetzt besitzen alle Meta-Objekte eine Referenz auf ihr zugeordnetes ODEMX-Objekt. Dies ist allerdings nicht ausreichend für Prozessobjekte. Lädt man einen Simulationszustand und setzt für alle Meta-Prozesse den Zeiger auf den funktionalen ODEMX-Prozess, wird bei weiterer Ausführung der Simulation trotzdem auf das alte, nicht veränderte Meta-Objekt zugegriffen. Benutzt ein Prozess in seiner `main`-Methode eines seiner Attribute, so wird dafür durch den `this`-Zeiger des Objektes die Speicheradresse des Attributes ermittelt. Der Inhalt `this`-Zeigers selbst befindet sich dabei auf dem Stack, der zu der Koroutine des Prozesses gehört. Dieser wird bei Veränderung des Simulationsmodells jedoch nicht modifiziert. Nach dem Laden der Simulation wird also weiterhin auf die Attribute der alten Prozessobjekte zugegriffen.

Um dies zu verhindern, müssen deshalb alle Meta-Prozessobjekte des Simulationsmodells, die vor einem Ladevorgang benutzt worden sind, weiter verwendet werden. Nach dem Laden wird wie folgt vorgegangen: Für alle Prozesse werden sämtliche Attribute und Referenzen des alten Meta-Prozessobjektes auf die Werte des neuen, geladenen Prozesses gesetzt. Danach wird das gesamte geladene Modell traversiert und dabei Referenzen, die auf ein Meta-Prozessobjekt verweisen, geändert, so dass sie anschließend auf das entsprechende alte Objekt verweisen. Bei Fortsetzung der Simulation werden so weiterhin die alten Prozessobjekte benutzt, deren Attribute jedoch sämtliche Änderungen enthalten.

Wird dem Modell ein neuer Prozess hinzugefügt, so wird nach dem Laden ein neuer ODEMX-Prozess erstellt. Zusätzlich werden Attribute der ODEMX-internen Objekte so gesetzt, wie sie im geladenen Modell definiert sind. Dies sind u. a. der Terminkalender oder Attribute der Zufallsverteilungen, z. B. der Erwartungswert der Klasse `Normal`.

Der hier beschriebene Algorithmus wurde in der Klasse `odemx::meta::Serializer` implementiert. Er kann für sämtliche durch `Ecore` beschriebene ODEMX-Modelle benutzt werden. Damit Änderung an ODEMX-internen Attributen möglich sind, ist es notwendig, die bereitgestellte Serialisierungsklasse einigen ODEMX-Klassen als `friend` hinzuzufügen.

## 4.6 Möglichkeiten und Grenzen

Durch die metamodellbasierte Beschreibung des Simulationsmodells können - unter Beachtung der in Kapitel 2.4 allgemein getroffenen Einschränkung - fast alle Aspekte eines

Modellzustands verändert werden. Es können alle primitive Attribute, die in dem Simulationsmodell definiert worden sind, verändert werden. Dazu gehören auch die von ODEMX intern verwalteten Attribute. Allerdings nur die, die auch im ODEMX Ecore-Metamodell enthalten sind. Sollte der Benutzer weitere Attribute von ODEMX-Klassen verändern wollen, so muss dieses Attribut zunächst dem ODEMX-Metamodell hinzugefügt werden. Darüber hinaus muss die `Serializer`-Klasse angepasst werden, so dass dieses Attribut nach dem Laden der Modelländerung in das entsprechende ODEMX-Objekt übernommen wird.

Neben den primitiven Attributen können Referenzen verändert werden. Dadurch ist es z. B. möglich, den von ODEMX intern verwalteten Terminkalender zu editieren. Dieser enthält eine Referenz auf jeden Prozess, dessen Ausführung in Zukunft fortgesetzt werden soll. Die Modellzeit, zu der der Prozess aktiviert wird, speichert das primitive Attribut `executionTime` in den Prozessobjekten.

