Untersuchung der Nebenläufigkeit, Latenz und Konsistenz asynchroner Interaktiver Echtzeitsysteme mittels Profiling und Model Checking

Dissertation
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften

der Fakultät für Mathematik und Informatik
der Julius–Maximilians–Universität Würzburg

vorgelegt von

Stephan Rehfeld

September 2016

Betreuer: Prof. Dr. Marc Erich Latoschik

Gutachter: Prof. Dr. Henrik Tramberend

Inhaltsverzeichnis

Danksagung

In den Jahren, in denen diese Dissertation entstanden ist, haben mich eine Vielzahl von Personen und Organisationen aktiv und passiv unterstützt. An diese geht mein Dank.

- Richard und Sophie.
- Marc Erich Latoschik und Henrik Tramberend, für die jahrelange Betreuung und Unterstützung meiner Forschungsvorhaben.
- Hartmut Schirmacher, für die moralische Unterstützung während meiner Forschung, sowie für das Feedback für die ersten Entwürfe der Doktorarbeit.
- Pedram Merrikhi, für das Korrekturlesen der meisten meiner englischsprachigen Veröffentlichungen.
- Michael Mirtschink, für das Feedback zu den Entwürfen meiner Doktorarbeit.
- Kristian Hildebrand, für das Feedback zu den Entwürfen meiner Doktorarbeit.
- Dennis Wiebusch, Martin Fischbach und Anke Giebler-Schubert, für die jahrelange Zusammenarbeit bei der Erschaffung von Simulator X.
- Christiane Hagedorn, für das Korrekturlesen einiger Kapitel der Arbeit.
- Der Beuth Hochschule für Technik Berlin, hier insbesondere Heike Ripphausen-Lipa und Sebastian von Klinski, für die finanzielle Unterstützung meiner Forschungsarbeiten, hier insbesondere die Übernahme der Kosten für die Konferenzbesuche der VRST 2014 und der IEEE VR 2016.
- Jens Pieper und Edzard Wittig für die zur Verfügung gestellten Ressourcen des CGM-Labors.
- Der Deutschen Forschungsgemeinschaft, für die Finanzierung des SIRIS-Projekts.
- Stefanie Girst vom Lektorat Berlin für die finalen Lektoratsarbeiten.

Zusammenfassung

Im Rahmen dieser Arbeit werden die Nebenläufigkeit, Konsistenz und Latenz in asynchronen Interaktiven Echtzeitsystemen durch die Techniken des Profilings und des Model Checkings untersucht. Zu Beginn wird erläutert, warum das asynchrone Modell das vielversprechendste für die Nebenläufigkeit in einem Interaktiven Echtzeitsystem ist. Hierzu wird ein Vergleich zu anderen Modellen gezogen. Darüber hinaus wird ein detaillierter Vergleich von Synchronisationstechnologien, welche die Grundlage für Konsistenz schaffen, durchgeführt. Auf der Grundlage dieser beiden Vergleiche und der Betrachtung anderer Systeme wird ein Synchronisationskonzept entwickelt.

Auf dieser Basis wird die Nebenläufigkeit, Konsistenz und Latenz mit zwei Verfahren untersucht. Die erste Technik ist das Profiling, wobei einige neue Darstellungsformen von gemessenen Daten entwickelt werden. Diese neu entwickelten Darstellungsformen werden in der Implementierung eines Profilers verwendet. Als zweite Technik wird das Model Checking analysiert, welches bisher noch nicht im Kontext von Interaktiven Echtzeitsystemen verwendet wurde. Model Checking dient dazu, die Verhaltensweise eines Interaktiven Echtzeitsystems vorherzusagen. Diese Vorhersagen werden mit den Messungen aus dem Profiler verglichen.

1 Einleitung

Computer- und Videospiele, sowie Virtual-Reality- und Augmented-Reality-Systeme benötigen als Grundlage Software-Systeme, die eine Vielzahl von unterschiedlichen Funktionen bündeln, diese einer konkreten Anwendung bereitstellen, Eingaben aus mehreren Kanälen, wie z. B. Tastatur, Maus oder Tracking-System, entgegen nehmen können, Ausgaben auf mehreren Kanälen, wie z. B. Monitor, Lautsprecher oder Haptik-Gerät, erzeugen können und sich hierbei an strenge zeitliche Anforderungen halten.

Computer- und Videospiele entwickelten sich von einem Nischenmarkt in den 1970er und 1980er Jahren zu einem Massenmarkt. Heute spielen in Deutschland 42 % der Bevölkerung ab 14 Jahren Computer- und Videospiele (Lutter et al., 2016), Produktionen einzelner Titel verfügen über ein Budget auf dem Niveau großer Kinoproduktionen (Pirzada, 2014) und der Umsatz mit Computerspielen hat sich, wie in Abbildung 1.1 zu sehen ist, in den USA im Zeitraum von 2000 bis 2015 verdreifacht.

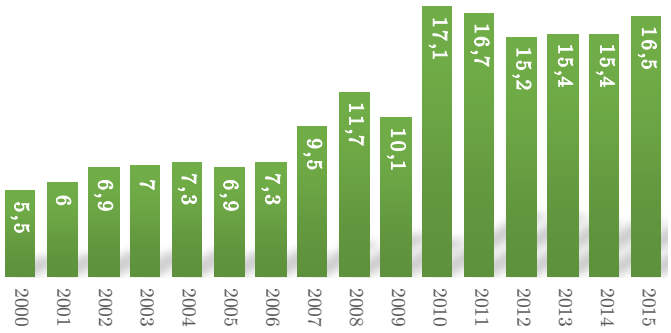


Abbildung 1.1: Umsatz mit Computer- und Videospielen in den USA von 2000 bis 2015 in Milliarden US-Dollar. (Statista, 2016)

Nach einem Hype um *Virtual Reality* (VR) in den 1990er Jahren rückte sie durch die Entwicklung leistungsstarker Head-Mounted-Displays, wie der Oculus Rift und der HTC Vive, erneut in den Fokus der Öffentlichkeit (Schnipper, 2013). Bei der VR wird, wie in Abbildung 1.2 zu sehen ist, auf technischem Wege versucht, eine für den Nutzer möglichst immersive Umgebung zu erschaffen.

Augmented Reality (AR) wurde als Begriff in den 1990er Jahren geprägt (Lee, 2012) und beschreibt die Erweiterung der Realität durch eingeblendete Zusatzinformationen. Ein großer Anwendungsbereich der AR ist die Unterhaltungsindustrie, so ist zum Beispiel der Nintendo 3DS standardmäßig mit einer einem AR-Spiel ausgestattet (Normand et al., 2012) und durch die Verbreitung der Smartphones wurde AR für die Allgemeinheit zu-

Abbildung 1.2: Ein Benutzer eines VR-Systems mit stereoskopischer Projektion und Head-Tracking.

gänglich. Abbildung 1.3 zeigt beispielhaft eine Augmented-Reality-Anwendung auf einem Smartphone.

Computer- und Videospiele, VR- und AR-Systeme lassen sich unter dem Begriff *Interaktive Echtzeitsysteme* zusammenfassen. Durch die stetig steigende Verbreitung von Interaktiven Echtzeitsystemen, stiegen seit den 1980er Jahren die Erwartungen der Benutzer in Bezug auf Funktionsumfang und die Anforderungen an die Qualität der erzeugten Ausgaben sowohl bei den VR- und AR-Systemen (Latoschik und Tramberend, 2011a) als auch bei Computerspielen (Blow, 2004).

Ein exemplarisches Beispiel für die Ausweitung der Funktionalitäten ist die Integration von Physik-Simulationen in den 2000er Jahren, welche heute fester Bestandteil von Computerspielen und Game-Engines sind. Der Unterschied wird z. B. beim Vergleich der Spiele Half-Life 1 und Half-Life 2 deutlich (Millington, 2010). Des Weiteren steigen die Erwartungen an die Qualität der generierten Audioausgaben, der Realitätsnähe der Grafik und dem Verhalten einer Künstlichen Intelligenz (KI), die Computergegner steuert. Um diesen stetig weiter wachsenden Erwartungen und Anforderungen gerecht zu werden, benötigen Interaktive Echtzeitsysteme immer mehr Rechenleistung.

Bis zur Jahrtausendwende wurde laut Sutter (2005) mehr Rechenleistung primär durch eine Erhöhung der Taktrate der CPU (Central Processing Unit) erreicht. Software-Entwickler profitierten somit automatisch von der höheren Taktrate, eine Anpassung der Software war hierfür nicht notwendig. Seit der Jahrtausendwende hat die Taktrate der CPU jedoch ihre physikalische Grenze erreicht, da der Energieverbrauch und die hierdurch produzierte Abwärme bei noch höheren Taktraten unverhältnismäßig ansteigen würde. Des Weiteren sind andere Optimierungstechniken wie Caching, Branch-Prediction sowie Out-of-Order-Execution weitestgehend in ihrer Entwicklung ausgereizt und werden nicht mehr für wesentliche Leistungssteigerung sorgen. Die einzige Lösung um die Rechenleistung weiter zu erhöhen, ist *Parallelität*. Von diese profitiert ein Software-Entwickler jedoch nicht automatisch, viel mehr muss er seine Anwendung explizit anpassen. „The free lunch is over“ konstatierte Sutter (2005).

Parallelität wird bereits für einige Bereiche in Interaktiven Echtzeitsystemen angewendet. So ist seit Mitte der 1990er Jahre Hardware zur Beschleunigung des Renderings für Endanwender erhältlich. Aus dieser Hardware entwickelten sich die heutigen GPUs

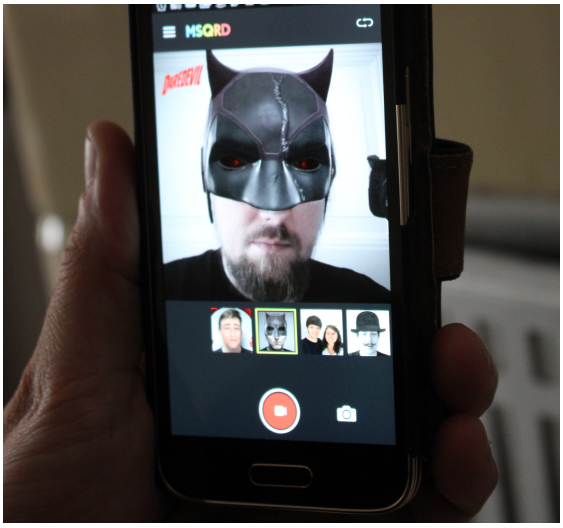


Abbildung 1.3: Die Augmented-Reality-Anwendung *MSQRD*, in der ein Benutzer Masken und Effekte zum eigenen Gesicht in Echtzeit und in Videoaufnahmen hinzufügen kann.

(Graphics Processing Unit), welche zunehmend weitere *datenparallele* Probleme, wie beispielsweise die Physik-Simulation, übernehmen.

Den datenparallelen Problemen stehen die *taskparallele Probleme* gegenüber. Bei diesen werden unterschiedliche Instruktionen auf die gleichen oder unterschiedliche Daten angewendet. Hierzu müssen Bestandteile identifiziert werden, die *nebenläufig* sind. Nebenläufigkeit bedeutet, dass zwei Berechnungen voneinander unabhängig sind, diese somit in einer beliebigen Reihenfolge oder gleichzeitig, also parallel, ausgeführt werden können. Nebenläufigkeit ist somit eine notwendige Vorbedingung für Parallelität, nebenläufige Berechnungen müssen jedoch nicht zwingend parallel zueinander ausgeführt werden.

Taskparallele Probleme lassen sich nicht auf der für datenparallele Probleme optimierten GPUs lösen, sondern benötigen zur parallelen Bearbeitung mehrere vollwertige Prozessoren. Aufgrund der vorher beschriebenen physikalischen Grenzen sind Mehrkernprozessoren heute die dominante Architektur von Computern und werden dies auch auf absehbare Zeit bleiben (Olukotun und Hammond, 2005; Creeger, 2005; Borkar und Chien, 2011).

Hieraus folgt, dass bei der Bereitstellung der notwendigen Rechenleistung, die benötigt wird, um den steigenden Erwartungen und Anforderungen der Endanwender gerecht zu werden, die optimale Nutzung von von Mehrkernprozessoren eine entscheidende Rolle spielt.

Hierzu existieren jedoch bisher kaum zufriedenstellende Lösungen oder, wenn Lösungsansätze existieren, wurden diese bisher nicht in einem ausreichenden Maß evaluiert. Kommerzielle Game-Engines, wie z. B. Unity 3D, verbieten es dem Anwendungsentwickler explizit, Funktionen der Engine außerhalb eines definierten Haupt-Threads aufzurufen. Eine Parallelisierung und somit optimale Nutzung des Systems ist so von vornherein erschwert.

Allgemein konzentrierten sich die Entwickler von Computerspielen und Game-Engines in den letzten Jahrzehnten eher darauf eine Game-Engine möglichst gut optimiert in einer sequentiellen Umgebung laufen zu lassen. Aus der Forschung zu VR-Systemen kommen vielversprechende Ansätze für die Parallelisierung, wobei hier der Fokus in den letzten Jahrzehnten eher auf Clustering als auf Mehrkernprozessoren lag.

Ein Grund für diesen unbefriedigenden Zustand ist, dass die Entwicklung einer Software durch Parallelität erheblich verkompliziert wird. Während bei einer sequentiell arbeitenden Software die Abfolge der Instruktionen klar und eindeutig ist, gibt es bei einer parallel arbeitenden Software eine sehr große Anzahl von möglichen Ausführungsszenarien. Dies führt dazu, dass sogar ausgewiesene Experten für parallele Programmierung, selbst bei einfachsten nebenläufigen Algorithmen mit wenigen Instruktionen, überraschende, ihnen unbekannte Ausführungsszenarien finden (Ben-Ari, 2010) oder dass bei komplexen Systemen, trotz sorgfältiger und kontinuierlicher Kontrolle, Fehler enthalten sind (Lee, 2006).

1.1 Forschungsfragen

Die im vorherigen Abschnitt erläuterten Probleme führen zu fehlerhafter Software und den typischen Problemen, wie z. B. Deadlocks. Hier eine Lösung zu finden erfordert ein generelles Umdenken, vergleichbar mit der Einführung der objektorientierten Programmierung (Sutter, 2005; Sutter und Larus, 2005), wenn es nicht sogar die größte Herausforderung darstellt, vor der die Informatik je stand (Borkar und Chien, 2011), denn die klassische direkte Benutzung von Threads und Locks ist in der Praxis nicht beherrschbar (Lee, 2006). Hieraus ergibt sich die erste Forschungsfrage dieser Arbeit:

Forschungsfrage 1. *Welche Modelle existieren, auf deren Grundlage ein nebenläufiges Interaktives Echtzeitsystem implementiert werden kann, wobei das Modell die typischen Probleme der parallelen Programmierung vermeidet?*

Die Entscheidung wird auf das *asynchrone Modell*, welches später im Laufe dieser Arbeit detailliert erläutert werden soll, fallen. Im asynchronen Modell wird ein Interaktives Echtzeitsystem in einzelne nebenläufige Teile zerlegt, wobei jedes dieser Teile mit einer eigenen Frequenz laufen kann. Diese Entkopplung eröffnet ein hohes Maß an Freiheit, die es erlaubt, ein konkretes System oder eine konkrete Anwendung hin zu einem gewünschten Verhalten zu optimieren. Hieraus leitet sich die zweite Forschungsfrage dieser Arbeit her:

Forschungsfrage 2. *Welche sinnvollen Nebenläufigkeits- und Synchronisationsschemata existieren für ein asynchrones Interaktives Echtzeitsystem und wie können diese verwendet werden, um eine Optimierung hin zu zu einem bestimmten Verhalten zu gewährleisten?*

Durch die Wahl des asynchronen Modells entsteht darüber hinaus ein neues Problem: Die Herstellung und Wahrung der *Konsistenz*. So müssen die einzelnen asynchronen Teile eines Interaktiven Echtzeitsystems dennoch konsistente Daten über die simulierte Szene

haben, da ansonsten sinnvolle Berechnungen kaum möglich sind. Zum anderen müssen auch die ausgegebenen Daten konsistent sein, damit beim Benutzer keine fehlerhaften Wahrnehmungen entstehen.

Ein weiterer wichtiger Punkt sind die strengen zeitlichen Anforderungen innerhalb der ein Interaktives Echtzeitsystem auf Eingaben des Benutzer reagieren muss. Die Zeit zwischen der Eingabe eines Benutzer bis zur Reaktion des Systems hierauf in Form einer Ausgabe wird als *Latenz* bezeichnet, wobei diese möglichst gering sein sollte. Ein Beispiel hierfür ist die Motion-To-Photon-Latency, die bei einem Head-Mounted-Displays die Zeit zwischen der Kopfbewegung des Benutzers und der Aktualisierung des dargebotenen Bildes mit der neuen Kopfposition beschreibt. Eine zu hohe Latenz erschwert die Fähigkeit eines Benutzers, schnell auf Veränderungen zu reagieren und macht somit die Ausführungen von Aufgaben schwieriger oder gar unmöglich. Dies ist für einen Benutzer mindestens ärgerlich, wenn er z. B. in einem Online-Spiel hierdurch einen Nachteil gegenüber anderen Spielern hat. Bei VR-Systemen kann dies aber auch zu Simulator Sickness führen, was das System für einen Benutzer unbrauchbar werden lässt. Latenz kann bei einem Interaktiven Echtzeitsystem viele Ursachen haben. Sie kann z. B. durch verwendete Ein- und Ausgabegeräte entstehen oder durch die Verarbeitung und Weitergabe von Daten innerhalb des Interaktiven Echtzeitsystems.

Hieraus leiten sich folglich drei Kriterien ab, nach denen das Modell, bzw. eine Implementierung dieses Modells, untersucht werden muss:

- Nebenläufigkeit
- Konsistenz
- Latenz

Diese Kriterien müssen in Form von numerischen Metriken abgebildet werden, um eine reproduzierbare Aussage aus Untersuchungen des Modells oder eine Implementierung zu erhalten.

