

Humboldt-Universität zu Berlin

Institut für Informatik

LFE Systemanalyse



Studienarbeit

Laufzeitvergleich von ODEMx und SLX

Natalia Morozova

Betreuer: Andreas Blunk

26. September 2012

Inhaltsverzeichnis

Inhaltsverzeichnis	2
1 Einführung.....	3
ODEMx	3
SLX	4
2 Laufzeitvergleich ODEMx und SLX.....	5
2.1 Zeitkonzept	7
2.2 Synchronisation	9
2.3 Zufallszahlen	11
2.4 Vergleich von rekursiven Algorithmen	12
2.4 Algorithmen ohne simulationsspezifische Funktionalität	15
3 Auswirkungen des Logging bei ODEMx	17
3.1 Implementation des Logging.....	17
3.2 Lösungsansatz zu Testzwecken.....	17
3.3 Laufzeitvergleich nach Anwendung des Lösungsvorschlages	19
3.3.1 Zufallszahlengeneratoren	19
3.3.2 Synchronisation.....	20
4 Ergebnis.....	22
5 Literaturverzeichnis.....	23

I Einführung

Diese Studienarbeit zum Thema „Laufzeitvergleich von ODEMx und SLX“ wurde am Lehrstuhl Systemanalyse, Modellierung und Computersimulation am Institut für Informatik der Humboldt-Universität zu Berlin erstellt.

Ziel dieser Arbeit ist zu prüfen, wie stark sich die Laufzeiten der Simulationsbibliothek ODEMx und des Simulationssystem SLX unterscheiden. Dazu sollen hier die Modellierungskonzepte systematisch verglichen werden. Im Anschluss wird das Logging, als eine der möglichen Ursachen für den Laufzeitunterschied, betrachtet.

ODEMx

ODEMx [Fischer96] (Object Oriented Discrete Event Modelling extended) ist eine Bibliothek für zeitdiskrete und zeitkontinuierliche Ereignis- und Prozesssimulation in C++. Sie basiert auf der Bibliothek ODEM, die Anfang 90er Jahre am Institut für Informatik der Humboldt-Universität, zur objektorientierten Simulation von Systemen entwickelt wurde. ODEMx wird sowohl in der Lehre als auch in Forschungsprojekten als Simulationswerkzeug verwendet.

Die Bibliothek zeichnet sich durch leichte Benutzbarkeit und durch automatische Beobachtung und Sammlung von Modellwerten aus. Sie ermöglicht außerdem die parallele Ausführung mehrerer unabhängiger System-Simulationen (mehrere Simulationskontexte). Dabei verwaltet jeder Simulationskontext ein eigenständiges Ensemble pseudo-paralleler Prozesse mit individueller Stack-Verwaltung, die in Abhängigkeit von jeweiligem Modellzeitverbrauch alternierend (als Koroutinen) ausgeführt werden.

SLX

SLX [Henriksen97] (Simulation Language with Extensibility) ist ein Simulationssystem zur diskreten Simulation, welches ab Mitte der 90er Jahre von der Wolverine Software Corporation entwickelt wurde. Die zum System gehörende Sprache vereint Elemente von C, Simula und GPSS, und erweitert diese um zusätzliche Simulationskonstrukte.

SLX bietet dem Entwickler Mechanismen zur Erweiterung der Kernprimitiven, sodass domänenspezifische Anforderungen integriert werden können. Zusätzlich zeichnet sich SLX durch Ausdruckstärke in Verbindung mit der Fähigkeit zu einer schnellen und klaren Fehleranalyse aus.

2 Laufzeitvergleich ODEMx und SLX

Bei der Untersuchung wurde SLX der Version 2.0 und ODEMx der Version 3.0 mit der fiber-basierten [Tannenbaum09] Implementierung der Koroutinen eingesetzt.

Um die Laufzeiten der beiden Simulationssysteme zu vergleichen, wurden verschiedene Modellierungskonzepte, mit ähnlicher Semantik herausgesucht und miteinander verglichen. Die untersuchten Konzepte gehören zu den Bereichen Simulationszeitkonzept, Synchronisierung und Zufallszahlengeneratoren. Verglichen wurden nur diejenigen Konzepte, die in beiden Systemen vorhanden sind. Die entsprechenden Funktionen wurden zu diesem Zweck in einer Schleife 1.000.000 aufgerufen (siehe Quelltext 1 und Quelltext 2). Die Zeit wurde direkt außerhalb der Schleifen gemessen, um weitere Faktoren ausschließen zu können. Es wurde bei ODEMx `clock()` (`time.h`) und bei SLX die Funktion `real_time()` zur Messung der Laufzeit verwendet. Die Tests wurden jeweils 25 Mal ausgeführt.

In den folgenden Diagrammen sind die Durchschnittswerte mit den ermittelten Standardabweichungen dargestellt. Alle Messwerte sind in Millisekunden angegeben.