Zusätzlich können bei einer Modelländerung auch neue Instanzen von beliebigen Klassen des spezifischen Simulationsmodells oder des ODEMX-Metamodells erzeugt werden. Für passive Klassen ist dies ohne Weiteres möglich. Bei aktiven Klassen (den Prozessen) hingegen muss beachtet werden, dass in der aktuellen Implementierung der `Serializer`-Klasse der Lebenslauf des neu erzeugten Prozesses stets vom Anfang startet (d. h. die zu dem Prozess zugehörige `main`-Methode vom Anfang aus abgearbeitet). Durch Verwendung eines weiteren Attributes könnte der Ablauf des Lebenslaufes jedoch beeinflusst werden, etwa mit Hilfe einer `switch`-Anweisung. Durch diese kann bei Beginn der Ausführung an eine beliebige Stelle innerhalb der `main`-Methode gesprungen werden. Alternativ wäre es eine nützliche Funktion, einen bestehenden Prozess bei einer Zustandsänderung zu kopieren und damit auch seinen Zustand im Lebenslauf. Dies könnte erreicht werden, in dem der Stack des zu duplizierenden Prozesses ebenfalls kopiert wird. Dieses ist jedoch in der aktuellen Version von der `Serializer`-Klasse nicht unterstützt.

Ferner können in einem Zustand verwendete Objekte auch entfernt werden. Wird vom Benutzer ein Prozess aus dem Zustand entfernt, so wird dessen Ausführung nach Laden der Zustandsänderung abgebrochen.

Durch das beschreiben des Simulationsmodells als Ecore-Metamodell können auch leicht weitere EMF-Funktionen genutzt werden. So kann für das erstellte Ecore-Simulationsmetamodell ein Baumeditor oder mit GMF sogar ein graphischer Editor generiert werden. Mit den Editoren können anschließend die im XMI-Format gespeicherten Zustände editiert werden, was das Verändern der Zustände erheblich erleichtert. Leider sind die von `emf4cpp` erstellten XMI-Dateien nicht in allen Fällen von den von EMF bzw. GMF generierten Editoren lesbar. Die XMI Dateien, die `emf4cpp` schreibt, sind nicht in allen Fällen gültig. Es wäre also notwendig, `emf4cpp` entsprechend anzupassen.

Ein weiteres Problem, dass im Zusammenhang mit Zustandsänderungen auftritt, ist es, den richtigen Zeitpunkt zu finden, an dem eine Änderung vorgenommen werden soll. Oft ist dabei nicht ausschließlich die Modellzeit von Interesse, sondern bestimmte Modellkonfigurationen. Es wäre denkbar, dass ein Experimentator bei der Fährnsimulation den Zustand genauer untersuchen möchte, in dem mehr als eine bestimmte Anzahl von Autos an einem Ufer wartet.

Auch hier bietet EMF bereits Unterstützung: die Notification API. So ist es möglich,

jedem Objekt einen sogenannten Adapter hinzuzufügen. Durch diesen können sämtliche Änderungen, die an einem Objekt vorgenommen werden, überwacht werden. Befindet sich ein solches Objekt nun in einem interessanten Zustand, kann das gesamte Modell gespeichert werden. Emf4cpp beinhaltet bereits eine rudimentäre Implementation der Notification API.

## 4.7 Beispiel einer komplexen Zustandsänderung

Abschließend soll in diesem Abschnitt gezeigt werden, wie mit Ecore-Metamodellen eine komplexe Zustandsänderung während eines Simulationslaufes durchgeführt werden kann. Dazu soll erneut das Autofahren-Beispiel eingesetzt werden. Dazu muss das Problem zuerst als Ecore-Metamodell beschrieben werden (Vorgehen siehe Abbildung 4). Anschließend wird daraus der C++-Quellcode generiert und zuletzt das Verhalten in den main-Methoden der beiden Prozesse beschreiben. Das Ecore-Metamodell der Fähre wird in Abbildung 5 gezeigt.