Für die konkrete Untersuchung bietet es sich an, zunächst eine Implementierung zu erstellen und diese im Anschluss durch *Profiling und Benchmarking* auf ihr Verhalten zu überprüfen. Tatsächlich ist dies das gängige Verfahren und kommerzielle Game-Engines werden in der Regel mit einem zugehörigen Profiler ausgeliefert. Diese Profiler sind stark an klassische Profiler für die allgemeine Software-Entwicklung angelehnt. Gerade die spezifischen Anforderungen und Eigenheiten Interaktiver Echtzeitsysteme eröffnen hier jedoch bisher ungenutztes Potenzial, die im Profiling verwendeten Darstellungen von Daten zu überdenken. Hieraus ergibt sich die dritte Forschungsfrage:

Forschungsfrage 3. *Wie können Profiling-Daten im Kontext eines asynchronen Interaktiven Echtzeitsystem für einen Entwickler aufbereitet werden, wobei man den spezifischen Eigenschaften eines Interaktiven Echtzeitsystems gerecht wird und welche insbesondere die Untersuchung der Nebenläufigkeit, Latenz und Konsistenz erlaubt?*

Ein wesentlicher Nachteil des Profilings ist, dass es naturgemäß nur auf eine konkrete Implementierung angewendet werden kann. Somit wird nicht nur das reine Modell

geprüft, sondern immer auch Implementierungsdetails der konkreten Implementierung. Um eine klarere Prüfung des reinen Modells zu ermöglichen, bietet sich die Technik des *Model Checkings* an. Hierbei wird zu einem Modell eine formale Spezifikation erstellt, die anschließend durch eine Software geprüft wird. Üblicherweise wird durch das Model Checking geprüft, ob eine Software die dieses Modell implementiert, aufgrund des Modells in einen Deadlock geraten kann, ob ein definierter Zustand in jedem Fall eintritt oder ob ein ungewünschter Zustand niemals eintreten kann. Das Model Checking sollte sich jedoch auch dazu eignen, das Modell eines Interaktiven Echtzeitsystems auf die Nebenläufigkeit, Konsistenz und Latenz zu prüfen. Model Checking wurde zuvor nicht im Kontext von Interaktiven Echtzeitsystemen eingesetzt. Hieraus ergibt sich die vierte und letzte Forschungsfrage dieser Arbeit:

Forschungsfrage 4. *Kann Model Checking verwendet werden, um das Modell eines Interaktiven Echtzeitsystems auf die zu erwartende Nebenläufigkeit, Konsistenz und Latenz zu prüfen und liegen die vorhergesagten Werte nahe genug an den gemessenen Werten des Profilings, damit die erzielten Resultate in der Praxis brauchbar sind?*

1.2 Struktur und Beiträge

Im Folgenden wird die Struktur dieser Arbeit erläutert. Anschließend wird aufgeführt, welche Beiträge zu den vier formulierten Forschungsfragen geleistet werden. In Kapitel 2 wird der Stand der Wissenschaft und Technik in Bezug auf Interaktive Echtzeitsysteme, Profiling und Model Checking aufgearbeitet. In Kapitel 3 werden Nebenläufigkeits- und Synchronisationsschemata für ein asynchrones Interaktives Echtzeitsystem definiert. Aus diesen Nebenläufigkeits- und Synchronisationsschemata werden darüber hinaus fünf Grundelemente abgeleitet, die zur Umsetzung der Schemata benötigt werden. Um über die Nebenläufigkeit, Konsistenz und Latenz reproduzierbare Aussagen treffen zu können werden für diese in Kapitel 4 auf formale Weise Metriken definiert. An diesen Metriken orientiert sich ebenfalls die weitere Arbeit. So wird in Kapitel 5 spezifiziert, wie aus Profiling-Daten aus einem asynchronen System diese Metriken berechnet werden können. Dies wird ebenfalls im Kontext des Model Checkings in Kapitel 6 gemacht. Die Beschreibung der hierfür erarbeiteten Algorithmen erfolgt implementierungsunabhängig mit der formalen Sprache PlusCal. Eine Beschreibung der Implementierung der Konzepte erfolgt in Kapitel 7. Hierbei wird zunächst das verwendete Simulator X Framework (Latoschik und Tramberend, 2011b) beschrieben. Simulator X ist ein asynchrones, auf dem Actor Model (Hewitt et al., 1973) basierendes Interaktives Echtzeitsystem, welches im Kontext der VR entwickelt wurde. Anschließend wird die Umsetzung der vorher erarbeiteten Konzepte beschrieben. Zunächst wird eine Domain Specific Language vorgestellt, mit deren Hilfe Simulator X konfiguriert werden kann, um sich entsprechend der vorgestellten Nebenläufigkeits- und Synchronisationsschemata zu verhalten. Darüber hinaus wird die Implementierung eines Profilers und die Anbindung eines Model Checkers beschrieben, welche die vorher spezifizierten Algorithmen zur Berechnung der Metriken implementieren. In Kapitel 8 wird die Domain Specific Language bei einer Demo-Anwendung verwendet, um auf diese die Nebenläufigkeits- und Synchronisationsschemata anzuwen-

den. Darüber hinaus wird sowohl der Model Checker und der Profiler verwendet, um die Nebenläufigkeit, Konsistenz und Latenz zu schätzen, bzw. zu messen. Die Arbeit schließt in Kapitel 9 mit einer Zusammenfassung der gefundenen Antworten auf die Forschungsfragen und einem Ausblick ab.

In Tabelle 1.1 werden die Beiträge dieser Arbeit zusammengefasst, der Bezug zu den Forschungsfragen gezeigt und angegeben, wo Informationen zu diesen Beiträgen zu finden sind.

Tabelle 1.1: Aufstellung der in dieser Arbeit geleisteten Beiträge zu den vier formulierten Forschungsfragen.

Nr.	Beitrag	zu Forschungsfrage	Kapitel
1	Es werden verschiedene Modelle für die Nebenläufigkeit in Interaktiven Echtzeitsystemen anhand definierter Kriterien untersucht, wobei die Vorzüge eines asynchronen Modells aufgezeigt werden.	1	2.1.4
2	Es werden verschiedene Technologien für die Synchronisation in nebenläufigen Systemen anhand von definierten Kriterien untersucht.	1	2.2
3	Es werden eine Reihe von Nebenläufigkeits- und Synchronisationsschemata herausgearbeitet, auf deren Grundlage eine Menge von Grundelementen definiert wird, die zu ihrer Umsetzung notwendig sind. Als Implementierung wird eine Domain Specific Language vorgestellt, mit der sich das Verhalten eines asynchronen Interaktiven Echtzeitsystems konfigurieren lässt.	2	3 und 7.2.1
4	Für die Nebenläufigkeit, Latenz und Konsistenz werden auf formale Weise numerische Metriken definiert, um diese Eigenschaften reproduzierbar abbilden zu können. Für diese Metriken wird implementierungsunabhängig und formal beschrieben, wie sowohl aus Daten, die durch Profiling erzeugt wurden, als auch aus Daten, wie sie durch Model Checking erzeugt wurden, die vorher definierten Metriken berechnet werden können.	3 und 4	4, 5 und 6
5	Es wird die Implementierung eines Profilers vorgestellt, der sich in der Darstellung von spezifischen Eigenschaften eines Interaktiven Echtzeitsystems an diesen und nicht an Profilern für die allgemeine Software-Entwicklung orientiert.	3	5 und 7.2.2
6	Es wird die Anbindung eines Model Checkers vorgenommen, mit dem sich Nebenläufigkeit, Konsistenz und Latenz von Nebenläufigkeit und Synchronisationsschemata im Kontext eines asynchronen Interaktiven Echtzeitsystems abschätzen lassen.	4	6 und 7.2.3
7	Die Ergebnisse vom Profiling und Model Checking werden im Kontext einer Beispielanwendung verglichen.	1, 2, 3 und 4	8

2 Stand der Wissenschaft und Technik

Im Rahmen dieser Arbeit werden Nebenläufigkeit, Latenz und Konsistenz im Kontext eines asynchronen Interaktiven Echtzeitsystems mit den Techniken des Profilings und Model Checkings untersucht.

Im Kontext der vier Forschungsfragen dieser Arbeit werden zunächst Informationen zu Interaktiven Echtzeitsystemen benötigt, welche in Abschnitt 2.1 aufgearbeitet werden. Hierbei wird in Abschnitt 2.1.4 insbesondere das Thema Nebenläufigkeit behandelt, um auf dieser Grundlage ein vielversprechendes Modell für die Untersuchung auszuwählen. In einem nebenläufigen System ist die Synchronisation der einzelnen Prozesse notwendig. Techniken hierfür werden in Abschnitt 2.2 vorgestellt. Der Abschnitt 2.3 beschäftigt sich zu Beginn mit Metriken, die die Grundlage bilden, um reproduzierbare Aussagen treffen zu können. Anschließend wird das Thema Profiling behandelt, wobei typische Techniken und Darstellungen im Profiling genannt werden.

2.1 Interaktive Echtzeitsysteme

Neben den bereits in Kapitel 1 erläuterten Anwendungsbereichen von Interaktiven Echtzeitsystemen, werden diese auch für die Bereiche Simulation und Training, Maschinenbau und Medizin verwendet.

Was alle Interaktiven Echtzeitsysteme, unabhängig von ihrem Anwendungsgebiet, gemein haben, sind die vom Benutzer gestellten Anforderungen. Die Computer, auf denen Interaktive Echtzeitsysteme ausgeführt werden, sind *diskret* arbeitende Maschinen. Dennoch soll dem Benutzer der Eindruck eines *kontinuierlichen* Vorgangs entstehen. Aufgrund der *begrenzten zeitlichen Auflösung* der menschlichen Sinne werden diskrete Schritte, die in schneller Abfolge dem Benutzer präsentiert werden, als kontinuierlicher Vorgang wahrgenommen. Die *Frequenz*, in der Benutzer einzelne Ausgaben präsentiert werden, muss, dem angesprochenen Sinn entsprechend, hoch genug sein und darf darüber hinaus möglichst nicht schwanken. Fällt die Frequenz unter einen bestimmten Schwellwert, nimmt der Benutzer, je nach Sinn, wieder einzelne Schritte wahr oder erlebt Wahrnehmungsartefakte.

Eine weitere Anforderung des Benutzers ist, dass das Interaktive Echtzeitsystem innerhalb einer hinreichend kurzen Zeit auf seine Eingaben reagiert, damit er mit dem System interagieren kann. Teather et al. (2009) fanden beispielsweise heraus, dass eine *hohe Latenz* zwischen Eingabe des Benutzers und Reaktion des Systems, die Fähigkeit, sinnvolle Aktionen zu tätigen, mindert.

Interaktive Echtzeitsysteme sind *multimodal*, da sie mehrere Sinne des Menschen ansprechen. Hieraus ergibt sich als dritte Anforderung, dass ein Interaktives Echtzeitsystem die unterschiedlichen Ein- und Ausgabekanäle synchron hält. Dies ist insbesondere

deshalb schwierig, weil verschiedene Ein- und Ausgabekanäle mit unterschiedlichen Frequenzen arbeiten. Während viele Computermonitore mit 60 Hz arbeiten, haben Haptik-Systeme in der Regel mehr als 1 kHz, damit keine ungewollten Vibrationen durch den Anwender wahrgenommen werden (Grange et al., 2001). Sind der visuelle und der akustische Kanal nicht synchron, kann dies zu einer Änderung der Wahrnehmung des Anwenders führen, wie dies z. B. beim Experiment von Mcgurck und Macdonald (1976) gezeigt wurde. Bei diesem Experiment wird einem Probanden ein Video einer Person gezeigt, die eine bestimmte Silbe von sich gibt. Akustisch wird dem Probanden jedoch nicht die echte Tonspur präsentiert, sondern eine, in der die Person eine andere Silbe spricht als für die Videoaufnahme. Im Ergebnis nehmen die meisten Probanden weder die visuell und akustisch vorgespielte Silbe wahr, sondern eine Mischung aus beiden.

Da, wie vorher beschrieben, die Reaktion eines Interaktiven Echtzeitsystems in einer hinreichend kurzen Zeit erfolgen muss, unterliegt solch ein System Echtzeitanforderungen. Shin und Ramanathan (1994) folgend, gibt es drei Kategorien von Echtzeitsystemen, die sich anhand der Konsequenzen bei Nichteinhaltung von zeitlichen Anforderungen definieren:

- Bei *harten* Echtzeitanforderungen hat die Nichteinhaltung katastrophale Konsequenzen
- Bei *festen* Echtzeitanforderungen werden die Daten bei Nichteinhaltung zunehmend nutzlos
- *Weiche* Echtzeitanforderungen fallen in keine der beiden vorherigen Kategorien.

Als Paradebeispiel für Systeme mit harten Echtzeitanforderungen gelten Flugsicherungssysteme oder Kontrollsysteme von Kernkraftwerken. Bei Interaktiven Echtzeitsystemen hingegen drohen in der Regel keine katastrophalen Folgen, wenn zeitliche Anforderungen nicht eingehalten werden. Mit zunehmenden Verzug verringert sich jedoch der Wert der generierten Information. Dies spricht dafür, dass Interaktive Echtzeitsysteme *feste* Echtzeitanforderungen haben. Laut Latoschik (2015) sind die Echtzeitanforderungen nicht weich.

Im Rahmen dieser Arbeit wird anhand dieser Anforderungen ein Interaktives Echtzeitsystem wie folgt definiert:

Definition 1. *Interaktive Echtzeitsysteme nehmen vom Benutzer Eingaben über mehrere Eingabekanäle entgegen und erzeugen unter festen Echtzeitanforderungen für den Benutzer eine kohärente multimodale Erfahrung über mehrere Ausgabekanäle.*

2.1.1 Simulation Loop

Valente et al. (2005) folgend, haben Interaktive Echtzeitsysteme drei Phasen:

- Lesen von Eingaben
- Berechnungen für die Simulation durchführen

- Ausgabe erzeugen

Diese drei Phasen werden, wie in Abbildung 2.1 gezeigt, in einer Schleife ausgeführt. Diese Schleife heißt *Simulation Loop* und ist ein fundamentales Element der meisten Interaktiven Echtzeitsysteme. Im Kontext von Computerspielen wird in der Regel der Begriff *Game Loop* verwendet. Diese Schleife muss schnell genug ausgeführt werden, damit beim Benutzer ein flüssiger, glaubhafter und konsistenter Eindruck der Simulation entsteht. Nach einem vollständigen Durchlauf dieser Schleife stehen dem Benutzer die neuen Ausgaben zur Verfügung. Solch ein vollständiger Durchlauf wird im Allgemeinen *Simulationsschritt* genannt. Im Kontext von Computerspielen wird häufig die Bezeichnung *Frame* verwendet, da hier ein starker Bezug zum Rendering üblich ist. Eine Simulation Loop läuft üblicherweise *nicht* mit der maximal möglichen Frequenz, sondern begrenzt diese mit folgenden Zielen:

- Vermeiden einer Vergeudung von Rechenzeit
- Angleichen der Simulationsgeschwindigkeit auf unterschiedlichen Computern
- Synchronisation von und mit Ein- und Ausgabekanälen, welche unterschiedliche Frequenzen haben

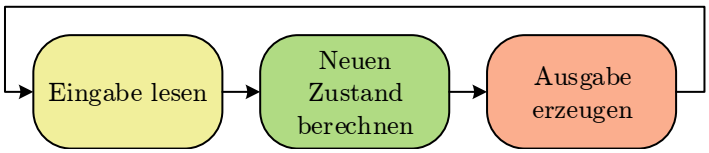


Abbildung 2.1: Ein einfacher *Simulation Loop* mit dem Lesen von Eingaben, Berechnung eines neuen Zustandes und der Ausgabe der Ergebnisse. Die farbliche Kennung mit Gelb für Eingabe, Grün für Zustandsberechnung und Rot für Ausgaben wird in der weiteren Arbeit beibehalten.

Abbildung 2.1 stellt jedoch eine starke Vereinfachung dar, da die Ausprägung der drei Phasen je nach Anwendungsgebiet sehr unterschiedlich sind. Während bei Computerspielen die Eingabephase mit Tastatur, Maus oder Game-Pads, recht einfach ist, sind im Bereich der VR multimodale Systeme üblich.

2.1.2 Sub-Systeme und Abhängigkeiten

Ein Interaktives Echtzeitsystem besteht, wie aus Abbildung 2.2 ersichtlich, aus mehreren *Sub-Systemen*. Hierbei sei ein Sub-System wie folgt definiert:

Definition 2. *Ein Sub-System kapselt domänenspezifische Berechnungen oder abstrahiert Ein- und Ausgabekanäle sowie die dahinterliegende Hardware.*

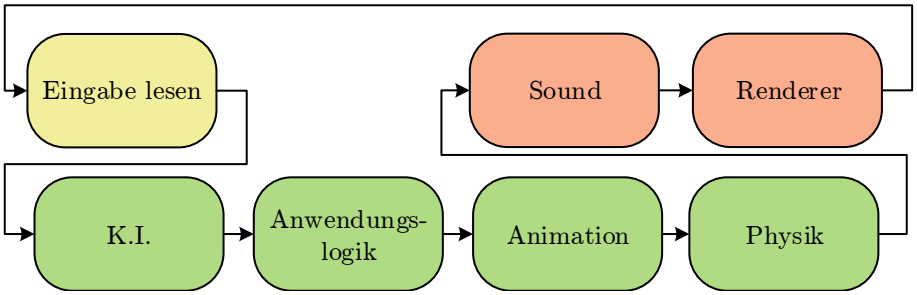


Abbildung 2.2: Ein Simulation Loop mit einigen üblichen Sub-Systemen.

Im Beispiel aus Abbildung 2.2 werden mehrere Sub-Systeme wie eine Künstliche Intelligenz (K.I.), eine Anwendungslogik, eine Animation und eine Physik-Engine in der Simulation verwendet. Eine Ausgabe findet sowohl visuell als auch akustisch statt.

Auch Gabb und Lake (2005) unterteilen eine Game Engine in die oben genannten drei Phasen, stellen diese jedoch in einem Diagramm dar, das in Abbildung 2.3 wiedergegeben ist. Gabb und Lake (2005) identifizierte zwei Arten von Abhängigkeiten, nämlich *inner- and inter-frame dependencies*. Die Abbildung 2.3 zeigt hierbei die logische Abhängigkeit zwischen den Sub-Systemen durch gestrichelte Linien, jedoch nicht die Reihenfolge im Prozessfluss, wie dies in Abbildung 2.2 der Fall ist.