Die folgende Tabelle zeigt die Hard- und Software-Konfigurationen des Testsystems, auf dem die Laufzeitmessung erfolgt ist:

Komponente	Produkt
Betriebssystem	Windows 7 Professional 64Bit
Prozessor	Intel® Core™ i5 CPU M480 @ 2.67GHz 2.67 GHz
Arbeitsspeicher	DDR3 4,00 GB
Mainboard	Hewlett-Packard 1411 KBC Version 57.31

```

#include <odemx/odemx.h>
#include <iostream>
#include <time.h>

class Prozess1 : public odemx::base::Process {
public:
    Prozess1(odemx::base::Simulation& sim):
        odemx::base::Process(sim, "prozess") { }

    int main () {
        double zzahl;
        long start_time, end_time, run_time;

        start_time = clock();

        for (int i=0; i< 1000000; i++) {
            // Funktion
        }
        end_time = clock();
        run_time = (end_time - start_time) * CLOCKS_PER_SEC/1000 ;
        std::cout<<"run_time   " <<run_time << std::endl;

        return 0;
    }
};

class MySimulation : public odemx::base::Simulation {
public:
    MySimulation (): odemx::base::Simulation() { }

    void initSimulation() {
        Prozess1* pr1 = new Prozess1(*this);
        pr1->activate();
    }
};

int main( int argc, char* argv[] )
{
    MySimulation* mySim = new MySimulation();
    mySim->run();
    return 0;
}

```

Quelltext 1 Versuchsaufbau ODEMX

```

#define SLX2 ON ;

module test {

    pointer( prozess1) pr1;

    class prozess1 (){
        actions{
            int i;

            double tstart, tend, runtime;

            tstart = real_time();

            for(i =0; i<1000000; i++){
                //Funktion
            }

            tend = real_time();
            runtime = (tend - tstart)* 1000;

            print (runtime)      " run_time      @ _ ns\n";
        }
    }

    procedure main() {
        pr1 = new prozess1;
        activate pr1;
        advance 1000000;
    }
}

```

Quelltext 2 Versuchsaufbau SLX

2.1 Zeitkonzept

Ein Grundlegendes Konzept der Modellierung ist der Modellzeitverbrauch. Sowohl bei ODEmx als auch bei SLX verläuft die Modellzeit nicht kontinuierlich sondern sprunghaft, d. h. die Modellzeit springt von einem Ereignis zum nächsten. Es wird also nicht zu jedem Zeitpunkt geprüft, ob eine Eintragung im Terminkalender vorliegt.

Die Prozesse haben die Möglichkeit sich zu einem bestimmten Zeitpunkt, in den Terminkalender einzutragen, und somit die Zeit voranschreiten zu lassen.

Im folgenden Diagramm (Abb. 1 Versuchsaufbau ODEMX) ist dargestellt wie viel Zeit eine solche Eintragung kostet. Bei ODEMX wurde dazu neben dem prozessbasierten, auch der ereignisbasierte Ansatz getestet (siehe Quelltext 3 und Quelltext 4). Bei der Untersuchung wurde auf Prozesswechsel verzichtet.

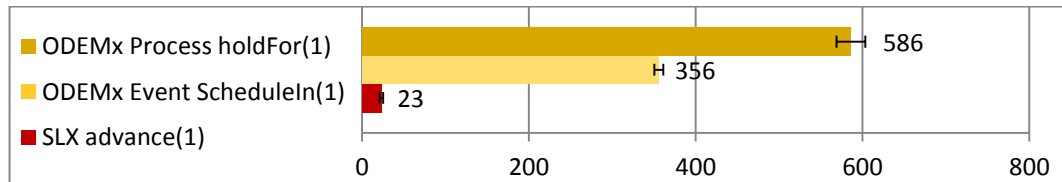


Abb. 1 Vergleich Zeitkonzept

Aus dem Diagramm ist ersichtlich, dass die Eintragung eines Prozesses in den Terminkalender in ODEMX um 25 Mal mehr Zeit verbraucht als in SLX. Die Eintragung eines Ereignisses in ODEMX ist zwar um 40% schneller als die eines Prozesses, übersteigt jedoch die Laufzeit des vergleichbaren Befehls in SLX immer noch um 13 Mal.

```
holdFor(1); // ODEMX Prozessbasiert
```

```
advance(1); // SLX
```

Quelltext 3 Funktionen

```
class Ereignis1: public odemx::base::Event{ //ODEMX Ereignisbasiert
public:
    Ereignis1 (): Event( odemx::getDefaultSimulation(), "er1" ) {}

    virtual void eventAction()      {
        scheduleIn( 1 );
    }
};
```

Quelltext 4 Ereignisbasierter Ansatz

2.2 Synchronisation

Bei den Synchronisationskonzepten verfügt ODEMX über eine weitaus größere Anzahl von implementierten Synchronisationsmöglichkeiten als SLX. So konnte z. B. zu dem „Port-Konzept“, bei welchem die Synchronisation mithilfe von übergebenen Port-Data-Objekten erfolgt, in SLX kein vergleichbares Konzept gefunden werden. Daher beschränkt sich hier der Vergleich auf die ODEMX Objekte **Bin** und **Res** und die SLX Objekte **storage** und **facility** (Abb. 2).