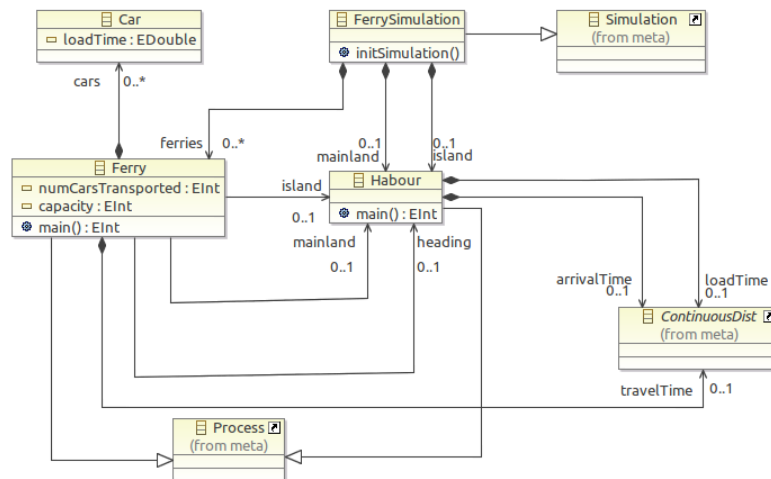


Abbildung 5: Das Ecore Autofahrenmodell

Die Abbildung zeigt, dass das Modell zwei verschiedene Prozesse definiert, nämlich die Klassen **Ferry** und **Harbour**. Beide erben von der Klasse **Process**, die im ODEMX-Metamodell definiert wurde. Darüber hinaus wird noch eine Simulationsklasse **FerrySimulation** definiert und eine passive Klasse **Car** zur Beschreibung der Autos. Daneben wird noch die Klasse **ContinuousDist** für Zufallsverteilungen eingesetzt. Das Modell verwendet noch die Synchronisationsklasse **Port** und einige Statistikklassen zur Auswertung. Der Übersichtlichkeit wegen werden diese aber nicht in der Abbildung dargestellt.

Nach den beschriebenen Schritten kann das Modell ausgeführt werden. Mit Hilfe der bereitgestellten **Serializer**-Klasse kann der gesamte Modellzustand zu einem beliebigen Zeitpunkt in einer XMI-Datei gespeichert werden. Listing 4 zeigt den zu um 14 Uhr Modellzeit gespeicherten Zustand.

In der XMI-Datei sind leicht die Objekte der verschiedenen Modellklassen mit ihren

Listing 4: XMI-Repräsentation einer Autofähren-Modellinstanz

---

```

1 <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
2 <ferry:FerrySimulation
3   xmlns:ferry="http://informatik.hu-berlin.de/sam/odemx/meta/ferry"
4   xmlns:xmi="http://www.omg.org/XMI"
5   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6   xmi:version="2.0"
7   label="Ferrysim" executionList="//island //mainland //@ferries.0">
8   <ferries numCarsTransported="156" executionTime="429.458"
9     label="ferry" mainland="//mainland" island="//island"
10    heading="//mainland" simulation="/">
11    <cars loadTime="0.480335"/>
12    <!-- und mehr... -->
13  </ferries>
14  <mainland executionTime="421.436" label="mainland" simulation="/">
15    <portHead label="mainland_port_head" tail="//mainland/portTail"
16      simulation="/">
17      <elements xsi:type="ferry:Car" loadTime="0.461754"/>
18      <!-- und mehr... -->
19    </portHead>
20  </mainland>
21  <!-- und mehr... -->
22 </ferry:FerrySimulation>

```

---

Attributen zu erkennen. In den Zeilen acht bis 13 wird etwa die Fähre beschrieben, das Festland von den Zeilen 14 bis 20. Der gespeicherte Zustand kann nun verändert werden. Als Beispiel soll um 14 Uhr eine zweite Fähre in das Modell eingefügt werden. Dazu muss eine XMI-Beschreibung einer weiteren Fähre in die Datei eingefügt werden. Diese sieht ähnlich aus wie die der bisher bestehenden Fähre. Es ist darauf zu achten, dass die neue Fähre keine Autos geladen hat und die Startzeit (Attribut `executionTime`) richtig gesetzt ist. Wichtig ist noch, dass der Name der neuen Fähre (`label`) eindeutig ist. Als letztes muss noch eine Referenz auf die neue Fähre in den Terminkalender hinzugefügt werden (Attribut `executionList1` im Objekt `ferry:FerrySimulation`). Referenzen werden im XMI-Format mit XPath-Ausdrücken beschrieben. In dem Beispiel beschreibt der Ausdruck `//@ferries.1` die Referenz auf das zweite Fahrenobjekt.