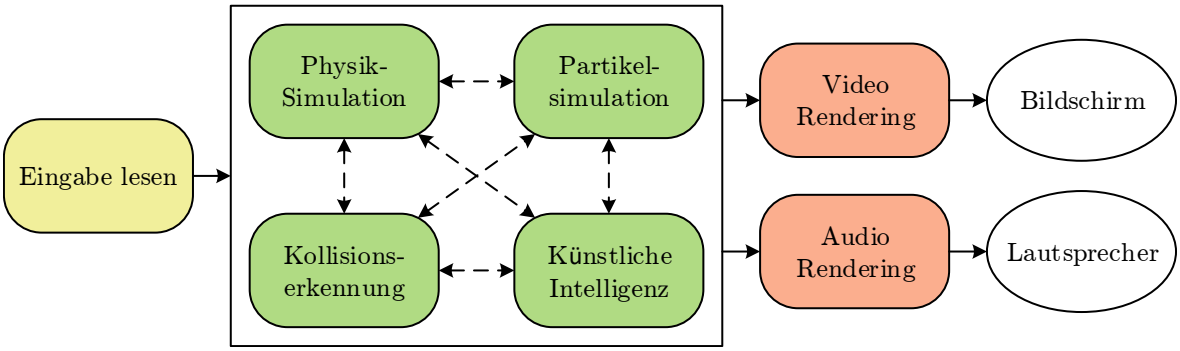


Abbildung 2.3: Abhängigkeit unterschiedlicher Sub-Systeme, wie sie von Gabb und Lake (2005) dargestellt wird.

Der in grün dargestellte Teil der *Simulation* verbraucht in der Regel die meiste Rechenzeit, weshalb die Sub-Systeme, die diesem Bereich angehören, in der Regel das Ziel einer Parallelisierung sein sollten. Eine wesentliche Herausforderung stellen hier die Abhängigkeiten zwischen den Sub-Systemen dar. Mit zunehmender Anzahl der Sub-Systeme nimmt hier die Komplexität stark zu, wie dies durch Blow (2004) anhand mehrerer Anwendungen der letzten Jahrzehnte herausgestellt wird.

Werden die Abhängigkeiten zwischen den Sub-Systemen nicht richtig beachtet, kann sich hieraus ein inkonsistenter Weltzustand ergeben.

Definition 3. *Als Weltzustand wird im Kontext dieser Arbeit eine Menge von Daten*

verstanden, die eine konsistente Beschreibung der simulierten Szene zu einem Zeitpunkt mit all ihren Eigenschaften darstellt.

2.1.3 Begriffe und Rollen

Im folgenden Abschnitt werden einige Begriffe und Rollen definiert und erläutert.

Middleware

Als Middleware wird im Rahmen dieser Arbeit eine Software verstanden, welche die Grundlage bildet, um auf ihrer Basis Anwendungen zu erstellen. Im Kontext von Computer-Spielen ist die analoge Bezeichnung *Game Engine* verbreitet. Beispiele für Game-Engines sind unter anderem Unity 3D (Unity 3D, 2016) und Unreal (Unreal, 2016) und für VR-Middleware Avango (Tramberend, 2003), Lightning (Bues et al., 2008), OpenMASK (Margery et al., 2002) oder Simulator X (Latoschik und Tramberend, 2011b).

Middleware-Entwickler

Der Middleware-Entwickler im Rahmen dieser Arbeit ist eine Person, die eine Middleware, nicht aber eine Anwendung, entwickelt.

Anwendung

Auf der Grundlage einer Middleware wird eine konkrete Anwendung entwickelt. Eine Anwendung ist zum Beispiel ein Computerspiel, eine Anwendung zur Visualisierung und Analyse von wissenschaftlichen oder technischen Daten oder militärische Trainingssimulationen.

Anwendungsentwickler

Ein Anwendungsentwickler entwickelt, im Gegensatz zum Middleware-Entwickler, nicht an der eigentlichen Middleware, sondern an einer Anwendung, die auf Grundlage einer bestehenden Middleware umgesetzt wird. Er nimmt die grundlegende Middleware als gegeben hin und beeinflusst deren Verhalten allenfalls durch Konfiguration einzelner Teile. Auch wenn er nicht an der Entwicklung der Middleware beteiligt ist, so gibt der Anwendungsentwickler mit die Anforderungen an die Middleware vor.

Anwender

Der Anwender verwendet die durch den Anwendungsentwickler erstellte Anwendung. Er ist weder an der Entwicklung der Middleware noch der Anwendung beteiligt. Er gibt primär die Anforderungen an die Anwendung und der darunterliegenden Middleware vor.

2.1.4 Nebenläufigkeit

In diesem Abschnitt wird das Thema Nebenläufigkeit in Interaktiven Echtzeitsystemen behandelt. Hierfür werden zunächst allgemeine Informationen über das Thema Nebenläufigkeit präsentiert. Anschließend wird ein Satz von Kriterien zur Bewertung von Nebenläufigkeitsmodellen für Interaktive Echtzeitsysteme definiert. Auf Grundlage dieser Kriterien werden eine Reihe von Modellen vorgestellt, welche abschließend ausgewertet werden.

Generell gibt es zwei Arten von nebenläufigen Problemen (Breshears, 2009):

- Datenparallele Probleme, bei denen der gleiche Befehl auf jedes Element einer Menge von Daten angewendet wird.
- Taskparallele Probleme, bei denen unterschiedliche Befehle auf die gleichen oder unterschiedliche Daten anwendet werden.

Für die Verarbeitung von datenparallelen Problemen existieren speziell angepasste Prozessoren, die als SIMD-Prozessoren (Single Instruction, Multiple Data) bezeichnet werden. Die Prozessoren der Grafikkarten (GPUs) ließen sich innerhalb der letzten Jahre für immer mehr Bereiche verwenden, weshalb von einer *General Purpose* GPU (GPGPU) gesprochen wird.

Dem gegenüber stehen die MIMD-Prozessoren (Multiple Instruction, Multiple Data), dessen Vertreter die Mehrkernprozessoren sind, welche sich für die Verarbeitung von taskparallelen Problemen eignen.

Auf den ersten Blick gibt es zwei Ansatzpunkte für die Parallelisierung eines Interaktiven Echtzeitsystems.

- Sub-Systeme werden parallel zueinander ausgeführt
- Berechnungen innerhalb eines Sub-Systems erfolgen parallel

Die Berechnungen innerhalb eines Sub-Systems stellen oft klassische datenparallele Probleme, wie beim Renderer oder der Physik-Simulation, dar.

Bei der Parallelisierung eines Interaktiven Echtzeitsystems müssen technische und konzeptionelle Probleme gelöst werden. Das technische Problem ist, dass MIMD-Prozessoren eine erheblich höhere Granularität eines Problems benötigen, um effizient zu arbeiten als SIMD-Prozessoren. SIMD-Prozessoren arbeiten effizient, wenn auf große Datenmengen die gleiche kleine Folge von Instruktionen angewandt wird. MIMD-Prozessoren hingegen haben mehrere voll funktionstüchtige Prozessorkerne. Durch Techniken wie Pipelining und Caching ist der Aufwand bei MIMD-Prozessoren wesentlich höher und die Zeit, in der ein Prozessorkern nicht mit Arbeit versorgt ist, verringert stark die Effizienz.

Gabb und Lake (2005) verdeutlichen das Problem des Overheads anhand eines Partikelsystems. Hier ist es vernünftig, die Frequenz des Partikelsystems an die Frequenz des Renderers zu binden. Ein Renderer läuft üblicherweise mit 30-60 Hz, folglich wird alle 16–33 ms ein neuer Frame erzeugt. Wenn das Partikelsystem nun 5 % der Rechenzeit in

Anspruch nimmt, sind dies lediglich 1,7 ms, was sehr kurz ist und möglicherweise dazu führt, dass sich eine Parallelisierung nicht lohnt, da der Overhead größer ist als der Nutzen.

Darüber hinaus können Effekte wie *false-sharing* und *cache-misses* die Leistung dramatisch reduzieren. Moderne Prozessoren setzen mehrere Caches als Zwischenspeicher ein, um die Latenz zwischen CPU und RAM zu reduzieren. Beim false-sharing arbeiten mehrere Prozessoren mit unterschiedlichen Daten, die jedoch in der gleichen Cache Line sind. In diesem Fall muss ein großer Aufwand betrieben werden, um die Cache Lines zwischen den Prozessoren synchron zu halten. Drepper (2007) zeigt, dass durch diesen Effekt parallelisierte Implementierungen langsamer sein können als sequentielle.

Bei *cache-misses* will die CPU auf Daten zugreifen, die sich nicht im Cache befinden. Folglich müssen diese nachgeladen werden, was einige Zeit in Anspruch nimmt. Auch für cache-misses zeigt Drepper (2007) wie stark sich diese auf die Leistung auswirken.

Das konzeptionelle Problem besteht darin, Tasks der Sub-Systeme zu identifizieren, die parallel zueinander ausgeführt werden können und auf dieser Grundlage die Prozessorkerne fortlaufend mit Arbeit auszulasten. Wenn bei einem System mit vier Prozessorkernen bereits einer nicht verwendet wird, bedeutet dies, dass 25 % der Rechenleistung nicht genutzt wird.

Viele Sub-Systeme parallelisieren ihre Berechnungen bereits mit Techniken für datenparallele Probleme. Innerhalb eines Sub-Systems sind die Abhängigkeiten zwischen Daten klar. Für den Entwickler des einzelnen Sub-Systems ist es am einfachsten, alle Werte, die für die Berechnung relevant sind, zu Beginn zu sammeln, anschließend die Berechnung parallel durchzuführen und dann die Ergebnisse anderen Sub-Systemen mitzuteilen. Dies erlaubt eine gute Optimierung für die jeweilige Domäne des Sub-Systems, impliziert jedoch eine sequentielle Ausführung der einzelnen Sub-Systeme.

Gabb und Lake (2005) weisen darüber hinaus darauf hin, dass mehrere Simulationsschritte nicht gleichzeitig berechnet werden können, da äußere Ereignisse wie die Eingabe des Benutzers nicht im Voraus bekannt sind.

Nebenläufige Software zu entwickeln besteht aus drei Aufgaben:

- *Zerlegen* des Problems in einzelne nebenläufige Aufgaben
- *Synchronisation* zwischen den Aufgaben
- *Zusammensetzen* der Ergebnisse der nebenläufigen Aufgaben zu einem konsistenten Ergebnis

Wie diese drei Aufgaben gelöst werden, wird durch das Modell bestimmt, auf dem das Interaktive Echtzeitsystem aufgebaut wird. Im Folgenden werden eine Reihe von Modellen mit Blick auf die Parallelisierung vorgestellt.

Kriterien

Die Modelle werden anhand der folgenden fünf Kriterien (MK1–5) bewertet. Das Ergebnis des Vergleichs wird in Tabelle 2.1 auf Seite 26 zusammengefasst.

MK1: Skalierbarkeit Mit diesem Kriterium wird bewertet, ob dieses Modell es ermöglicht, ein skalierbares nebenläufiges Interaktives Echtzeitsystem zu implementieren. Skalierbarkeit bedeutet, dass bei zunehmender Anzahl der Prozessoren, diese auch genutzt werden können, ohne dass Anpassungen an der Implementierung erfolgen müssen. Hier ist eine hohe Skalierbarkeit als positiv anzusehen.

MK2: Änderungen an Sub-Systemen Mit der Wahl eines Modells sind u. U. Anpassungen an den verwendeten Sub-Systemen notwendig. Als positiv ist hier anzusehen, wenn keine oder nur wenig Änderungen notwendig sind, da dies den Implementierungsaufwand gering hält.

MK3: Anwendbarkeit auf bestehende Middlewares Mit diesem Kriterium wird bewertet, ob ein Modell auf ein bestehendes Interaktives Echtzeitsystem angewendet werden kann oder ob starke Änderungen auf der Ebene der Middleware notwendig sind. Als positiv anzusehen ist, wenn ein Modell auf bestehende Middlewares angewendet werden kann.

MK4: Bekannte Anwendungen Wenn ein Modell auch in einer existierenden Middlware verwendet wurde, bedeutet dies, dass dieses Modell funktioniert und in der Praxis benutzbar ist.

MK5: Load-Balancing-Fähigkeit Load-Balancing bedeutet, dass die zu erledigende Arbeit gleichmäßig auf die zur Verfügung stehenden Prozessoren verteilt werden kann. Während also die Skalierbarkeit aussagt, ob viele Prozessoren verwendet werden können, besagt die Load-Balancing-Fähigkeit, ob diese Prozessoren auch möglichst gleichmäßig ausgelastet werden können.

Multithread Uncoupled Model

Ein erster einfacher Ansatz zur Parallelisierung eines Interaktiven Echtzeitsystems wurde durch Valente et al. (2005) als *Multithread Uncoupled Model* beschrieben. In diesem in Abbildung 2.4 dargestellten Modell werden zwei Threads verwendet. Der eine Thread liest die Eingabe und führt die Simulation durch, während der andere Thread das Rendering durchführt. Dieses Modell wurde laut Valente et al. (2005) ursprünglich nicht entwickelt, um Mehrkernsysteme auszulasten, sondern um Single-Core Hardware besser auszulasten.

Wie in Abschnitt 2.1.1 beschrieben, enthält ein Simulation Loop eine Begrenzung der Frequenz. Wird der Simulation Loop wesentlich schneller durchlaufen, wird Rechenzeit vergeudet. Das Multithread Uncoupled Model erlaubt es, die Ausgabe in einer festen Frequenz zu machen, während die Frequenz der Simulation von der Leistung des Computers abhängt. Im Gegensatz zur Beschreibung von Valente et al. (2005) wurde in Abbildung 2.4 auch der Weltzustand mit in die Grafik eingefügt, weil beide Threads sich über diesen synchronisieren müssen. Hierbei findet die Kommunikation jedoch nur ein eine Richtung statt, die Ausgabe schreibt somit nicht zurück in den Weltzustand.

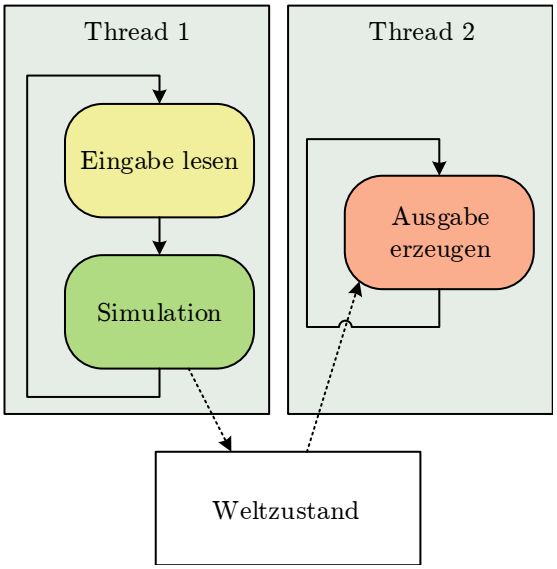


Abbildung 2.4: *Multithread Uncoupled Model*: Ein Thread liest die Eingaben des Benutzers und führt die Simulation durch, während der andere Thread die Ausgabe erzeugt. Die Synchronisation findet über den Weltzustand statt, wobei die Flussrichtung der Daten unidirektional ist.

Synchronous Functional Parallel Model

Ein Blick auf die Sub-Systeme in Abbildung 2.2 zeigt, dass einige dieser Sub-Systeme parallel zueinander ausgeführt werden können. Mönkkönen (2006) bezeichnet dies als *Synchronous Functional Parallel Model*. Abbildung 2.5 zeigt hier, wie exemplarisch die Animation parallel zur Physik-Simulation ausgeführt wird. Dieses Modell kann auf Sub-Systeme angewendet werden, die untereinander keine Abhängigkeit in ihren Daten haben.

Sarmiento (2004) nennt mehrere Beispiele, in denen dieses Modell verwendet wurde. Auch Gregory (2009) beschreibt dieses Modell und gibt Vorschläge, welche Aufgaben die unterschiedlichen Threads übernehmen können.

Ein wesentlicher Vorteil dieses Modells ist, dass keine oder kaum Änderungen an den Sub-Systemen notwendig sind (Mönkkönen, 2006). Lediglich der Simulation Loop muss verändert werden. Dieses Modell skaliert jedoch nicht sehr gut, da die Anzahl der Sub-Systeme begrenzt ist (Mönkkönen, 2006; Best et al., 2009). Es eignet sich am besten für existierende Systeme, die leicht optimiert werden sollen.

Simulation/Output Parallel Model

Auch Simulation und Ausgabe sowie Sub-Systeme mit Abhängigkeiten in den Daten lassen sich parallelisieren. Dieses Modell heißt *Simulation/Output Parallel Model*. Hierzu werden mehrere Instanzen des Weltzustandes benötigt, wie dies in Abbildung 2.6 zu sehen ist. Während die Sub-Systeme für die Simulation einen neuen Weltzustand berechnen, nutzt der Renderer den Weltzustand des vorherigen Durchlaufs der Simulation.

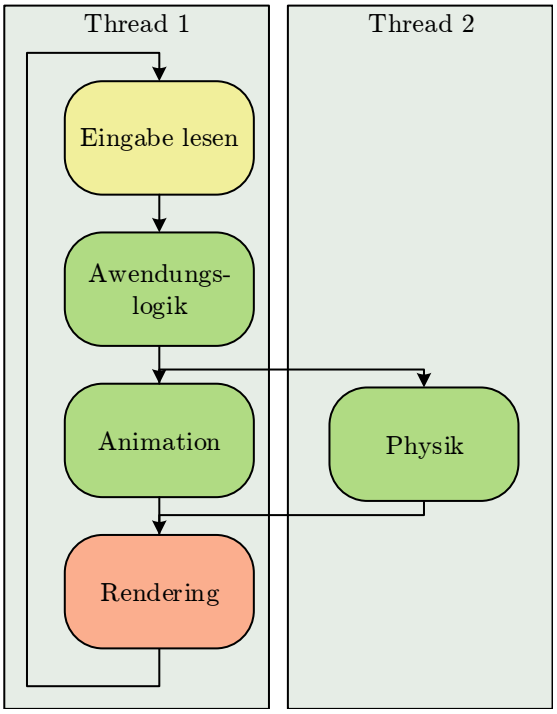


Abbildung 2.5: *Synchronous Functional Parallel Model*: Sub-Systeme, die eine Simulation durchführen, werden parallel zueinander ausgeführt, sofern keine Abhängigkeit in den verwendeten Daten besteht.