Bin und **Res** verwalten Ressourcen und deren geteilte Nutzung. Ressourcen sind in diesem Fall Token ohne Struktur. Ein Prozess wird blockiert, und in einer Warteschlange vermerkt, wenn die Ressource bei der Anforderung nicht verfügbar ist. Im Gegensatz zu **Bin**, beschränkt **Res** die maximal verfügbare Token-Menge, durch eine Fehlerausgabe bei deren Überschreitung (siehe Quelltext 6).

storage ist ein Speicher in dem Prozesse (Pucks) verweilen können. Dabei kann ein Prozess blockiert werden, u. A. wenn nicht genügend freie Plätze verfügbar sind. Bei der Blockierung wird der Prozess in einer Warteschlange abgelegt. Eine **facility** ist ähnlich einem **storage** mit der Kapazität 1 (siehe Quelltext 5).

Sowohl die beschriebenen ODEMX als auch SLX Elemente können für die Blockierung und Wiederfreigabe von Prozessen, abhängig von einer bestimmten Kapazität, eingesetzt werden, und bilden somit ähnliche Synchronisationskonzepte, die im folgendem Diagramm verglichen werden (Abb. 2).

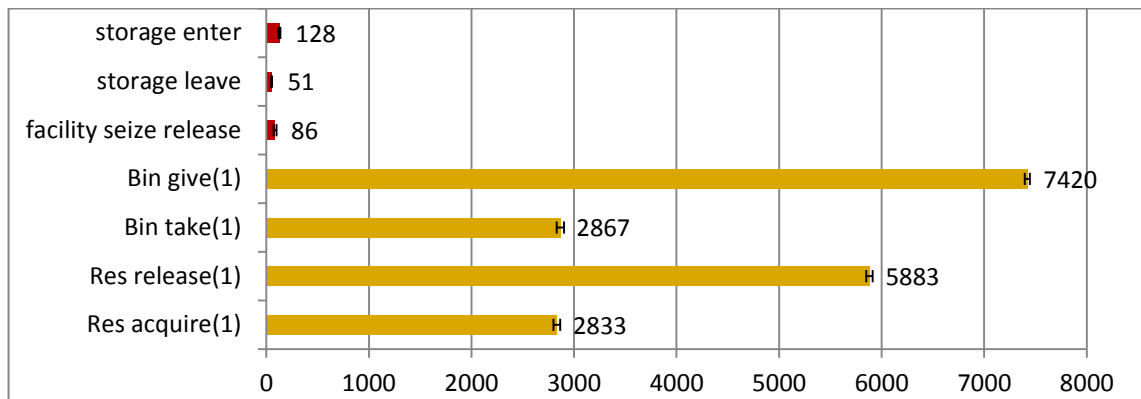


Abb. 2 Vergleich der Synchronisationskonzepte

Bei der Synchronisation ist ODEMX langsamer als SLX. So ist bei SLX der Eintritt in einen **storage** etwa 50 Mal schneller als bei ODEMX in ein **Bin**-Objekt oder **Res**-Objekt. Das Verlassen ist bei allen Objekten um etwa 50% schneller als das Eintreten.

```
storage s capacity=1000000;
//facility f;

//enter s units= 1000000; // fuer das Testen des leave

for ( j=1; j<1000000;j++){
    gate sv s;
    enter s ;
    //    leave s;

    //    seize f;
    //    release f;
}
```

Quelltext 5 SLX Synchronisation

```
binobj=new odemx::synchronization::Bin(sim, "b", 1);
//resobj=new odemx::synchronization::Res(sim, "r", 1, 1000000);

// binobj->give(1000000); // fuer das Testen des take
// resobj->release(1000000); // fuer das Testen des acquire

for (i=0; i< 1000000; i++) {
    binobj->give(1);
    //binobj->take(1);
    //resobj->release(1);
    //resobj->acquire(1);
}
```

Quelltext 6 ODEMX Synchronisation

2.3 Zufallszahlen

Die Zufallszahlengeneratoren basieren bei ODEMx und bei SLX auf dem „Lehmer-Generator“ [Lehmer49] mit der Periodenlänge $2^{31}-2$, welcher Pseudozufallszahlen im Intervall von $[0,1]$ generiert. Durch unterschiedliche Transformationsfunktionen werden aus diesen Zufallszahlenfolgen verschiedene Verteilungen realisiert.

In dem Diagramm (Abb. 3) sind die Laufzeiten für die Ermittlung des nächsten Zufallswertes mit dem Befehl `sample` dargestellt, und zwar für Verteilungsfunktionen, die sowohl in SLX als auch in ODEMx implementiert sind (Siehe Quelltext 7 und Quelltext 8).