Nun kann die Simulation mit der getätigten Änderung ausgeführt werden. In der Abbildung 6 werden die Anzahl der Transportierten Autos der Fähren gezeigt. Das linke Diagramm zeigt das Simulationsergebnis ohne Zustandsänderung. Die beiden anderen Diagramme zeigen die Anzahl der transportierten Autos der beiden Fähren, die bei Laden der Zustandsänderung existieren. Erwartungsgemäß verkehrt die zweite Fähre erst um 14 Uhr. Zu erkennen ist auch, dass die erste Fähre ab 14 Uhr wesentlich weniger Autos pro Stunde transportiert als davor.



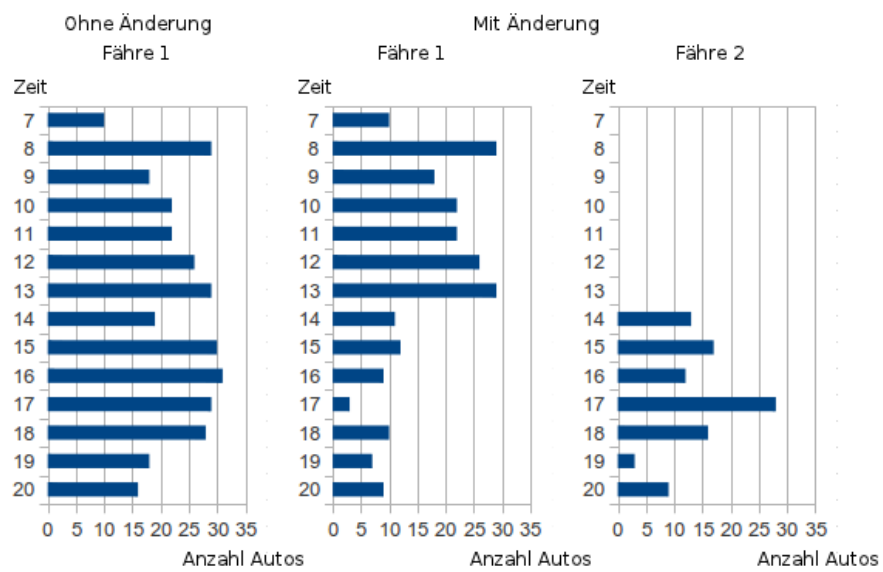


Abbildung 6: Veränderung des Autofährenmodells

## 5 Zusammenfassung

In der Arbeit wurde dargestellt, auf welche Weise einzelne Zustände von ODEMX-Simulationen gespeichert und verändert werden können, so dass von dort aus die Simulation fortgesetzt werden kann. Mit gewissen Einschränkungen ist dies auf eine generische Art und Weise möglich, so dass diese Funktion für sämtliche ODEMX Simulationen eingesetzt werden kann.

Das in Kapitel 3 vorgestellte Verfahren, Zustände direkt mit Boost zu serialisieren, erlaubt das Verändern von primitiven Attributen. Das Ändern von Referenzen oder erzeugen bzw. entfernen von Objekten ist jedoch nicht möglich. Trotzdem können auf diese Weise bereits einige interessante Zustandsänderungen untersucht werden. Um ein ODEMX-Modell auf mit dem Ansatz verändern zu können, muss den Modellklassen eine `serialize`-Methode hinzugefügt werden. In dieser müssen die Objektattribute einem Archiv übergeben werden.