Dieses Modell hat zwei Nachteile:

1. Da mehrere Weltzustände verwendet werden, wird mehr Speicher verbraucht.
2. Da z. B. der Renderer den Weltzustand des vorherigen Durchlaufs verwendet, erhöht dieses Verfahren die Latenz.

Wie bereits vorher erwähnt, erlaubt dieses Modell auch Sub-Systeme zueinander parallel laufen zu lassen, die eigentlich eine Abhängigkeit der Daten haben. Jede Stufe erhöht hierbei jedoch den Speicherbedarf und die Latenz.

Chabukswar et al. (2005) nutzte dieses Modell, um die Physik-Simulation und das Rendering zueinander parallel auszuführen. Auch Sarmiento (2004) beschreibt, wie dieses Modell in einer Anwendung eingesetzt wurde. Darüber hinaus unterstützte auch IRIS Performer dieses Modell, um den Anwendungscode mit dem Vorgangs des Cullings und des Zeichnens zu parallelisieren. Rohlf und Helman (1994) untersuchten hierbei detailliert, wie dies die Latenz beeinflusst.

Data-parallel Model

Da in absehbarer Zeit die Anzahl der Kerne die Anzahl der Sub-Systeme übersteigen wird (Borkar und Chien, 2011), schlägt Mönkkönen (2006) vor, die Sichtweise zu ändern

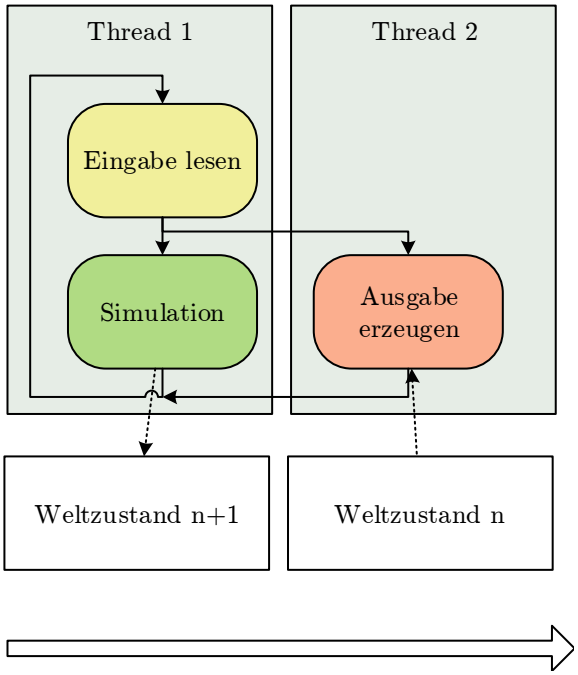


Abbildung 2.6: *Simulation/Output Parallel Model*: Sub-Systeme mit Datenabhängigkeit zwischen einander werden parallelisiert, in dem mehrere Instanzen des Weltzustandes existieren und von Sub-System zu Sub-System weitergereicht werden.

und den Fokus nicht auf die Sub-Systeme, sondern auf die simulierten Objekte zu legen. Hierbei werden dann die Entitäten datenparallel verarbeitet. Die einzelnen Sub-Systeme sind hierbei Funktionen, die auf die einzelnen Objekte angewandt werden. Wie in Abbildung 2.7 dargestellt ist, können durch dieses Modell so viele Kerne verwendet werden, wie Entitäten in einer Szene sind.

Um dieses Modell in einem konkreten System umzusetzen, bedarf es jedoch einen höheren Aufwand als in den vorher beschriebenen Modellen. In den vorherigen Modellen können die Internas des jeweiligen Sub-Systems bei der Integration in ein System als monolithischer Block behandelt werden. Das bedeutet, dass Daten eines kompletten Weltzustandes an dieses Sub-System übergeben werden, Berechnungen ausgeführt werden und im Anschluss die Ergebnisse abgerufen werden können. Wie und wann dieses Sub-System Berechnungen durchführt, bleibt verborgen. Für die Anwendung des Data-parallel Model ist es jedoch notwendig, dass feingranular gesteuert werden kann, wann welche Berechnungen von welchem Sub-System für welches simulierte Objekt durchgeführt werden.

Dieses Modell führt das Zerlegen auf der Ebene der simulierten Objekte durch. Jedes Sub-System stellt hierbei einen Task dar, der auf diese Entität ausgeführt wird. Eine Synchronisation findet dadurch statt, dass nur ein Typ von Tasks gleichzeitig auf alle Objekte ausgeführt wird und anschließend synchronisiert wird.

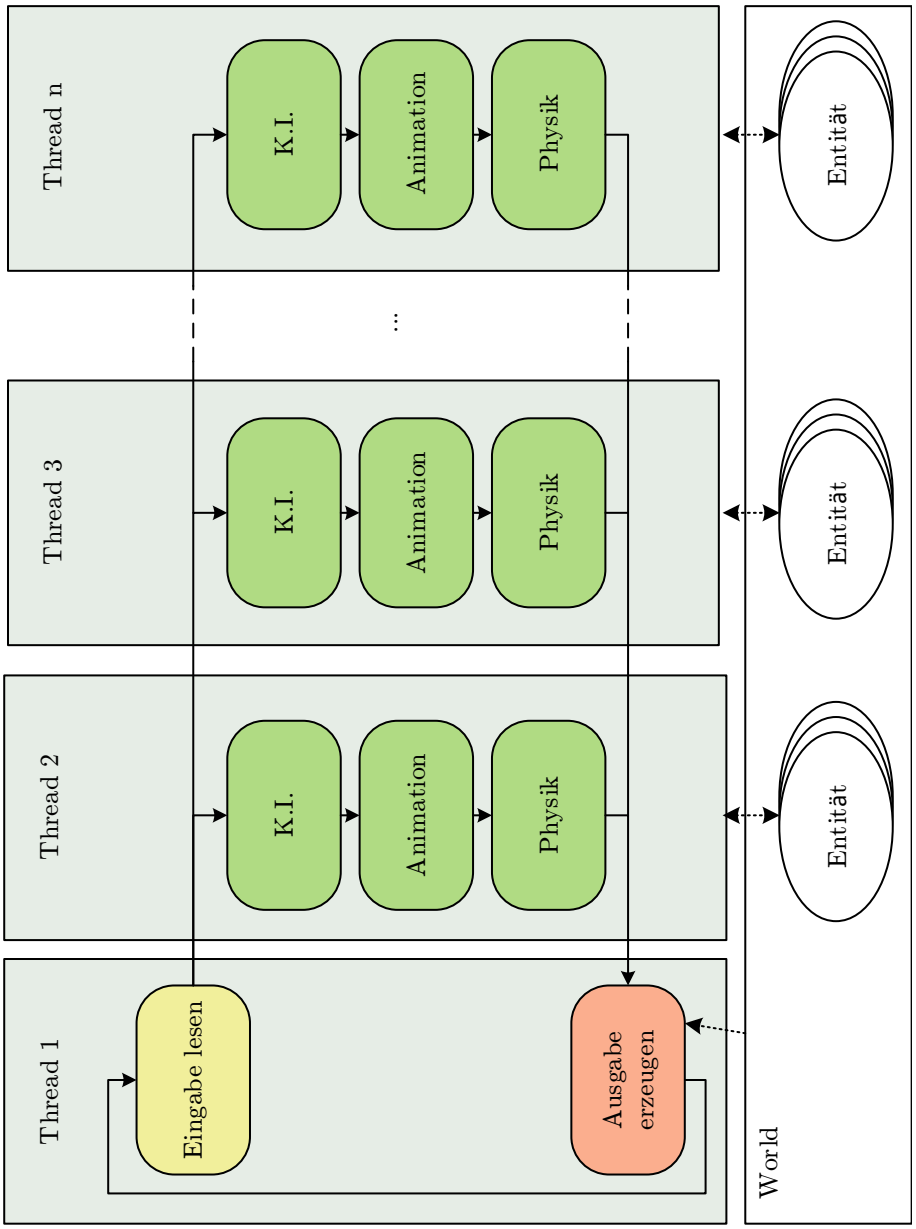


Abbildung 2.7: *Data-parallel Model*: Die Verarbeitung der einzelnen Elemente findet parallel statt. Hierbei werden die einzelnen Sub-Systeme als Operationen verstanden, die auf die simulierten Objekte angewendet werden.

Task-Tree Model

Im *Task-Tree Model* werden die Tasks vorher nach ihren Abhängigkeiten in einem Baum geordnet. Durch die Abhängigkeit ist klar, welche Tasks nebenläufig sind und parallel verarbeitet werden können. Ein Scheduler verteilt diese Aufgaben dann auf die Kerne, die zur Verfügung stehen. Tulip et al. (2006) bezeichnet diesen Ansatz als flexibel, skalierend und effizient.

Dieses Modell wurde durch El Rhalibi et al. (2005) verwendet um die Tasks in einer *Game Loop* zu parallelisieren. Sie identifizierten 19 Tasks und die Abhängigkeiten eines kleinen capture-the-flag Spiels mit K.I. gesteuerten Computergegnern. Durch die Verwendung von *Bernstein's constraints* erzeugten El Rhalibi et al. (2005) einen zyklischen Graphen, der die Abhängigkeiten repräsentiert.

Über den Leistungszuwachs, der durch dieses Modell erreicht wurde, gibt es unterschiedliche Angaben. Während Tulip et al. (2006) über einen Leistungszuwachs berichten, berichten El Rhalibi et al. (2005) über eine abnehmende Leistung.

Task-Verteilung zwischen GPU und CPU Die Verwendung von SIMD Prozessoren ist sehr verbreitet bei Interaktiven Echtzeitsystemen. Joselli et al. (2010) stellen eine Implementierung vor, in der die Tasks zwischen CPU und GPU verteilt werden. Wie in Abbildung 2.8 gezeigt, ist ein Thread hierbei für die Kommunikation mit der GPU zuständig.

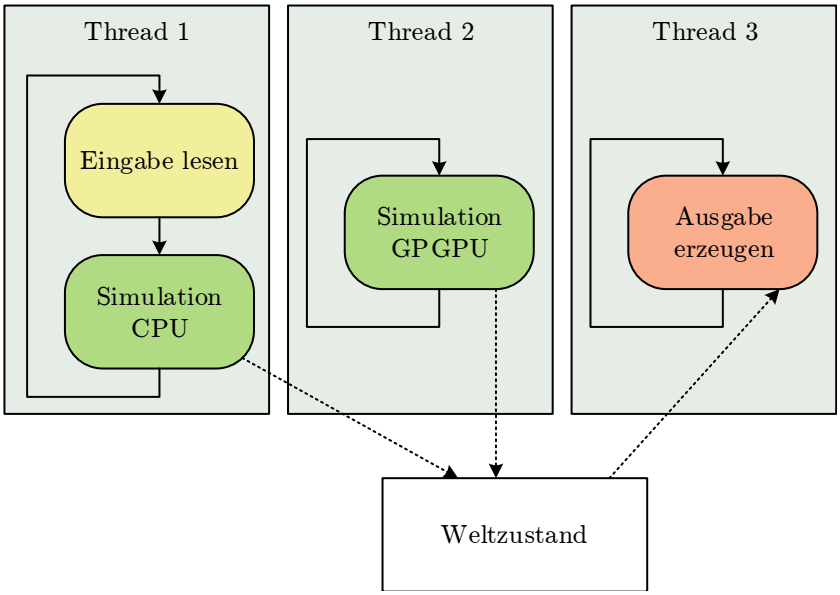


Abbildung 2.8: *Task-Tree Model mit GPGPU*: Der erste Thread ist verantwortlich für den Simulation Loop, der zweite für die Kommunikation mit der GPGPU, der Dritte für die Erzeugung der Ausgabe.

Laut Joselli et al. (2010) existieren drei Arten von Tasks:

1. Tasks, die sequentiell auf der CPU laufen
2. Tasks, die parallelisiert auf der CPU laufen
3. Tasks, die auf der GPU laufen

Einige Tasks, wie z. B. die Kommunikation mit Eingabegeräten, können ausschließlich für die CPU implementiert werden. Viele Tasks der Simulation hingegen lassen sich in allen drei Formen implementieren. Der System- oder Anwendungsentwickler müsste hierbei alle Implementierungen zur Verfügung stellen, um vollständig von diesem Modell zu profitieren. Wenn mehrere Implementierungen verfügbar sind, kann entschieden werden auf welchem Prozessor ein Task ausgeführt wird. Dies ist hilfreich, wenn ein Prozessor bereits vollständig mit Arbeit ausgelastet ist.

Die Entscheidung, ob ein Tasks auf der CPU oder der GPU ausgeführt wird, kann manuell getroffen oder einer von sechs beschriebenen Heuristiken überlassen werden.

Joselli et al. (2010) testeten dieses Modell anhand einer Schwarmsimulation. Interessant hierbei ist, dass die verwendeten Heuristiken niemals die multithreading CPU-Implementierung wählten, da dies stets die langsamste war. Dies zeigt, dass die Granularität solch eines datenparallelen Problems zu fein ist für die CPU.

Consumer/Producer Pattern Based Model

Das Consumer-Producer-Pattern ist in der nebenläufigen Programmierung weit verbreitet (Breshears, 2009). Best et al. (2009) nutzten dies, um darauf aufbauend ein Modell für Interaktive Echtzeitsysteme zu erstellen. Dieses Modell wird in Abbildung 2.9 dargestellt.

Von jedem Producer gibt es lediglich eine Instanz. Die Aufgabe eines Producers ist es, Tasks zu erstellen, die nebenläufig abgearbeitet werden können. Von den Consumern gibt es mehrere Instanzen, welche die von den Producern erstellten Aufgaben abarbeiten.

Beispielsweise kann bei einer Physiksimulation der Producer Kollisionen berechnen, während der Consumer die Auswirkungen dieser Kollisionen berechnet. Oder bei einer K.I. berechnet der Producer die Sichtbarkeit zwischen verschiedenen Entitäten, während die Consumer die Reaktion der Entität berechnen.

Der große Vorteil dieses Modells ist, dass die Consumer sofort mit der Arbeit beginnen können, selbst wenn die Producer noch nicht alle Aufgaben erstellt haben, was zu einer guten Auslastung des Systems führt.

Best et al. (2009) wendeten ihr Modell auf das Open-Source-Spiel *Cube 2* an und erreichten einen um 51 % reduzierten Zeitbedarf für die Simulation auf einem System mit acht Prozessorkernen. Neben diesem sehr guten Ergebnis zeigt sich auch, dass für einen Benchmark am besten vollständige und echte Anwendungen verwendet werden sollten, anstatt kleiner künstlicher Benchmarks. .

Ein weiteres System, welches dieses Modell verwendet, ist OpenMASK (Margery et al., 2002). Es zerlegt Tasks auf der Ebene der simulierten Objekte. Diese Tasks werden in einer festen Frequenz abgearbeitet.

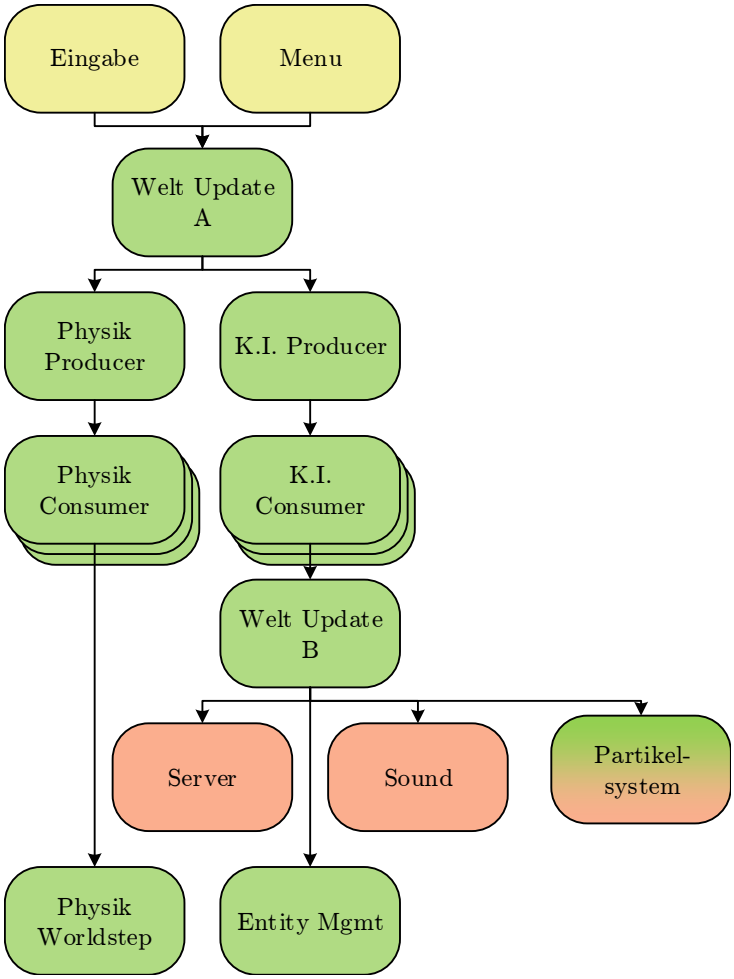


Abbildung 2.9: *Consumer/Producer Pattern Based Model*: Das von Best et al. (2009) entwickelte Modell zur Anwendung des Consumer-Producer-Patterns auf ein Interaktives Echtzeitsystem.

Free-Step Mode Model

Alle vorherigen Modelle hatten immer noch gemeinsam, dass sie grundlegend auf dem Simulation Loop beruhten und ein neuer Weltzustand immer wieder durch einen vollständigen Schleifendurchlauf erzeugt wurde. Bei jedem Durchlauf dieser Schleife fand somit eine Synchronisation statt. Diese Modelle behandeln jedoch nicht den Fall, dass einzelne Sub-Systeme wesentlich kürzer oder länger für ihre Berechnungen brauchen könnten als andere. Andrews (2009) nennt dieses in Abbildung 2.10 dargestellte Vorgehen *Lock-Step Mode*.