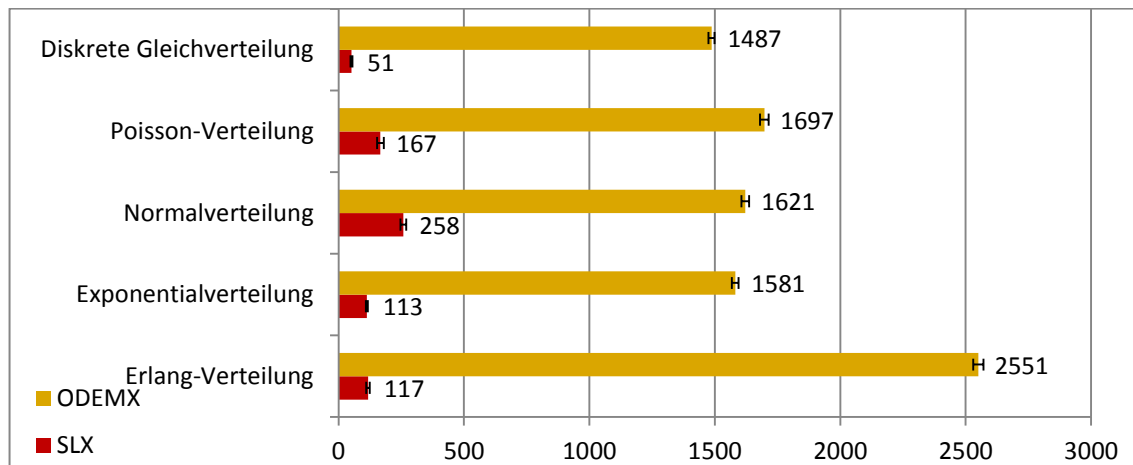


Abb. 3 Vergleich der Zufallszahlengeneratoren

Auch hier ist ein Unterschied der Laufzeiten vorhanden. So sind die untersuchten Generatoren in SLX im Durchschnitt 16 Mal schneller als in ODEMx. Am größten fällt der Unterschied bei der Diskreten Gleichverteilung aus (`RandomInt / rv_discrete_uniform`). So generiert SLX 30 Mal schneller gleichverteilte Pseudozufallswerte als ODEMx.

```
rn_stream stream;  
  
// Unterschiedliche Zufallszahlengeneratoren  
random_input x = rv_discrete_uniform(stream,1, 100);
```

```

random_input x = rv_poisson(stream, 5 );
random_input x = rv_normal(stream, 100,10 );
random_input x = rv_expo(stream, 10 );
random_input x = rv_erlang(stream,1000, 10,1);

// Funktionsaufruf
value=sample_x() ;

```

Quelltext 7 SLX Zufallszahlengeneratoren

```

//Unterschiedliche Zufallszahlengeneratoren

zzgen = new odemx::random::RandomInt(sim, "random ",1 , 100);
zzgen = new odemx::random::Poisson(sim, "random ",5);
zzgen = new odemx::random::Normal(sim, "random ",100,10);
zzgen = new odemx::random::NegativeExponential(sim, "random ",10);
zzgen = new odemx::random::Erlang(sim, "random ",1,10);

// Funktionsaufruf
value = zzgen->sample();

```

Quelltext 8 ODEmx Zufallszahlengeneratoren

2.4 Vergleich von rekursiven Algorithmen

Bei diesem Vergleich (Abb. 4), sollte geprüft werden, wie sich die Laufzeit bei einer rekursiven Funktion verhält, bei der ein Prozess auf jeder Rekursionsstufe angehalten wird. Dazu wurde eine Funktion implementiert, die sich rekursiv 1.000.000 aufruft, durch wait bzw. sleep angehalten wird, und von einem äußeren Prozess reaktiviert wird (siehe Quelltext 9 und Quelltext 10). Um einen Vergleichswert zu erhalten, wurde in einem weiteren Test der Prozess direkt in einer Schleife 1.000.000 angehalten und reaktiviert.

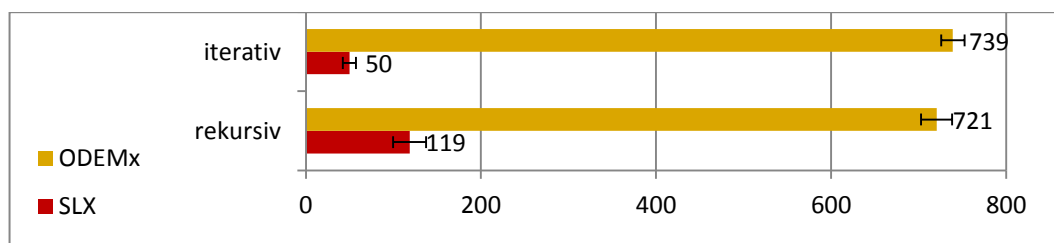


Abb. 4 Vergleich rekursiver Funktionen

ODEMx ist in beiden Fällen wieder langsamer als SLX. Es fällt jedoch auf, dass der rekursive Aufruf in ODEMx zu einer geringen Abnahme der Laufzeit führt, bei SLX hingegen, verdoppelt sich die Laufzeit im Vergleich zur iterativen Variante.

```
pointer( prozess1) pr1;
pointer( prozess2) pr2;

class prozess1 ( )
{
    pointer ( puck ) my_puck;
    actions{
        int i;
        my_puck = ACTIVE;
        for(i =0; i<1000000; i++){
            reactivate pr2->my_puck;
            wait;
        }
    }
}
class prozess2{
    pointer ( puck ) my_puck;
    actions{
        my_puck = ACTIVE;
        int i ;
        double tstart;
        tstart = real_time();
        rek_func(1000000);
        print ( (real_time() - tstart)* 1000) " @ _\n";
    }

    procedure rek_func(int x ) returning int{
        x--;
        wait;
        reactivate pr1 ->my_puck;
        if (x > 0 )
            rek_func(x);
        return x;
    }
}

procedure main(){
    pr2 = new prozess2;
    pr1 = new prozess1;
    activate pr2;
    activate pr1;
    advance 1;
}
```