Im Folgenden Kapitel 4 wurde gezeigt, wie ODEMX-Simulationen mit Ecore-Metamodellen beschrieben werden können. Das Verfahren dazu unterscheidet sich erheblich von dem bisherigem Weg, ODEMX-Modelle ausschließlich mit C++-Quelltext zu definieren. Klassenstrukturen können mit Ecore-Metamodellen übersichtlicher dargestellt werden als durch reinen C++-Code. Andererseits können bereits bestehende ODEMX-Modelle nicht ohne erheblichen Aufwand auf das Verfahren umgestellt werden. Mit Ecore-Metamodellen beschriebenen Simulationsmodellen können im Vergleich zu dem Boost-Ansatz komplexere Zustandsänderungen vorgenommen werden. Neben primitiven Objektattributen können auch Referenzen geändert werden oder Objekte hinzugefügt bzw. entfernt werden.

Um das metamodellbasierte Verfahren in der Praxis einzusetzen, wäre es zunächst notwendig, das ODEMX-Metamodell zu erweitern, damit auch weitere bestehende Funktionen genutzt werden können. Darüber hinaus sind Verbesserungen bzw. Erweiterungen an der `emf4cpp` Bibliothek nötig. Aber bereits die bereitgestellten Grundfunktionen ermöglichen es, eine Vielzahl von Simulationen durch ein EMF Modell zu beschreiben.

Es ist auch denkbar, Metamodelle auf andere Arten zu beschreiben als mit Ecore. So bieten XCppTypeRef-Bibliothek [1] oder das Qt-Framework [9] Möglichkeiten, Metamodelle für C++ zu beschreiben. Es wäre zu untersuchen, ob damit die gleichen Zustandsänderungen möglich wären wie mit dem EMF-Ansatz. Möglicherweise könnte die Serialisierung mit einer dieser Bibliotheken in bestehende Modelle leichter verfügbar gemacht werden. Auf die zahlreichen zusätzlichen EMF-Tools, wie die Generation von Editoren, müsste dann jedoch verzichtet werden.

## Literaturverzeichnis

- [1] Tharaka Devadithya u. a. “C++ Reflection for High Performance Problem Solving Environments”. In: High Performance Computing Symposium. 2007.
- [2] Ingmar Eveslage. *Reimplementation einer Stahlwerkssimulation auf der Basis der Simulationsbibliothek ODEmX*. 2006.
- [3] Eclipse Foundation. *Graphical Modeling Project (GMP)*. URL: <http://www.eclipse.org/modeling/gmp/> (besucht am 26. 11. 2011).
- [4] Eclipse Foundation. *MDT/OCLEcore*. URL: <http://wiki.eclipse.org/MDT/OCLEcore> (besucht am 27. 11. 2011).
- [5] Eclipse Foundation. *Xtext 2.1 Documentation*. 2011. URL: <http://www.eclipse.org/Xtext/documentation/2.1.0/Xtext%202.1%20Documentation.pdf> (besucht am 26. 11. 2011).
- [6] Joachim Fischer und Klaus Ahrens. *Objektorientierte Prozeßsimulation*. Addison-Wesley, 1996.
- [7] Ronald Kluth, Joachim Fischer und Klaus Ahrens. “Ereignisorientierte Computersimulation mit ODEmX”. In: *Informatik-Berichte* 218 (2007).
- [8] Ed Merks. *Xcore: Ecore Meets Xtext*. Ludwigsburg: EclipseCon Europe, 2011.
- [9] Nokia. *Introduction to Qt Quick for C++ Developers*. URL: [http://developer.qt.nokia.com/wiki/Introduction\\_to\\_Qt\\_Quick\\_for\\_Cpp\\_developers](http://developer.qt.nokia.com/wiki/Introduction_to_Qt_Quick_for_Cpp_developers) (besucht am 29. 11. 2011).
- [10] *OMG Meta Object Facility (MOF) Core Specification*. Version 2.4.1. Aug. 2011.
- [11] *OMG MOF 2 XMI Mapping Specification*. Version 2.4.1. Aug. 2011.
- [12] Andrés Senac, Diego Sevilla und Gregorio Martínez. *EMF4CPP: a C++ Ecore Implementation*. Techn. Ber. University of Murcia, 2010.
- [13] Dave Steinberg u. a. *EMF: Eclipse Modeling Framework*. 2nd Edition. Addison-Wesley Professional, 2009.
- [14] W3C. *XML Schema Part 0: Primer Second Edition*. Okt. 2004.