Anstatt solch eines starren Taktes, könnte man auch die Sub-Systeme entscheiden lassen, wie lang diese brauchen. Andrews (2009) nennt dies *Free-Step Mode*. Ein *Clock-Step* ist hierbei keine Zeitspanne einer bestimmten Länge, sondern die Zeit, die das

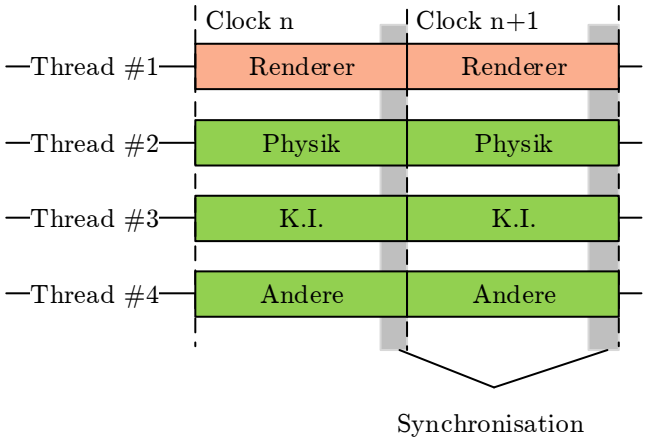


Abbildung 2.10: *Lock-Step Mode*: Alle Sub-Systeme synchronisieren sich am Ende eines Durchlaufs des Systems.

System braucht, um eine neue Ausgabe zu erzeugen. Jedes Sub-System entscheidet hierbei selbst, wie viele Clock-Steps es braucht, wodurch wie in Abbildung 2.11 einzelne Sub-Systeme auch mehr Zeit für ihre Berechnungen haben können. Andrews (2009) stellt keine Ergebnisse eines Benchmarks zur Verfügung.

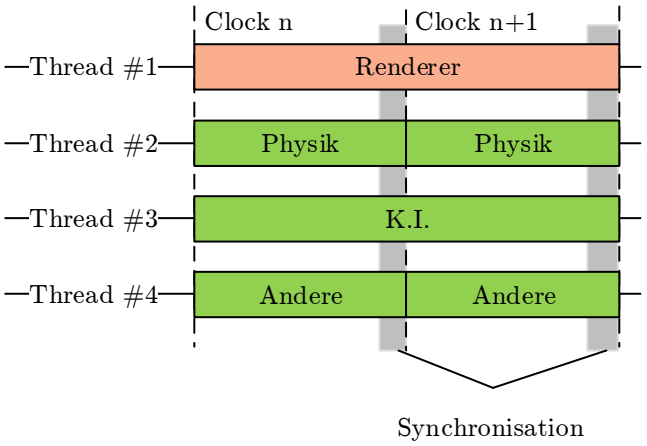


Abbildung 2.11: *Free-Step Mode*: Es ist eine Taktung vorgegeben, wobei jedes Sub-System entscheiden kann, wie viele dieser Taktzyklen es für seine Berechnung braucht.

Asynchronous Model

Ein asynchrones Modell ohne zentralen Simulation Loop wurde durch Gabb und Lake (2005) sowie Mönkkönen (2006) vorgestellt. In einem asynchronen Modell ist die Ausfüh-

rungszeit nicht an die Zeit zur Erstellung eines Ausgabe-Frames gekoppelt. Wie bereits aus Abbildung 2.3 ersichtlich, ist es schwierig, sämtliche Abhängigkeiten zwischen den Sub-Systemen zu eliminieren. Deshalb schlägt Gabb und Lake (2005) vor, dass alle Sub-Systeme mit ihrer eigenen Frequenz laufen und Berechnungen auf Grundlage ihres zuletzt bekannten Weltzustandes durchführen, wie dies in Abbildung 2.12 dargestellt wird.

Als Beispiel erzeugt der Renderer fortlaufend neue Frames, auf Basis des zuletzt bekannten Weltzustandes. Währenddessen erzeugen sowohl die K.I., die Animation und die Physik-Simulation neue Daten. Egal ob einer oder mehrere dieser Sub-Systeme länger brauchen, der Renderer erzeugt einen neuen Frame. Hierdurch wird jedoch die Konsistenz des Weltzustandes ein zu lösendes Problem.

Gabb und Lake (2005) nennen die Komplexität der Software und die Verwaltung von geteilten Daten als Hauptproblem dieses Modells. Darüber hinaus gestaltet sich auch das Benchmarking eines solchen Systems schwierig, da z. B. die weit verbreitete Metrik „Frames pro Sekunde“ nur noch eine Aussage über den Renderer, aber nicht mehr über das gesamte System liefert. Auf diese Problematik wird später noch einmal gesondert eingegangen.

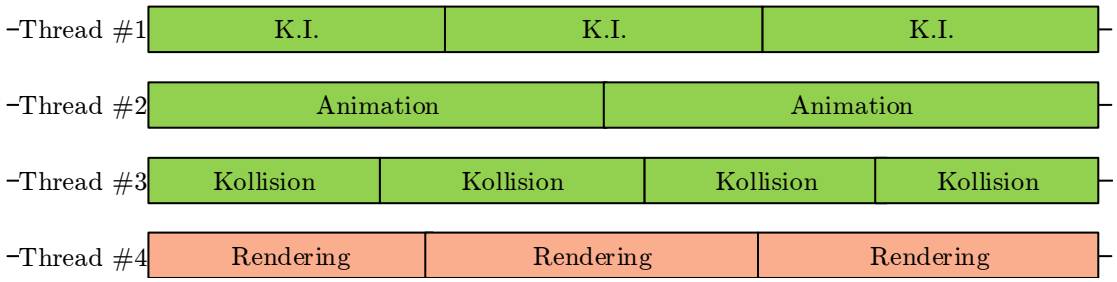


Abbildung 2.12: *Asynchronous Model*: Es existiert kein zentral vorgegebener Takt und alle Sub-Systeme laufen asynchron zueinander.

Simulator X (Latoschik und Tramberend, 2011b,a) stellt eine Implementierung dieses Modells dar. Simulator X basiert auf dem Hewitts Actor Model (Hewitt et al., 1973). Auf Grundlage des Message-Passings des Actor Models kapselt Simulator X die Sub-Systeme in Komponenten, stellt simulierte Objekte als Entitäten und deren Eigenschaften als Zustandsvariablen dar.

Latoschik und Tramberend (2011b,a) geben an, dass sich Simulator X eignet, um Systeme mit unterschiedlicher Granularität zu erstellen, da Komponenten entweder monolithische Sub-Systeme kapseln können oder zur Berechnung weitere Prozesse in Form von Aktoren starten können.

Auswertung

In Tabelle 2.1 wird das Ergebnis des Vergleichs zusammengefasst. Hierbei ist zu erkennen, dass das *Asynchronous Model* nach den vorher definierten Kriterien das geeignetste Modell für die Untersuchung ist.

Tabelle 2.1: Vergleich von acht Modellen für nebenläufige Interaktive Echtzeitsysteme. Es werde die fünf Bewertungsstufen ++, +, 0, - und −− verwendet. Die Bewertung ++ bedeutet, dass ein Modell hier eine besonders positive Eigenschaft hat, die Bewertung −−, dass das Modell hier eine besonders negative Eigenschaft hat. Die Einstufung erfolgt anhand der Auswertung der vorher präsentierten Informationen zu den jeweiligen Modellen.

	Multithread Uncoupled Model	Synchronous Funktional Parallel Model	Simulation/Output Parallel Model	Data-Parallel Model	Task-Tree Model	Consumer/Producer Based Model	Free-Step Mode Model	Asynchronous Model
MK1: Skalierbarkeit	−−	−−	-	++	++	+	+	+
MK2: Änderungen an Sub-Systemen	++	++	++	−−	−−	0	+	++
MK3: Anwendbarkeit	++	++	++	−−	−−	++	+	+
MK4: Bekannte Anwendungen	++	++	++	−−	++	++	+	++
MK5: Load-Balancing-Fähigkeit	−−	−−	-	++	++	++	+	++
Punkte	2	2	2	-2	2	7	5	8

2.1.5 Latenz

Latenz ist im Allgemeinen die Verzögerung zwischen Ursache und Wirkung. In Bezug auf Interaktive Echtzeitsysteme ist die für den Benutzer wichtigste Latenz, die *Ende-zu-Ende-Latenz*. Laut Steed (2008) ist die Ende-zu-Ende-Latenz die Zeit, die es braucht, dass Änderungen des Zustands an einem Eingabegerät zu einer Reaktion auf dem Anzeigegerät führen. Eine hohe Ende-zu-Ende-Latenz haben laut Frank et al. (1988) sowie Di Luca (2010) folgende Auswirkungen:

- Unwohlsein oder Übelkeit beim Anwender, auch bekannt als *Simulator Sickness*
- Änderung des Benutzerverhaltens und Minderung seiner Effizient bei der Interaktion mit Objekten in der simulierten Szene
- Fehlwahrnehmungen, aufgrund widersprüchlicher Informationen aus unterschiedlichen Sinnen

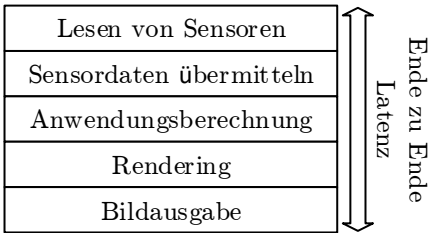


Abbildung 2.13: Ursachen, die laut Mine (1993) sowie Steed (2008) zur Ende-zu-Ende-Latenz beitragen.

Auf Grundlage der Erkenntnisse von Mine (1993) identifizierte Steed (2008) die in Abbildung 2.13 dargestellten Ursachen für Latenz. Für die Reduzierung der Latenz wurden eine Vielzahl von Techniken entwickelt. Beim *Post-Rendering 3D Warping* (Mark et al., 1997) wird das bereits gerenderte Bild am Ende der Rendering-Prozedur an die neu ermittelte Position und Ausrichtung des Kopfes des Benutzers angepasst. Dies reduziert die Latenz zwischen Kopfbewegung und Ausgabe des gerenderten Bilds, was auch erfolgreich bei der Oculus Rift eingesetzt wurde (Beeler und Gosalia, 2016). Ähnlich dazu werden beim *Frameless Rendering* (Dayal et al., 2005) nur einzelne Pixel aktualisiert um die letzten Änderungen darzustellen.

Beide Techniken beschäftigen sich damit, die Latenz zwischen dem Lesen eines Sensors und dem Rendering zu minimieren. Sie eigenen sich jedoch nicht, eine Latenz in der Anwendungsberechnung zu eliminieren.

2.1.6 Kommunikation und Konsistenz

Wird eines der in Abschnitt 2.1.4 beschriebenen Modelle für die Nebenläufigkeit verwendet, muss dafür gesorgt werden, dass neue Weltzustände zwischen den nebenläufigen Teilen der Systeme kommuniziert werden und dass diese eine konsistente Sicht auf diesen haben. Valente et al. (2005) schreiben, dass bereits beim einfachen *Multithread Uncoupled Model* die typischen Probleme bei der parallelen Programmierung auftreten. In diesem Modell muss zum Beispiel sichergestellt werden, dass der zweite Thread, der für das Rendering zuständig ist, nicht Daten liest, die gerade vom ersten Thread verändert werden. Die Daten fließen bei diesem Modell jedoch nur in eine Richtung, von der Eingabe zur Ausgabe.

In einem Interaktiven Echtzeitsystem ohne Nebenläufigkeit, welches also einfach nur einen, wie in Abbildung 2.1 dargestellten, Simulation Loop hat, kann der Weltzustand in einer nahezu beliebigen Art im Speicher gehalten werden, da ohnehin nur ein Thread hierauf lesend sowie schreibend zugreift. Komplizierter ist dies jedoch beim Asynchronen Modell, wie dies in Abbildung 2.12 dargestellt ist. Denn hier greifen mehrere Threads gleichzeitig lesend sowie schreibend auf die Daten zu.

Generell gibt es zwei Strategien, um mit diesem Umstand umzugehen:

- Es gibt weiterhin einen globalen Weltzustand, wobei durch einen Synchronisationsmechanismus sichergestellt wird, dass nur konsistente Daten in diesen geschrieben

und von diesem gelesen werden.

- Jedes Sub-System hat einen eigenen lokalen Weltzustand, aus dem gelesen wird und in den er schreibt. Hierbei wird ein Mechanismus benötigt, der dann die lokalen Weltzustände untereinander synchronisiert.

Gerade der zweite Lösungsweg stellt hierbei nur die spezielle Ausprägung eines allgemeinen Problems dar, welcher im Kontext von *verteilten Systemen* gut erforscht ist. *Verteilte Interaktive Echtzeitsysteme* stehen regelmäßig vor dem Problem, wie sie es realisieren, mehreren Anwendern, die mit ihren eigenen Clients zu einer Simulation verbunden sind, eine konsistente Sicht auf die Welt zu ermöglichen. Beispiel für solch eine Anwendung ist jedes Multi-Player-Online Spiel (z. B. Counterstrike, EVE Online, League of Legends oder World of Warcraft) oder militärische Trainingsanwendungen wie SIMNET (Miller und Thorpe, 1995).

Steed und Oliveira (2009) fassten umfänglich verschiedene Techniken zur Synchronisation von Verteilten Interaktiven Echtzeitanwendungen zusammen.

Eine inkonsistente Sicht auf den Weltzustand kann einzelne Benutzer zu Fehlentscheidungen verleiten, was z. B. im Kontext eines Spiels zu unfairen Situationen führen kann. Solche unfairen Situationen führen zu einer geringeren Akzeptanz einer Anwendung.

In Bezug auf die Konsistenz innerhalb eines Interaktiven Echtzeitsystems ist ebenso anzunehmen, dass Inkonsistenzen dazu führen, dass ein Anwender eine Applikation als nicht zufriedenstellend ansehen wird.

2.2 Techniken zur Synchronisation in asynchronen Systemen

Beide im letzten Abschnitt 2.1.6 genannten Strategien für einen konsistenten Weltzustand benötigen Techniken zur Synchronisation. Im Folgenden werden sowohl Techniken vorgestellt, um einen globalen Weltzustand zu implementieren, als auch, um ein System mit lokalen Weltzuständen für jeden Prozess zu realisieren.

2.2.1 Kriterien

Die Techniken zur Synchronisation in einem asynchronen System werden nach den folgenden sieben Kriterien untersucht und bewertet.

SyK1: Skalierbarkeit

Mit diesem Kriterium wird bewertet, ob diese Technik zur Synchronisation es ermöglicht, ein skalierbares nebenläufiges Interaktives Echtzeitsystem zu implementieren (vgl. Paragraph 2.1.4).

SyK2: Overhead

Jede Synchronisationstechnik hat einen Overhead, bedingt durch die zugrundeliegenden Verfahren und Algorithmen. Mit diesem Kriterium wird bewertet, wie hoch der Overhead der Technik ist.

SyK3: Speicherbedarf

Bei einigen Synchronisationstechniken wird u. U. mehr Arbeitsspeicher für Kopien von Daten gebraucht. Hier ist es wünschenswert, wenn durch die Technik möglichst wenig zusätzlicher Speicherbedarf entsteht.

SyK4: Ausgereiftheit

Neue Technologien leiden u. U. unter Bugs oder Problemen bei der Performance. In diesem Kriterium wird abgebildet, wie ausgereift eine Technologie ist.

SyK5: Entwicklungszeit

Die Zeit, die benötigt wird, um eine Technologie in einer Implementierung zu verwenden, ist ein entscheidender ökonomischer Faktor. Eine sehr performante Technologie ist u. U. nicht die beste Grundlage, wenn der Aufwand in der Implementierung extrem ansteigt.

SyK6: Zuverlässigkeit

In der parallelen Programmierung kommt es häufig zu typischen Problemen, wie beispielsweise einem Deadlock. Mit diesem Kriterium wird bewertet, ob eine Technologie die typischen Probleme vermeidet.

SyK7: Anwendbarkeit auf bestehende Middlewares

In diesem Kriterium wird bewertet, ob eine Technologie in bestehenden Middlewares leicht verwendet werden kann.

2.2.2 Techniken für einen geteilten globalen Weltzustand

Wie in Abbildung 2.14 dargestellt, könnten alle nebenläufigen Threads auf einen gemeinsamen Weltzustand zugreifen. Dies erfordert, dass der Schreib- und Lesezugriff so geregelt wird, dass keine inkonsistenten Daten entstehen.

Abbildung 2.14: Schematische Darstellung eines zentralen Weltzustandes. Alle nebenläufigen Prozesse greifen auf diesen zu. Hier ist eine Technik zur Regelung des Zugriffs notwendig.

Klassische Locks

Ein erster Ansatz wäre hier die Verwendung klassischer *Locks*, die sicherstellen das nur ein Thread zur gleichen Zeit auf die Daten zugreift. Entscheidend ist hierfür die Granularität der Daten, die durch den Lock geschützt werden. So könnte der gesamte Weltzustand geschützt werden, was laut Zyulkyarov et al. (2009) und Gajinov et al. (2009) ein sehr einfacher, jedoch auch wenig performanter Ansatz wäre, da die Threads einen großen Teil der Zeit mit Warten verbringen würden und dieser Ansatz auch schlecht skaliert. Die Skalierbarkeit war im Fall von Gajinov et al. (2009) sogar negativ, was bedeutet, dass die Implementierung mit jedem zusätzlichen Thread langsamer wurde. Eine feinere Granularität schafft hier Abhilfe.

Abdelkhalek und Bilas (2004) evaluierten diesen Ansatz bei der Parallelisierung des Game Servers des Computerspiels *Quake*. Hierzu wurde vorab sichergestellt, dass es die Kapazität der CPU ist, die die Anzahl der Spieler auf einem Server begrenzt und nicht etwa die Netzwerkverbindung (Abdelkhalek et al., 2003). Hierbei stellten Abdelkhalek und Bilas (2004) fest, dass die Struktur des Locks im Speicher den geometrischen Gegebenheiten in den simulierten Szenen entsprechen sollten. In einem Benchmark zeigten sie, dass der mit Locks parallelisierte Server 25 % mehr Benutzer bedienen konnte als die sequentielle Implementierung. Der Overhead des Lockings betrug im Verhältnis zur Ausführungszeit 35 %. Die Wartezeit in Locks betrug 40 % der absoluten Ausführungszeit des Servers. Abdelkhalek und Bilas (2004) äußerten jedoch die Vermutung, dass diese durch weitere anwendungsspezifische Anpassungen auf 20 % reduziert werden könnte.