Quelltext 9 SLX rekursive Funktion

```

#include <iostream>
#include <time.h>
#include "main.h"

int Prozess2::main(){
    long start_time, end_time, run_time;

    for (int i=1; i< 1000002; i++)
        pr->activate();
    return 0;
}

int Prozess::main(){
    long start_time, end_time, run_time;
    start_time = clock();
    double x =0;
    int x = this-> rek_func(1000000);
    end_time = clock();
    run_time = end_time - start_time ;
    std::cout<<"run_time   " <<run_time << std::endl;
    return 0;
}

int Prozess::rek_func(int x){
    x--;
    sleep();
    if(x>0){
        x= rek_func (x);
    }
    return x;
}

void MySimulation::initSimulation(){
    Prozess* prozess = new Prozess(*this);
    Prozess2* prozess2 = new Prozess2(*this);

    prozess->activate();
    prozess2->activate();
}

int main( int argc, char* argv[] ){
    MySimulation* mySim = new MySimulation();
    mySim->run();
    return 0;
}

```

Quelltext 10 ODEmx rekursive Funktion

2.4 Algorithmen ohne simulationsspezifische Funktionalität

Um zu prüfen, ob auch bei nicht simulationsspezifischen Berechnungen SLX schneller als ODEMx ist, wurden zwei verschiedene Algorithmen implementiert. Bei dem ersten Algorithmus handelt es sich um die Wurzelberechnung nach dem Heron-Verfahren (Siehe Quelltext 12). Es wurde die Laufzeit für die Wurzelberechnung der Zahlen 1 bis 1.000.000 gemessen.

Der zweite Algorithmus sortiert nach dem Bubblesort-Verfahren 10.000 normalverteilte Zahlen (Siehe Quelltext 11). Beide Algorithmen sind als Funktionen in einem Prozess implementiert.

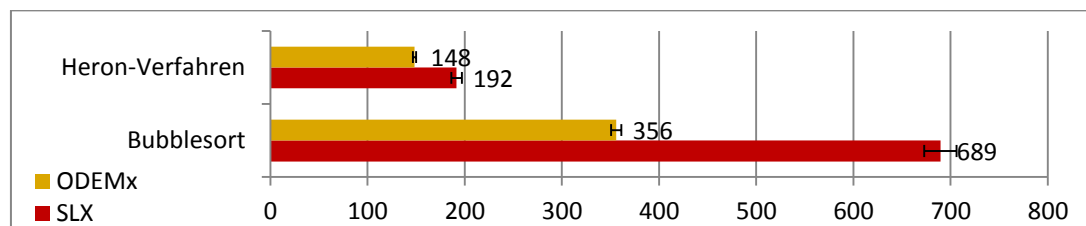


Abb. 5 Vergleich nicht simulationsspezifischer Algorithmen

Die Ausführung beider Algorithmen ist in SLX langsamer. Bei der Wurzelberechnung ist SLX um 30% langsamer. Bei dem Sortierverfahren benötigt SLX die doppelte Zeit.

```
double* bubblesort (double a[]){
    int x;
    for (int n=10000; n>1; n--){
        for (int i=0; i<n-1; i++){
            if (a[i] > a[i+1]){
                x= a[i];
                a[i]= a[i+1];
                a[i+1]=x;
            }
        }
    }
    return a;
}
```

Quelltext 11 BubbleSort

```
double heronwurzel(double a){  
    double eps = 0.0000001;  
    double b =(a+1)/2;  
    while(true){  
        b = (b+(a/b))/2;  
        if (b*b - a < eps && a- b*b < eps){  
            return b;  
        }  
    }  
}
```

Quelltext 12 Heron-Verfahren zur Wurzelberechnung

3 Auswirkungen des Logging in ODEMX

Im Folgenden soll dargestellt werden, wie sich das Logging [Kluth10] auf die Laufzeit von ODEMX auswirkt. Hierzu wurde die Implementation von ODEMX betrachtet und zum Vergleich das Logging in den von **Producer** ableitenden Klassen, durch Anpassung des Quelltextes, ausgeschaltet. Da SLX eine kommerzielle Software ist, ist die Implementation nicht einsehbar und konnte hier daher nicht weiter untersucht werden.

3.1 Implementation des Logging

Bei der Untersuchung der einzelnen Klassen ist aufgefallen, dass die meisten Objekte statistische Daten sammeln, unabhängig davon ob diese später benötigt werden oder nicht. Die meisten Objekte in ODEMX leiten von der Klasse **Producer** ab, die alle Log-Channels verwaltet. Dazu gehören die Channels **trace**, **debug**, **info**, **warning**, **error**, **fatal** und **statistics**. Diese werden durch das Macro `ODEMX_DECLARE_DATA_CHANNEL` in `ManagedChannels.cpp` erzeugt. Ist der **statistics**-Channels einmal erzeugt, hat der Benutzer keine Möglichkeit mehr diesen wieder zu deaktivieren. Die Funktion `disableLogging()` im **Producer** sollte eigentlich, durch das Löschen des in den Logging-Kanälen referenzierten `shared_ptr`, das Logging verhindern [Kluth10], dies scheint jedoch zumindest im Fall des **statistics**-Kanals nicht zu funktionieren. Somit werden auch nach dem Aufruf von `disableLogging()` weiterhin Daten gesammelt.

3.2 Lösungsansatz zu Testzwecken

Ein einfacher Lösungsansatz ist der Einbau einer zusätzlichen Membervariable, mit der dazugehörigen Set-Funktion in den **Producer** (siehe Quelltext 13 und Quelltext 14). In dieser kann festgehalten werden, ob zu einem abgeleiteten Objekt Daten gesammelt werden sollen oder nicht. Durch eine einfache if-Abfrage kann so das Sammeln von statistischen Daten für einzelne Objekte deaktiviert werden. Der Vorteil von dieser Lösung

ist, dass nicht global entschieden werden muss, ob statistische Daten gesammelt werden sollen. Dem Nutzer wird die Möglichkeit gegeben wird, nur für einzelne Objekte Daten zu sammeln. So könnte z.B. das Logging für einen Zufallszahlengenerator ausgeschaltet, für ein Synchronisationsobjekt jedoch eingeschaltet werden.

```
class Producer:    public ProducerBase
{
public:
...
    void enableLogging(); //geändert
    void disableLogging();//geändert
...
    void enableStatistics(); //hinzugefügt
    void disableStatistics();//hinzugefügt
...
    bool statistics_enabled ;//hinzugefügt ...
```

Quelltext 13 Ausschnitt Producer.h hinzugefügte bzw. veränderte Member

```
Producer::Producer( base::Simulation& sim, const Label& label ):
ProducerBase( sim, label ),    sim_( &sim ){
    statistics_enabled = true;
}
...
void Producer::enableLogging(){
    trace = sim_->getChannel( channel_id::trace );
    debug = sim_->getChannel( channel_id::debug );
    info = sim_->getChannel( channel_id::info );
    statistics = sim_->getChannel( channel_id::statistics );
    statistics_enabled = true; //hinzugefügt
}

void Producer::disableLogging(){
    statistics_enabled = false;//hinzugefügt
    trace.reset();
    debug.reset();
    info.reset();
    statistics.reset();
}
...
void Producer::enableStatistics(){ //hinzugefügt
    statistics = sim_->getChannel( channel_id::statistics );
    statistics_enabled = true;
}

void Producer::disableStatistics(){ //hinzugefügt
    statistics_enabled =false;
    statistics.reset();} ...
```

Quelltext 14 Ausschnitt Producer.cpp hinzugefügte bzw. veränderte Funktionen

3.3 Laufzeitvergleich nach Anwendung des Lösungsvorschlages

3.3.1 Zufallszahlengeneratoren

Vergleicht man nun die Laufzeiten der Zufallszahlengeneratoren (Abb. 6), so stellt man fest, dass diese um bis zu 95% gesunken sind. So sinkt z. B. bei der Berechnung einer gleichverteilten Zufallszahlenfolge von 1.000.000 Werten die Laufzeit von 1487ms auf 89ms. Somit ist ODEmx bei der Zufallszahlerzeugung, nach Abschalten des Logging, fast ebenso schnell wie SLX.

Die diskrete Gleichverteilung implementiert in C++, ohne die Ableitung von ODEmx Objekten, beträgt durchschnittlich 27ms. Somit ist der Overhead durch ODEmx weiterhin für eine längere Laufzeit verantwortlich.

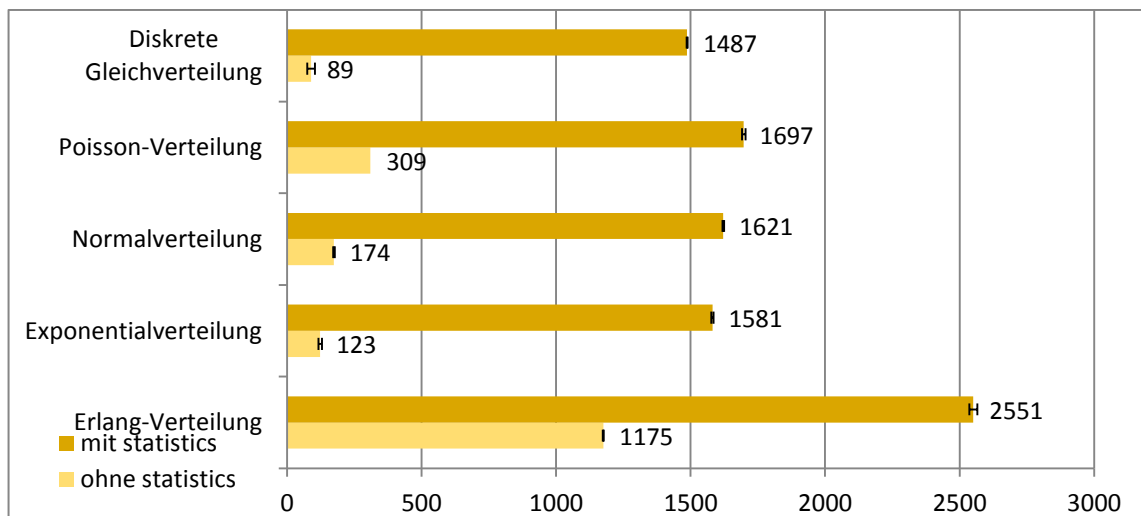


Abb. 6 Vergleich Zufallszahlengeneratoren in ODEmx mit und ohne Logging

In SLX gibt es ebenfalls die Möglichkeit bei den Zufallszahlengeneratoren die Sammlung statistischer Daten ein oder auszuschalten. In der folgenden Tabelle (Abb. 7) sind die Laufzeiten der Zufallszahlenerzeugung mit und ohne statistische Daten dargestellt.