Software Transactional Memory

Software Transactional Memory (STM) ist eine lock-freie Art der Synchronisation (Herlihy und Moss, 1993; Adl-Tabatabai et al., 2006; Drepper, 2008). STM orientiert sich hierbei an transaktionalen Datenbanken. Diese haben das Problem, dass zwei verbundene Benutzer u. U. innerhalb von komplexen Operationen die gleichen Daten verändern, was in der Regel zu Inkonsistenzen und beschädigten Daten führt. Um dieses Problem zu lösen, können komplexe Operationen in eine *Transaktion* zusammengefasst werden. Werden während der Ausführung von Transaktion A Teile der Daten, die diese Transaktion verändern würde, bereits erfolgreich durch Transaktion B verändert, dann werden die gemachten Änderungen von A zurückgesetzt und es wird anschließend erneut versucht, auf Grundlage der aktualisierten Daten die Transaktion A auszuführen. Eine Transaktion fasst somit mehrere Aktionen zu einem atomaren Block zusammen, der entweder ganz oder gar nicht ausgeführt wird.

Für Anwendungsentwickler ist STM einfacher zu verwenden als klassische Locking-Systeme, welche Lee (2006) zufolge ohnehin nicht beherrschbar sind. Die Verwendung von STM erzeugt jedoch, wie durch Cascaval et al. (2008) herausgestellt, einen sehr großen Overhead. Dieser Overhead kann jedoch laut Sweeney (2006) akzeptabel sein, wenn die eingesetzte Technologie dazu führt, dass das Interaktive Echtzeitsystem gut skaliert.

Aufbauend auf der Arbeit von Abdelkhalek und Bilas (2004), parallelisierten Zyul-

kyarov et al. (2009) den Game Server einer reduzierten Variante des Spiels Quake, welches sie daher *Atomic Quake* nannten. Hierbei nutzten sie STM und verglichen die Performance mit einer Variante, die klassisches Locking verwendet, sowie einer Hybrid-Implementierung, welche beide Techniken benutzt. Im Benchmark stellte sich heraus, dass das klassische Locking bei bis zu 8 Threads skaliert, die gemischte Variante bei bis zu 4 Threads und die reine STM Implementierung bei bis zu 2 Threads. Zyulkyarov et al. (2009) deklarierten ihre Ergebnisse jedoch als vorläufig, da sie die Optimierung des Compilers abschalten mussten, um STM verwenden zu können.

Erfolgreicher waren Gajinov et al. (2009) mit ihrer Implementierung mit dem Namen *QuakeTM*. Diese verglichen eine Implementierung mit globalem Lock mit zwei STM Implementierungen, wobei die eine STM Implementierung eine grobe und die andere eine feine Granularität hatte.

Sie fanden heraus, dass die Variante mit dem globalen Lock *negativ skalierte*, also pro hinzugefügtem Thread immer langsamer wurde. Beide Varianten mit STM skalierten, wobei die feingranulare Implementierung performanter war als die grobgranulare Implementierung. Des Weiteren stellten sie heraus, dass die feingranulare STM Implementierung um den Faktor 2 langsamer ist als eine feingranulare Implementierung mit Locks.

Lupei et al. (2010) erreichten schließlich, dass die STM Implementierung eine bessere Performance erreichte als die Lock-basierte Implementierung. Neben eines Benchmarks auf technischer Ebene, werteten Lupei et al. (2010) auch den Implementierungsaufwand aus. Während die Lock-basierte Implementierung mehrere Monate an Analyse, Code-Verfeinerung und Optimierung bedurfte, wurde die STM-Implementierung in weniger als einem Monat erstellt.

Laut Sweeney (2006) ist jedoch eine um zwei- bis vierfach langsamere Implementierung akzeptabel, wenn sie skaliert und die Implementierung vereinfacht. Dies macht STM zu einen interessanten Kandidaten für Interaktive Echtzeitsysteme, wobei eine Untersuchung in Bezug auf Latenz bisher nicht stattgefunden hat und somit noch evaluiert werden muss.

Synchronization via Scheduling

Einen weiteren Ansatz stellten Best et al. (2011) unter dem Namen „Synchronization via Scheduling“ (SvS) vor. Hierbei liegt der Fokus auf effizientem Scheduling von Tasks, um gleichzeitigen Zugriff auf Daten auszuschließen. Best et al. (2011) haben beobachtet, dass in einem Interaktiven Echtzeitsystem zwei Arten von Abhängigkeiten existieren. Bei *expliziten Abhängigkeiten* haben die Tasks eine logische Reihenfolge in der sie abgearbeitet werden müssen. Das Ergebnis unterscheidet sich also, wenn mehrere Tasks mit expliziter Abhängigkeit in unterschiedlicher Reihenfolge ausgeführt werden. Bei *impliziten Abhängigkeiten* verwenden zwei Tasks die gleichen Daten, müssen jedoch nicht in einer bestimmten Reihenfolge ausgeführt werden. Das Ergebnis unterscheidet sich somit nicht, wenn Tasks mit impliziter Abhängigkeit in unterschiedlicher Reihenfolge ausgeführt werden.

Best et al. (2011) stellten fest, dass implizite Abhängigkeiten sehr häufig sind, diese aber in vielen Modellen als explizite Abhängigkeiten behandelt werden. Sie beobachte-

ten weiterhin, dass selbst wenn Tasks implizite Abhängigkeiten haben, ein gleichzeitiger Datenzugriff so gut wie nie stattfindet.

Um auf Grundlage dieser Erkenntnisse unnötiges Locking zu vermeiden, setzt SvS ein zweistufiges Verfahren ein. Im ersten Schritt werden implizite Abhängigkeiten mittels *statischer Code-Analyse* ermittelt. Das Ergebnis sind Abhängigkeiten, die möglicherweise zu Laufzeit auftreten können aber nicht zwangsläufig müssen. Im zweiten Schritt wird zu Laufzeit diese Information verfeinert und überflüssige implizite Abhängigkeiten werden entfernt. Die so gewonnenen Informationen werden von einem Scheduler verwendet, um die Tasks so abzuarbeiten, dass kein gleichzeitiger Zugriff auf Daten stattfindet.

Best et al. (2011) prüften ihren Ansatz an der Animationsbibliothek Cal3d und am Spiel QuakeSquad, und verglichen die Ergebnisse ihres Verfahrens mit Implementierungen die Intels Threading Building Blocks (Intel, 2012) und STM verwendeten. SvS skalierte bis zu dreimal besser als STM und leicht besser als Intels Threading Building Blocks.

Zwischen impliziten und expliziten Abhängigkeiten zu unterscheiden und diese Information für ein intelligentes Scheduling zu verwenden, steigert die Skalierbarkeit und die Leistung. Darüber hinaus wird anwendungsspezifisches Scheduling von Zhuravlev et al. (2012) als zukünftiger Trend für die optimale Nutzung von Mehrkernprozessoren gesehen.

2.2.3 Prozessvariablen

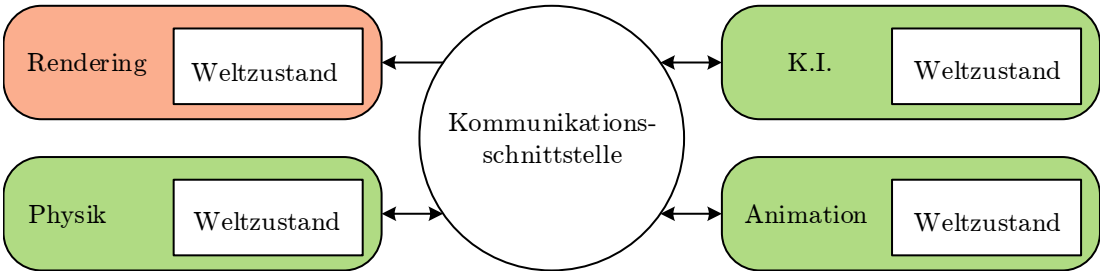


Abbildung 2.15: Schematische Darstellung eines lokalen Weltzustandes. Jeder nebenläufige Prozess hat eine Kopie des Weltzustandes. Diese Kopien müssen untereinander synchronisiert werden.

Anstelle eines globalen Weltzustandes, könnte auch jedes Sub-System, wie in Abbildung 2.15 dargestellt, eine eigene Kopie des Weltzustandes haben. Hierbei müssen diese lokalen Weltzustände durch eine Kommunikationsschnittstelle synchronisiert werden. Kontrollsysteme für Echtzeitsysteme haben ein sehr ähnliches Problem. Hier steuern eine Vielzahl von Computern mehrere Geräte in einer großen technischen Anlage, wie beispielsweise einem Teleskop oder einem Teilchenbeschleuniger. Ein gängiges Modell ist hierbei, die einzelnen Geräte in Strukturen zu abstrahieren, wobei die einzelnen Eigenschaften jedes Geräts als *Prozessvariablen* dargestellt werden. Prozessvariablen können hierbei beobachtet werden, um auf Veränderungen zu reagieren, oder geschrieben werden, um ein Gerät bestimmte Aktionen ausführen zu lassen. Die Kontrollsystemsoftware EPICS verwendet zum Beispiel dieses Konzept (Johnson et al., 2012).

Simulator X (Latoschik und Tramberend, 2011b,a) überträgt dieses Konzept auf VR-Systeme. In Simulator X werden die einzelnen Simulationsobjekte als Entitäten dargestellt, wobei die einzelnen Eigenschaften als *State Variables* dargestellt werden. Eine State Variable enthält hierbei nicht den eigentlichen Wert, sondern stellt eine normierte Schnittstelle zur Verfügung, um lesen, beobachten und schreiben zu können. Beim Lesen eines Wertes wird also eine Nachricht an den Prozess gesendet, der diese Variable verwaltet, der wiederum den aktuellen Wert zurückschickt.

Solch ein System braucht keine ausgefeilte globale Locking-Strategie, verlagert das Problem jedoch in die Komponente. Darüber hinaus existiert kein globaler Weltzustand und keine globale Sicht auf die simulierte Szene mit all ihren Aspekten. Dies eröffnet neue und derzeit unbeantwortete Fragen in Bezug auf die Konsistenz in einem Interaktiven Echtzeitsystem.

2.2.4 Auswertung

In Tabelle 2.2 wird das Ergebnis des Vergleichs dargestellt. Die wenigsten Punkten hat SvS erhalten, was jedoch dem Umstand geschuldet ist, dass es sich um einen nicht öffentlichen Forschungsprototypen handelt. Klassisches Locking hat fünf Punkte erhalten und ist somit auf dem dritten Platz. Mit sieben Punkten ist STM auf dem zweiten Platz, womit Sweeney (2006) in seiner Aussage bestätigt werden kann, dass es sich bei STM um eine interessante Technik für Interaktive Echtzeitsysteme handelt. Die im Kontext dieser Arbeit geeignetste Technologie ist die Verwendung von Prozessvariablen.

Tabelle 2.2: Vergleich von vier Techniken zur Synchronisation in asynchronen Interaktiven Echtzeitsystemen. Es werde die fünf Bewertungsstufen ++, +, 0, - und −− verwendet. Die Bewertung ++ bedeutet, dass ein Modell hier eine besonders positive Eigenschaft hat, die Bewertung −−, dass das Modell hier eine besonders negative Eigenschaft hat.

	Locks	STM	SvS	Prozessvariablen
SyK1: Skalierbarkeit	+	++	++	++
SyK2: Overhead	++	−−	0	0
SyK3: Speicherbedarf	++	+	++	-
SyK4: Ausgereiftheit	++	0	−−	++
SyK5: Entwicklungszeit	−−	++	+	++
SyK6: Zuverlässigkeit	−−	++	++	++
SyK7: Anwendbarkeit	+	++	−−	+
Punkte	5	7	3	8

2.3 Techniken zur Schätzung und Messung in Interaktiven Echtzeitsystemen

In diesem Abschnitt werden Techniken zur Schätzung und Messung in Interaktiven Echtzeitsystemen vorgestellt. In Abschnitt 2.3.1 wird das Thema *Metriken* behandelt. Hierbei werden Anforderungen an Metriken definiert, um die die Nebenläufigkeit, Latenz und Konsistenz quantifiziert darstellen zu können. Die nächsten zwei Abschnitte setzen sich mit den verwendete Techniken auseinander. Das in Abschnitt 2.3.2 beschriebene *Profiling* stellt hierbei den aktuellen Stand der Technik bei der Performance-Analyse von Software im Allgemeinen und Interaktiven Echtzeitsystemen im Speziellen dar. Hierzu wird zunächst allgemein behandelt, wie Profiler Daten sammeln und visualisieren. Anschließend werden noch einmal Profiler für Interaktive Echtzeitsysteme vorgestellt. Als Kontrast zum Profiling wird *Model Checking* als alternativer Ansatz untersucht. In Abschnitt 2.3.3 findet hierfür eine kurze Einführung in das Thema Model Checking statt.

2.3.1 Metriken

Als Metriken werden häufig numerische Werte verstanden, die quantifiziert objektive und reproduzierbare Eigenschaften einer Software wiedergeben (Lincke et al., 2008; Dept, 1998). Hierbei bildet eine einzelne Metrik, einen einzelnen Aspekt einer Software ab. Ein komplettes System wird über einen Satz von Metriken quantifiziert dargestellt. Dieser Satz von Metriken soll Hu und Gorton (1997) zufolge die folgenden drei Eigenschaften erfüllen:

MR1 *Niedrige Schwankungen*: Der numerische Wert einer Metrik soll sich langsam im Verhältnis zum abgebildeten Sachverhalt ändern und keine Sprünge aufweisen.

MR2 *Keine Redundanz*: Mehrere Metriken sollen nicht den gleichen Sachverhalt repräsentieren.

MR3 *Vollständigkeit*: Der Satz von Metriken soll alle Aspekte des Systems beschreiben.

Die Entwicklung von Metriken für asynchrone Interaktive Echtzeitsysteme ist aus zwei Gründen eine wesentliche Herausforderung. Erstens lassen sich laut Hollingsworth et al. (1995) Metriken für sequentielle Systeme nicht ohne Weiteres auf nebenläufige Systeme anwenden. Zweitens haben Interaktive Echtzeitsysteme ihre eigenen Anforderungen (Waldo, 2008), die sich von denen anderer nebenläufiger Systeme unterscheiden (Abdelkhalek und Bilas, 2004).

Ein Beispiel für die Unübertragbarkeit von Metriken für sequentielle Systeme auf parallele Systeme, ist die häufig verwendete Metrik *Frames per Second* (fps). Diese wird in allen Anwendungsbereichen von Interaktiven Echtzeitsystemen zur Leistungsmessung verwendet, wie in Futuremark (2014) oder von Bierbaum (2000) und Deligiannidis (2000), wobei ermittelt wird, wie viele neue Bilder pro Sekunde vom Renderer erzeugt werden. In einem sequentiellen Interaktivem Echtzeitsystem trifft diese Metrik eine Aussage über die Leistung des kompletten Systems, da in ihr auch die benötigte Simulationszeit von

Sub-Systemen enthalten ist, welche dem Renderer vorgelagert sind. In einem asynchronen System hingegen läuft der Renderer u. U. unabhängig davon, ob vorherige Sub-Systeme mit ihrer Arbeit fertig sind. Die fps verlieren hierdurch ihre Aussagekraft.

Ein Beispiel für spezifische Anforderungen eines Interaktiven Echtzeitsystems ist die Metrik *Speedup*. Diese quantifiziert den Zeitgewinn, der durch eine parallelisierte Implementierung eines Algorithmus gewonnen wird. Hierzu werden die gleichen Eingabedaten einmal durch eine sequentielle und eine parallele Implementierung verarbeitet. Der Quotient aus beiden Werten ergibt den Speedup. Diese Metrik ist auf ein Interaktives Echtzeitsystem im Gesamten nicht anwendbar, da die Eingabedaten im Voraus nicht vollständig bekannt sind. Denn neben des dargestellten Assets und der Programmlogik, gehören auch die Eingaben des Benutzers dazu. Über die Eingaben des Benutzers könnten zwar Annahmen getroffen werden, jedoch würden diese Testdaten nur einen kleinen Ausschnitt aus der hohen Anzahl von möglichen Kombinationen von Interaktionen durch den Benutzer darstellen. Darüber hinaus basiert der Speedup darauf, dass ein Programm mit Eingabedaten gestartet wird und sich nach Fertigstellung beendet. Ein Interaktives Echtzeitsystemen läuft hingegen, wie in Abschnitt 2.1.1 dargestellt, in einem Simulation Loop, bis es vom Anwender beendet wird.

Ein Satz von neun Metriken, welche die Anforderungen MR1–MR3 erfüllen, wurden bereits in (Rehfeld et al., 2014) veröffentlicht. Im Rahmen dieser Arbeit sind hieraus die Metriken für Nebenläufigkeit, Latenz und Konsistenz relevant.

2.3.2 Profiling

Profiler werden von Software-Entwicklern verwendet, um ein besseres Verständnis über geschriebene Software zu entwickeln und kritische Abschnitte im Quelltext zu identifizieren (Srivastava und Eustace, 1994). Heute sind Profiler ein Standardwerkzeug in der Software-Entwicklung. So wird z. B. das Java Development Kit standardmäßig mit dem Profiler VisualVM (2014) ausgeliefert und auch Visual Studio enthält standardmäßig Tools zum Profiling. Darüber hinaus verfügen Interaktive Echtzeitsysteme zunehmend über eigene Profiler, zur Analyse von auf ihnen basierenden Anwendungen. Im Folgenden wird zunächst auf die Art, wie Profiler Daten sammeln, eingegangen. Anschließend werden gängige Visualisierungen dieser Daten gezeigt. Abschließend werden einige Profiler für Interaktive Echtzeitsysteme vorgestellt.