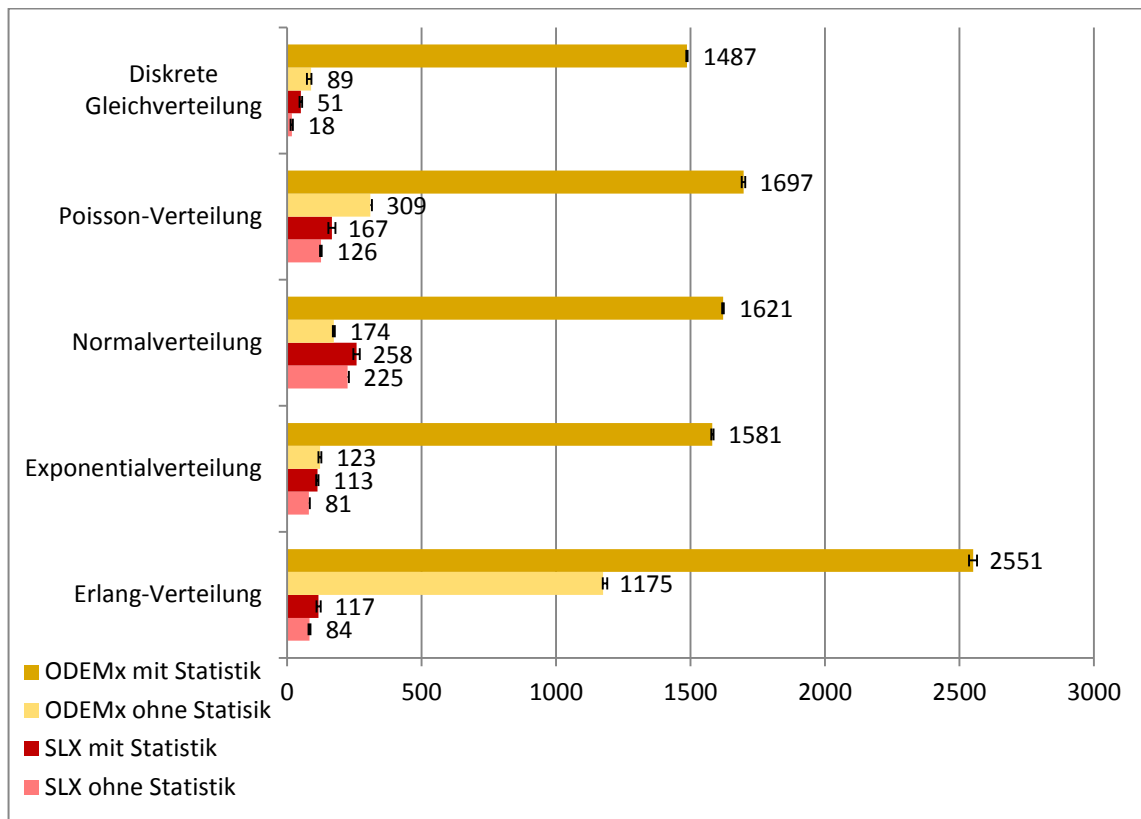


Abb. 7 Vergleich Zufallszahlengeneratoren in ODEMx und SLX mit und ohne Logging

Der Unterschied bei Ein- bzw. Ausschalten der Statistiken fällt bei SLX deutlich geringer aus als bei ODEMx. Es können maximal 30 ms, die 36 % der Laufzeit entsprechen, eingespart werden. Der Unterschied bei ODEMx beträgt bis zu 95% der Laufzeit. Ohne die Sammlung der statistischen Daten ist ODEMx bei den meisten Verteilungen fast ebenso schnell wie SLX, und übertrifft SLX sogar bei der Normalverteilung um 30%.

3.3.2 Synchronisation

Bei der Synchronisation lässt sich durch das Ausschalten der Datensammlung bis zu 98% der Laufzeit einsparen.

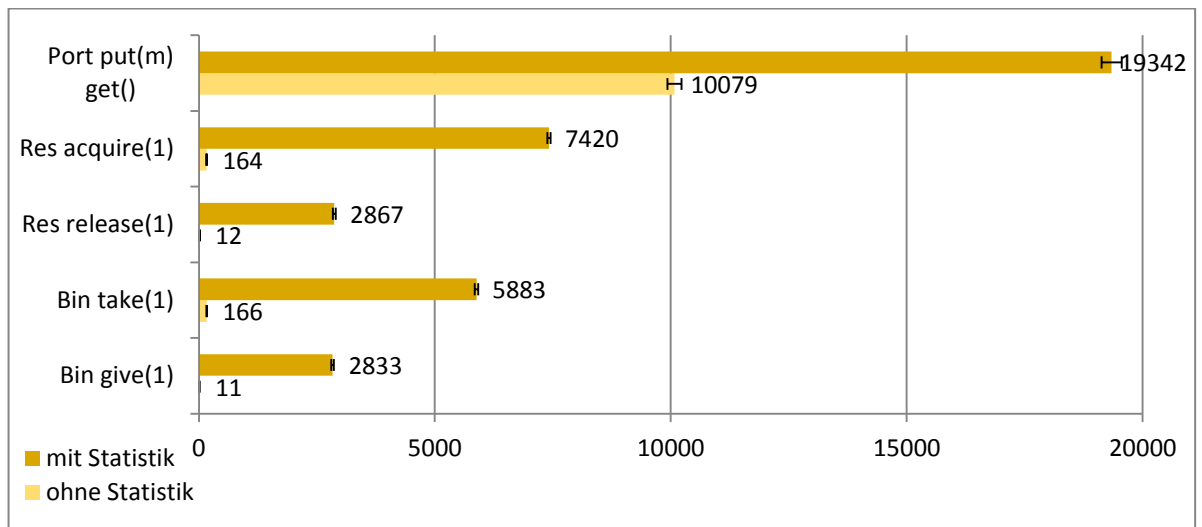


Abb. 8 Vergleich der Synchronisationsfunktionen in ODEMx mit und ohne Logging

Ohne die Sammlung der statistischen Daten ist ODEMx immer noch langsamer als SLX. Zwar ist die Funktion `acquire` des `Res`-Objektes um 22% schneller als das `storage leave` in SLX, da jedoch die Eintritt und die Austrittsfunktionen meistens zusammen verwendet werden, ist die Laufzeit in der Summe trotzdem in ODEMx länger.

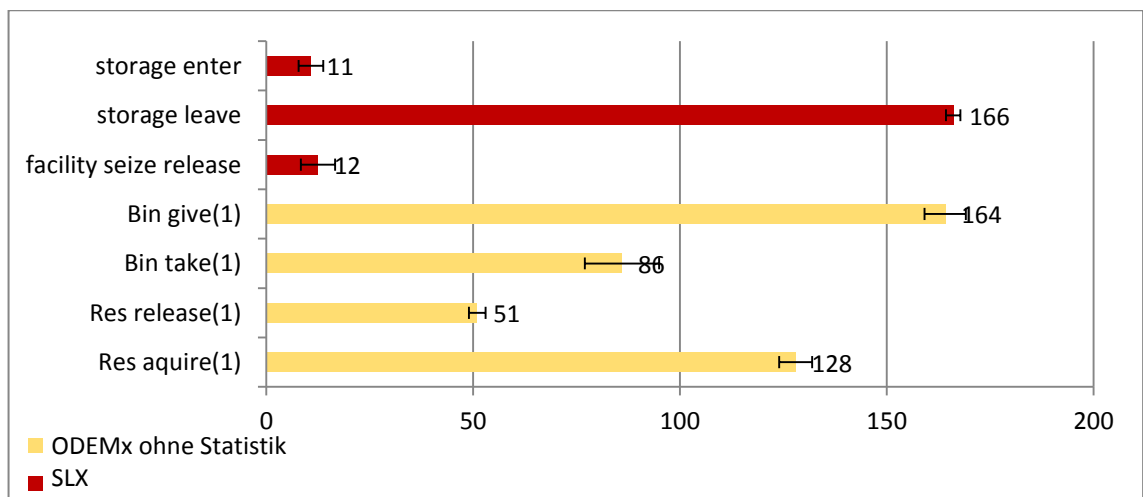


Abb. 9 Vergleich der Synchronisationsfunktionen von ODEMx ohne Logging und SLX

4 Ergebnis

Die Untersuchung hat ergeben, dass ODEmX bei allen getesteten simulationsspezifischen Modellierungskonzepten deutlich langsamer ist als SLX. Als einer der Gründe hierfür wurde das Logging identifiziert. Die untersuchten Funktionen implementieren die wichtigsten Konzepte der Simulationen. Es kann daher gefolgert werden, dass der Einsatz von ODEmX sich im Allgemeinen als weniger Laufzeiteffizient erweist. Die einzige Ausnahme bilden rechenintensive Algorithmen, die keine simulationsspezifischen Funktionen, und somit auch kein Logging, verwenden.

Die Laufzeit kann durch das Ausschalten des Logging erheblich reduziert werden. Soll die Laufzeit von ODEmX verbessert werden, so wäre eine weitere Untersuchung des Logging sinnvoll. Es wäre herauszufinden was genau bei dem Logging den größten Zeitverbrauch verursacht, und zu prüfen ob eine andere Implementation in Frage kommen könnte.

5 Literaturverzeichnis

Fischer, J., & Ahrens, K. (1996). *Objektorientierte Prozeßsimulation in C++*. Bonn: Addison-Wesley.

Henriksen, J. (1997). An Introduction to SLX. *Proceedings of the 1997 Winter Simulation Conference*. New Jersey: Institute of Electrical and Electronics Engineers.

Kluth, R. (2010). Revision der Simulationsbibliothek ODEMx. *Diplomarbeit*. Humboldt-Universität zu Berlin.

Lehmer, D. H. (1949). Mathematical methods in large-scale computing units. *Proceedings of a Second Symposium on Large-Scale Digital Calculating Machinery*. Cambridge, Massachusetts: Harvard University Press.

Tannenbaum, A. S. (2009). *Moderne Betriebssysteme* (3. Ausg.). München: Pearson Studium.