Datensammlung

Ein Profiler benötigt zur Analyse des Laufzeitverhaltens einer Anwendung Daten, die während der Ausführung der Anwendung gesammelt wurden. Ilsche et al. (2015) zufolge gibt es hierzu zwei verschiedene Techniken:

- Instrumentierung
- Sampling

Neben diesen beiden Varianten sei hier noch das Abrufen von Hardware-Zählern genannt, was ebenfalls behandelt wird.

Instrumentierung Bei der Instrumentierung wird der Quell- oder Binärcode einer Anwendung verändert, um Daten über das Laufzeitverhalten zu sammeln. Diese Instrumentierung kann manuell oder automatisch erfolgen. Bei der manuellen Instrumentierung wird der Quelltext durch den Entwickler der Anwendung so verändert, dass die Sammlung von Daten erfolgt. Bei der automatischen Instrumentierung erfolgt dies z. B. durch den Compiler. Beispiel für solch eine automatische Instrumentierung ist der GNU C Compiler (GCC) im Zusammenspiel mit gprof (Graham et al., 1982). Wird ein Quelltext mit GCC mit dem Argument `-pg` übersetzt, baut der GCC eine Instrumentierung ein (Stallman und the GCC Developer Community, 2015). Wird die übersetzte Anwendung ausgeführt, schreibt die eingebaute Instrumentierung die gewonnenen Daten auf die Festplatte. Diese Daten können anschließend mit gprof ausgewertet werden. Ein anderer Ansatz wird von *Java* verfolgt, wo die Instrumentierung mittels *Java Agents* erfolgt. Java Agents werden durch die Java Virtual Machine noch vor der eigentlichen Anwendung ausgeführt und erlauben die Registrierung von *Transformatoren*. Ist ein Transformator durch einen Java Agent registriert und wird eine Klasse geladen, wird der Byte Code dieser Klasse an den Transformator übergeben. Dieser kann anschließend Änderungen am Byte Code vornehmen. In diesem Fall findet die Instrumentierung also nicht durch den Compiler statt, sondern durch Java Agents, die durch den Entwickler des Profilers bereitgestellt werden.

Darüber hinaus ist zu unterscheiden, ob die Instrumentierung die Daten erst nach Ende des Durchlaufs der Applikation bereitstellt, oder ein Echtzeit-Profiling erlaubt, während die Applikation läuft. Insbesondere letzteres ist verbreitet bei Profilern für Java wie YourKit (2014) und VisualVM (2014).

Ein Vorteil der Instrumentierung ist, dass beliebig bestimmt werden kann, welche Daten wie gesammelt werden. So kann das Verhalten sehr feingranular gemessen und anschließend analysiert werden. Der wesentliche Nachteil ist der große Overhead, den diese Methode erzeugt.

Sampling Anstatt eine vollständige Instrumentierung durchzuführen und somit jedes Ereignis aufzuzeichnen, kann auch eine zeitliche Abtastung (Sampling) verwendet werden. Hierbei wird der Prozess regelmäßig durch den Profiler kurz angehalten, um den Stack jedes einzelnen Threads auszuwerten. Durch eine niedrigere Abtastfrequenz wird auch die Last der Datensammlung reduziert, wobei jedoch die Ergebnisse eine geringere Genauigkeit haben. Die Genauigkeit nimmt mit höherer Frequenz zu, wobei auch die Last durch die Datensammlung steigt.

Abruf von Hardware-Zählern Eine weitere Technik zum Sammeln von Daten ist der Abruf von Hardware-Zählern. Diese Zähler werden durch CPUs und GPUs bereitgestellt und können durch Bibliotheken des Chip-Herstellers abgerufen werden. Für CPUs von Intel ist dies der Intel Performance Counter Monitor (Willhalm et al., 2012), für GPUs von NVIDIA das NVIDIA PerfKit (NVIDIA, 2014) und für GPUs von AMD die AMD GPUPerfAPI (AMD, 2015).

Die Menge der Zähler, die abgerufen werden können, variiert nach Hersteller und Chip.

So kann beispielsweise über das NVIDIA PerfKit abgerufen werden, wie viel Zeit zwischen zwei Swaps des Framebuffers vergangen ist. Die Bibliotheken erlauben in der Regel eine Abfrage zu Zählern über das Speicherverhalten, z. B. die Häufigkeit von Cache-Misses. Ein wesentlicher Vorteil besteht darin, dass der Abruf von Hardware-Zählern einen geringeren Overhead als eine Instrumentierung oder Sampling hat. Ein wesentlicher Nachteil ist, dass man nur auf die Daten zurückgreifen kann, die durch die Hardware-Zähler geliefert werden.

Mischverfahren Moderne Profiler setzen eine Mischung aus den genannten Verfahren ein, wobei vor der Messung eingestellt werden kann, welche Daten überhaupt erhoben werden. Hierdurch soll eine Verfälschung durch die Messung möglichst minimiert werden. Dies kann genutzt werden, um den gleichen Programmabschnitt immer wieder auszuführen und unter unterschiedlichen Aspekten zu untersuchen. Ein gutes Beispiel hierfür ist der Frame Profiler im AMD GPU PerfStudio. Dieser unterstützt einen Modus, in dem der gleiche Frame mehrfach gerendert wird, wobei bei jedem Durchlauf Daten zu anderen Aspekten des Renderings gesammelt werden.

Darstellung der erhobenen Daten

Die erhobenen Daten sind in ihrer rohen Fassung in der Regel umfangreich und in dieser Form kaum direkt interpretierbar. Die gemessenen Daten müssen somit aufbereitet werden, damit ein Entwickler auf ihrer Basis sinnvolle Entscheidungen zur Optimierung treffen kann. In diesem Abschnitt werden einige der gängigsten Darstellungen erläutert.

Tabellen Die einfachste Darstellung von Messdaten erfolgt durch Tabellen. Ein Beispiel hierfür ist eine Tabelle, die für jede Funktion aufführt, wie viel Zeit mit der Ausführung dieser Funktion verbracht wurde. Darüber hinaus kann die Häufigkeit des Aufrufs angegeben werden. Abbildung 2.16 zeigt solch eine Tabelle aus dem Profiler VisualVM.

Dieses Konzept lässt sich auch auf die GPU übertragen. So verwenden zum Beispiel NVIDIA NSight und AMD GPU PerfStudio Tabellen, um darzustellen, wie viel Zeit für die Ausführung von Shadern benötigt wurde. Tabellen haben den Vorteil, große Mengen an Informationen kompakt darstellen zu können. Der Nachteil ist, dass einige Zusammenhänge, insbesondere zeitlicher Natur, nicht erkennbar sind.

Graphen Im Folgenden werden einige, in Profilern übliche, graphische Darstellungen erläutert.

Timeline Die häufigste Darstellung innerhalb von Profilern ist die Timeline-Darstellung. Hierbei wird, wie in Abbildung 2.17 exemplarisch anhand des Profilers VisualVM gezeigt, die Aktivität über die Zeit dargestellt.

Solch eine Darstellung erlaubt beispielsweise bei der Optimierung eines Rendering-Vorgangs innerhalb eines Multi-Pass-Renderers, die Identifikation von Passes oder Shadern, die viel Zeit benötigt haben. Des Weiteren wird diese Darstellung auch in Profilern

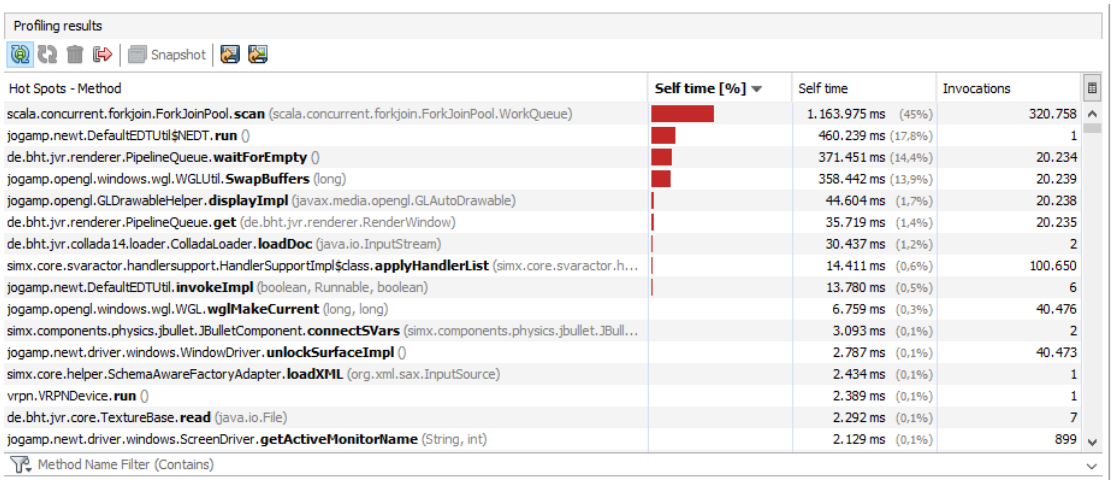


Abbildung 2.16: Darstellung der Ergebnisse des Profilings in Form einer Tabelle im Profiler VisualVM.

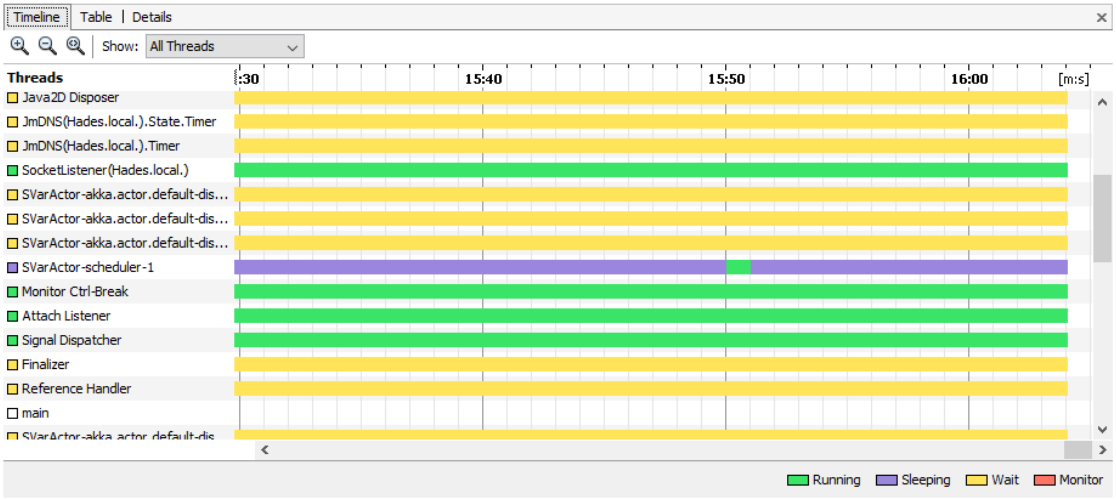


Abbildung 2.17: Darstellung der Aktivitäten einzelner Threads in Form einer Timeline im Profiler VisualVM.

für die GPU verwendet, um die Auslastung der einzelnen Prozessorkerne über die Zeit zu visualisieren. Abbildung 2.18 zeigt hier als Beispiel den Profiler NVIDIA Nsight.

Hardware Topologie Eine weitere Möglichkeit der Visualisierung ist, die Topologie der Hardware darzustellen und dort hervorzuheben, welche Teile der Hardware stark genutzt werden. Ein Beispiel hierfür ist der Profiler von NVIDIA Nsight, der wie in Abbildung 2.19 die Topologie der Grafikkarte darstellt und hier die Belastung der einzelnen Elemente visualisiert.

Eine ähnliche Darstellung wird auch durch AMD PerfStudio verwendet, um einzelne

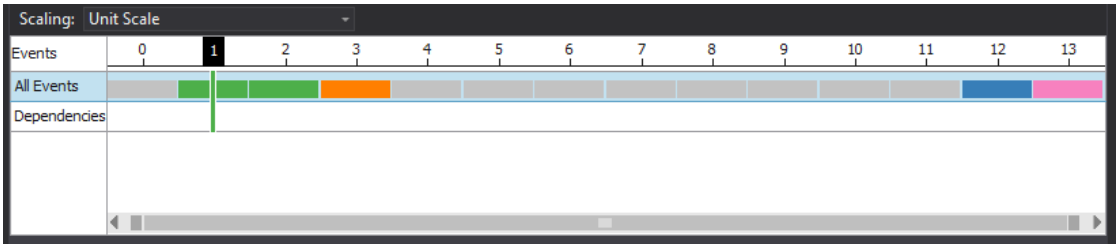


Abbildung 2.18: Darstellung der Aktivitäten auf der GPU im Profiler NVIDIA Nsight.

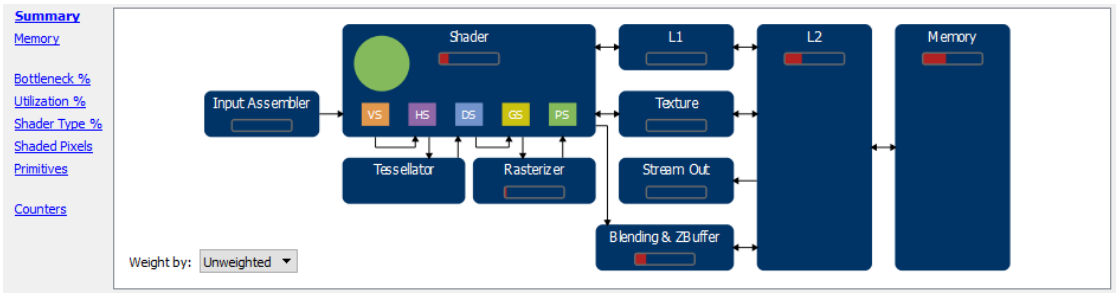


Abbildung 2.19: Darstellung der Hardware-Topologie in NVIDIA Nsight.

Elemente der Rendering-Pipeline darzustellen.

Vorschläge durch Algorithmen zur Analyse Eine weitere Art der Aufbereitung der Daten ist die Erstellung von Vorschlägen zur Optimierung. Hierbei werden die gemessenen Daten nach häufigen Optimierungszielen durchsucht. Eine Beispiel hierfür ist der NVIDIA Visual Profiler, der wie in Abbildung 2.20 gezeigt feststellt, ob nur ein geringer Teil der zur Verfügung stehenden Bandbreite des Speichers genutzt wurde.

Workflow während der Optimierung

Neben der Darstellung der gesammelten Daten ist für die Verwendung eines Profilers auch der Workflow zur Optimierung wichtig. Bei Profilern für C++, wie z. B. gprof oder VTune Amplifier XE (Intel, 2015), ist ein Neukompilieren der Software notwendig. Eine komfortablere Option bieten hier die Frame-Profiler aus NVIDIA Nsight und AMD GPU PerfStudio. Beide Profiler sind fähig, Daten zum Rendering eines einzelnen Frames zu sammeln, wobei dieser Frame später auf Grundlage dieser Daten reproduziert werden kann. Bestandteil dieser Daten sind unter anderem die Quelltexte der verwendeten Shader. Diese lassen sich durch NVIDIA Nsight oder AMD GPU PerfStudio verändern und anschließend kann der Frame mit diesen veränderten Shadern neu gerendert werden. Dies erlaubt kurze Optimierungszyklen bei der Optimierung von Shadern.

Ein weiteres Tool zur schnellen Identifikation von Flaschenhälsen ist das HUD von NVIDIA Nsight, welches in Abbildung 2.21 gezeigt wird. Das Tool kann Applikationen mit einem speziellen HUD versehen, was unter anderem die Frames per Second anzeigt.

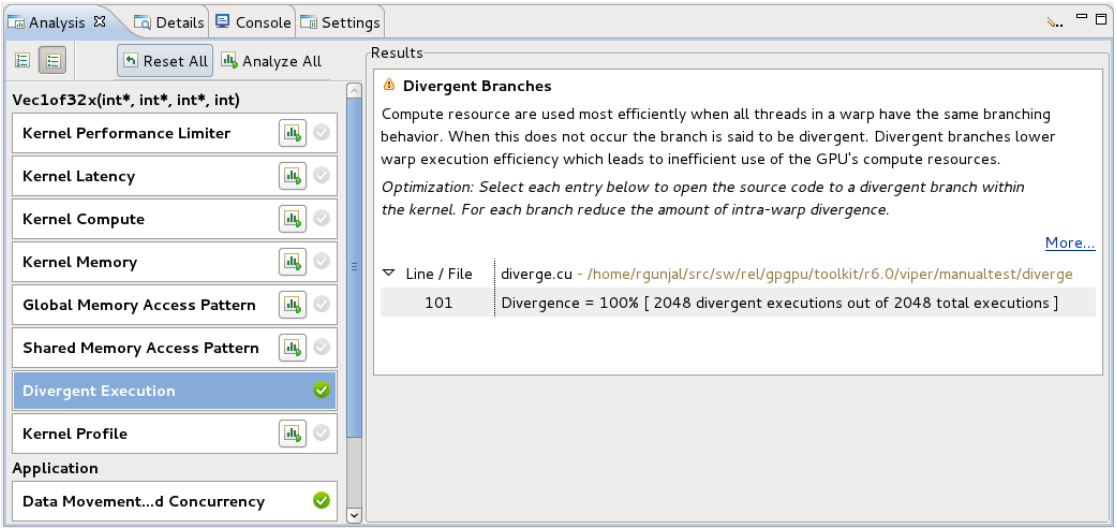


Abbildung 2.20: Durch Visual Profiler generierte Vorschläge zur Optimierung (NVIDIA, 2015).

Darüber hinaus zeigt es den Aufwand einzelner Batches, die sich über das HUD auch abschalten lassen. Die betreffenden Objekte verschwinden hierdurch aus dem gerenderten Bild und gleichzeitig kann die Veränderung der Frames per Second beobachtet werden.

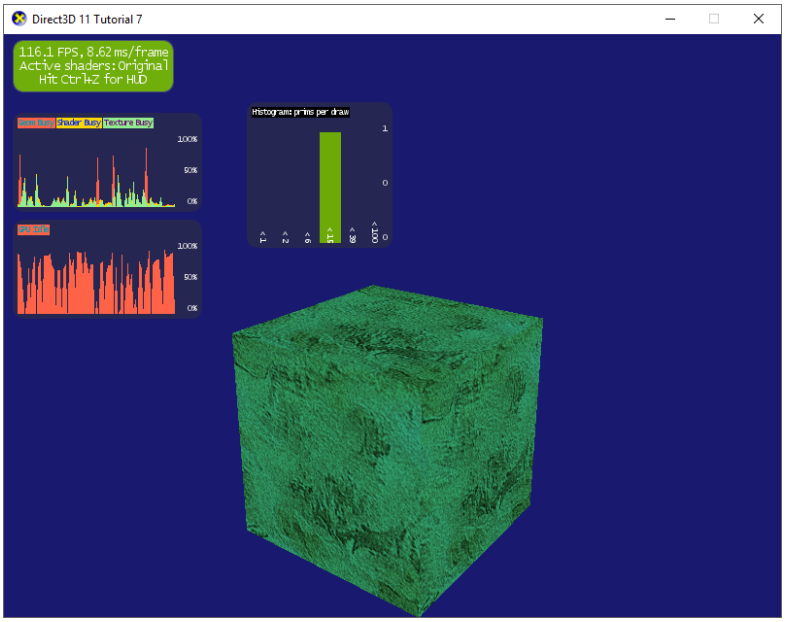


Abbildung 2.21: Anwendung mit HUD von NVIDIA Nsight.

Profiler für Interaktive Echtzeitsysteme

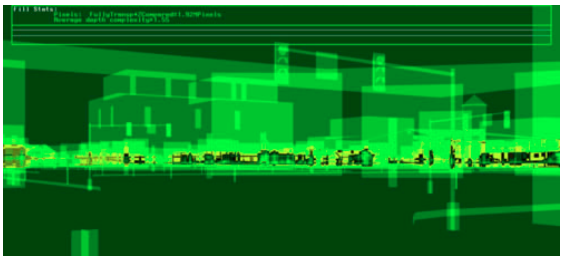
Während der vorherige Teil dieses Abschnitts sich überwiegend mit Profilern im Allgemeinen auseinander gesetzt hat, folgt nun die Betrachtung von drei Profilern für Interaktive Echtzeitsysteme.

Performer IRIS Performer (Rohlf und Helman, 1994) war in den 1990er Jahren ein weit verbreiteter Renderer im VR-Bereich. Performer verfügte bereits über ein eigenes Tool zum Profiling, um laut Rohlf und Helman (1994) folgende Flaschenhälse zu identifizieren:

- Verbrauchte CPU-Leistung der Anwendung
- Transfer von Grafikinformationen auf dem Bus
- Transformation von Geometrien
- Zeichnen der einzelnen Pixel der projizierten Geometrie

Die Abbildung 2.22a zeigt die Ausgabe des Profilers. Hierbei ist auf dem oberen Teil der Abbildung eine Timeline Darstellung zu sehen, wie diese bereits vorher beschrieben wurde. Darüber hinaus werden eine Reihe von Metriken für die einzelnen Phasen von Performer dargestellt.

(a) Profiler von IRIS Performer.



(b) Darstellung des Aufwands pro Pixel in IRIS Performer.

Abbildung 2.22: Der Profiler von Performer entnommen aus (Rohlf und Helman, 1994).

Darüber hinaus erlaubt Performer die Darstellung der Komplexität für jeden einzelnen Pixel, wie in Abbildung 2.22b dargestellt. Je heller ein Pixel in dieser Darstellung ist, umso höher war der Aufwand für diesen Pixel.

Unity Die Game-Engine Unity3D verfügt über einen integrierten Profiler. Dieser kann sowohl zum Untersuchen von Anwendungen im Editor von Unity 3D, als auch von exportierten Anwendungen verwendet werden. Der Profiler zeichnet hierfür Daten über ein vordefiniertes Zeitintervall auf. Die gemessenen Daten werden, wie in Abbildung 2.23 zu sehen ist, durch mehrere Graphen visualisiert, welche die durch einzelne Subsysteme (z. B. Rendering, Physics und Sound) verbrauchte Zeit visualisieren. Darüber hinaus zeigt es den Speicherverbrauch der Anwendung.

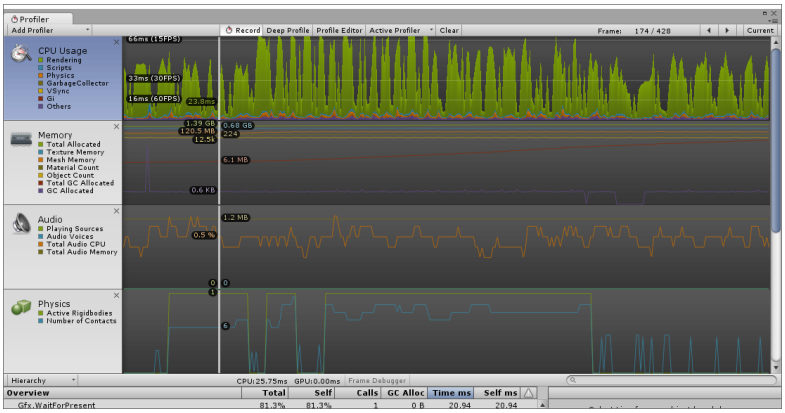


Abbildung 2.23: Visualisierung der Aktivitäten der einzelnen Sub-Systeme der Game Engine Unity 3D durch den integrierten Profiler.

Neben diesen allgemeinen Informationen zur Anwendung kann der Profiler auch Informationen zu jedem einzelnen Frame darstellen. Hierbei zeigt der Profiler dann, wie viel Zeit für jede einzelne Funktion der Sub-Systeme benötigt wurde und wie häufig diese aufgerufen wurde (siehe Abbildung 2.24).

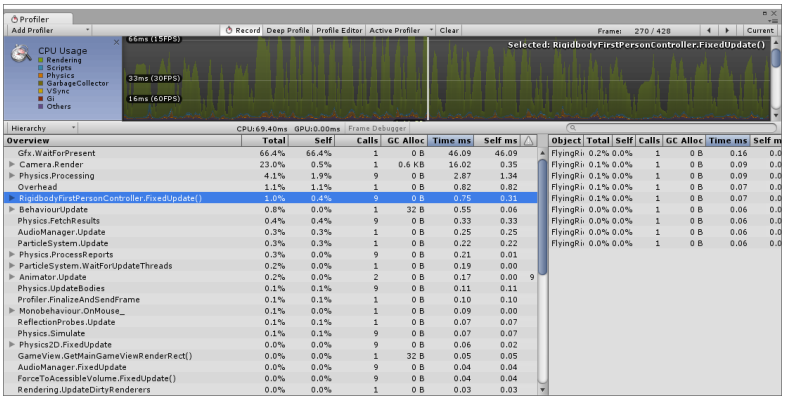


Abbildung 2.24: Auflistung des Zeitbedarfs ausgewählter Funktionen der einzelnen Sub-Systeme.

Als weiterer Modus steht das Deep Profiling zur Verfügung, wobei auch die durch den Anwendungsentwickler erstellten Skripte ins Profiling miteinbezogen werden.

Unreal Auch die Unreal Engine verfügt über einen Profiler. Dieser kann sich sowohl mit einer laufenden Anwendung verbinden, als auch bereits gemessene, in einer Datei gespeicherte Daten anzeigen. In dem Profiler können eine Vielzahl von sehr detaillierten Metriken dargestellt werden. Die Darstellung erfolgt im Data Graph in Form von Graphen einer Metrik über die Zeit.

Des Weiteren kann in einem Event Graphen der Aufruf von Funktionen dargestellt werden. Hierbei wird detailliert aufgeführt, welche Funktion welche anderen Funktionen aufgerufen hat, wie häufig dies geschah und wie viel Zeit in den Funktionen verbracht wurde.

2.3.3 Model Checking

Ben-Ari (2010) folgend existieren zur Überprüfung von Algorithmen und Systemen zwei Techniken:

1. Beweisführung durch mathematische Deduktion
2. Modellprüfung durch einen Model Checker

Bei der Deduktion wird eine mathematische Beweisführung erstellt, die am Ende belegt, ob ein Algorithmus in jedem Fall korrekt ist. Hierfür wird ein Satz von Axiomen und Ableitungen erstellt. Diese Methode wurde maßgeblich durch Tony Hoare geprägt und war von den 1960ern bis in die 1980er die vorherrschende Variante zur Programmverifikation (Clarke et al., 2009). Ist ein Beweis erstellt, gilt ein Algorithmus als korrekt. Die Deduktion hat den Vorteil, dass sie ein abschließendes Ergebnis liefert. Sie ist jedoch bei komplexen nebenläufigen Algorithmen mit erheblichem Aufwand verbunden. Dieser Aufwand wird bei realen Systemen schnell so groß, dass eine Überprüfung durch Deduktion zu aufwendig wird, um sie in der Praxis durchzuführen.

Model Checking entstand in den frühen 1980er Jahren (Clarke und Emerson, 1982; Queille und Sifakis, 1982). Es beruht auf der Beobachtung, dass die meisten nebenläufigen Systeme sich mittels eines endlichen gerichteten Graphen darstellen lassen. Hierbei wird dieser Graph M über die folgenden drei Mengen beschrieben:

- Die endliche Menge S, welche alle möglichen Zustände enthält.
- Die Menge R, welche alle möglichen Zustandsübergänge enthält, somit gilt $R \subseteq S \times S$.
- Die Menge L, welche Fakten zu einzelnen Zuständen enthalten kann.

Ist ein Programm oder Algorithmus mit Hilfe einer *formalen Spezifikationssprache* beschrieben, lässt sich hieraus der Graph M, welcher *State Space* genannt wird, erzeugen. Dieser State Space wiederum wird darauf geprüft, ob er die Bedingung f erfüllt ($M \models f$). Die Eigenschaften f können unter anderem in Linearer Temporaler Logik (LTL) (Pnueli, 1977) definiert werden. Ein Vorteil von LTL liegt darin, dass sich LTL Formeln in Büchi-Automaten (Vardi und Wolper, 1994) übersetzen lassen, welche wiederum in der Lage sind, den State Space M zu überprüfen.

Wird festgestellt, dass M die Eigenschaft f nicht erfüllt, geben Model Checker üblicherweise ein Gegenbeispiel aus, welches wiederum verwendet werden kann, um das Modell zu berichtigen. Gerade die Erstellung eines Gegenbeispieles ist eines der nützlichsten Funktionalitäten eines Model Checkers (Clarke et al., 2009).

Ben-Ari (2010) argumentiert, dass durch Model Checking kein endgültiger Beweis über die Korrektheit eines Algorithmus erzeugt wird. Dies liegt grundlegend daran, dass die Menge der natürlichen oder rationalen Zahlen unendlich groß ist. Ein Rechner ist jedoch durch seinen Speicher immer begrenzt, wodurch eine Deduktion gar nicht notwendig ist, um ein verlässliches Ergebnis zu bekommen. Ein wesentliches Problem des Model Checkings ist, dass der State Space mit der Größe des Problems explosionsartig wächst. Diese *State Space Explosion* war eines der wesentlichen Forschungsthemen in Bezug auf Model Checking in den letzten drei Dekaden (Clarke et al., 2009).

Model Checking ist bereits weit verbreitet bei der Software-Entwicklung von Systemen, die keine Toleranz für Fehler haben. So wurden die Sprachen TLA+ (Lamport, 2002) und PlusCal (Lamport, 2009) erfolgreich bei der Entwicklung einiger Systeme von *Amazon Web Services* verwendet (Newcombe et al., 2015). Ein weiteres Beispiel ist der Mars Rover *Curiosity*, bei dessen Software-Entwicklung die Sprache PROMELA eingesetzt wurde (Holzmann, 2014, 1997).

Die Verwendung eines Model Checkers bedeutet jedoch zusätzlichen Aufwand. Zunächst muss ein Entwickler sich mit den Grundlagen des Model Checkings vertraut machen und eine formale Spezifikationssprache lernen. Darüber hinaus müssen Spezifikationen geschrieben und geprüft werden. Laut Newcombe et al. (2015) brauchten Entwickler bei Amazon Web Services in der Regel 2–3 Wochen zum Erlernen von TLA+ oder PlusCal. Das Schreiben von Spezifikationen dauert „wenige Wochen“ (Newcombe et al., 2015). Lamport (2006a) nutzte Model Checking zur Untersuchung eines Algorithmus (Detlefs et al., 2000), von dem jedoch bekannt war, dass er fehlerhaft ist (Doherty et al., 2004). Lamport brauchte ca. 10 Stunden, um in der Sprache PlusCal eine Spezifikation des Algorithmus zu schreiben.

Obwohl Interaktive Echtzeitsysteme mit ihren hohen Anforderungen an die Hardware, ihren Echtzeitanforderungen sowie den Qualitätsansprüchen der Benutzer prädestiniert für das Model Checking sind, wurde Model Checking bisher nicht im Kontext von Interaktiven Echtzeitsystemen eingesetzt.

2.3.4 Auswertung

Wie in diesem Abschnitt gezeigt wurde, ist Profiling die gängigste Methode der Performance-Analyse in der Software-Entwicklung. Profiler für Interaktive Echtzeitsysteme orientieren sich hierbei stark an Profilern für die generelle Software-Entwicklung. Hier ergibt sich die Frage, ob die Daten, welche spezifisch für ein Interaktives Echtzeitsystem sind, nicht anders dargestellt werden können, um so leichter Aussagen zur Performance einer Anwendung treffen zu können.

Model Checking ist weit verbreitet in der Software-Entwicklung von Systemen, die keine Toleranz für Fehler haben. Die Tatsache, dass Model Checking bei Interaktiven Echtzeitsystemen bisher nicht eingesetzt wurde, überrascht. Insbesondere der Umstand,

dass Leistungssteigerungen in Zukunft nur noch durch Parallelisierung zu erwarten sind, lässt Model Checking zu einer interessanten Technik bei der Entwicklung von Interaktiven Echtzeitsystemen werden.

3 Nebenläufigkeit und Synchronisation

In diesem Abschnitt wird aus den Informationen aus Kapitel 2.1 ein *Synchronisationskonzept* erarbeitet, um das Verhalten eines asynchronen Interaktiven Echtzeitsystems konfigurieren zu können. Ein asynchrones Interaktives Echtzeitsystem bietet ein hohes Maß an Freiheitsgraden in der Konfiguration der Abläufe. Hierdurch lässt sich ein System hin zu bestimmten Verhaltensweisen und Eigenschaften optimieren. Im Kontext dieser Arbeit sind dies:

- Hoher Grad an Parallelität und somit hoher Durchsatz
- Niedrige Latenz
- Gewahrte Konsistenz

Zum Erreichen dieser unterschiedlichen Optimierungsziele werden in Abschnitt 3.1 fünf Schemata erarbeitet. Hieraus werden in Abschnitt 3.2 fünf Grundelemente abgeleitet, aus denen die vorher vorgestellten Schemata zusammengesetzt werden können. Diese Grundelemente stellen das Synchronisationskonzept dar. Darüber hinaus bilden sie im Laufe dieser Arbeit die Grundlage für eine *Domain Specific Language*, die in Kapitel 7.2.1 vorgestellt wird. Des Weiteren werden die hier vorgestellten Schemata in Kapitel 8 im Kontext verschiedener Anwendungen verwendet.

3.1 Schemata

Eine Schemata beschreibt, wie einzelne Sub-Systeme, oder noch feingranularere Teile dieser Sub-Systeme, in ihrer Frequenz begrenzt werden oder in ihrer Ausführung voneinander abhängen.

3.1.1 Schema 1: Sub-Systeme laufen sequentiell

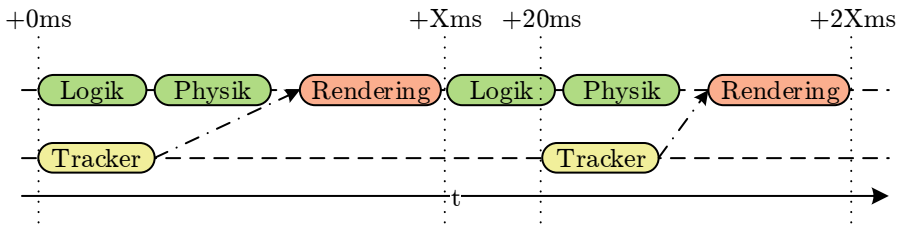


Abbildung 3.1: **Schema 1**: Sub-Systeme laufen sequentiell

Das erste Basisschema ist die Nachbildung eines klassischen Simulation Loops, wodurch alle Sub-Systeme sequentiell ausgeführt werden. Es wird demnach als erstes die Anwendungslogik ausgeführt, anschließend die Physik-Engine und als letztes der Renderer. Die Frequenz der Tracking-Anbindung wird durch das Tracking-System bestimmt, weshalb die Anbindung nicht Teil der Schleife ist. Das Schema wird in Abbildung 3.1 dargestellt.

3.1.2 Schema 2: Alle ohne Begrenzung

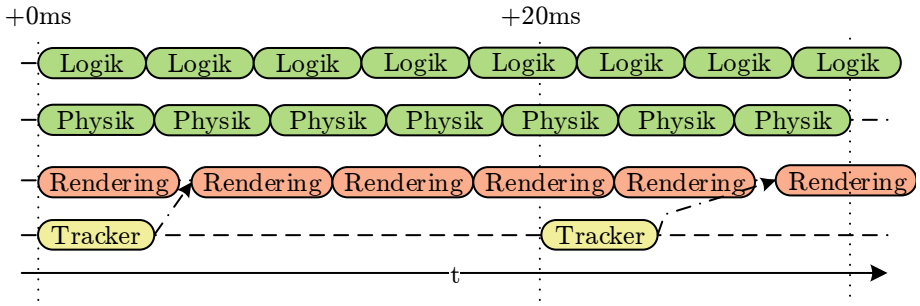


Abbildung 3.2: **Schema 2**: Alle ohne Begrenzung

Im zweiten Basisschema laufen alle Sub-Systeme mit ihrer maximalen Frequenz. Nachdem ein Simulationsschritt ausgeführt wurde, verarbeiten die Sub-Systeme die eingehenden Daten und triggern sich anschließend, wie in Abbildung 3.2 dargestellt, selbst. Solch eine maximale Frequenz ist in der Praxis weder notwendig noch wünschenswert, da sie viel Rechenzeit verbraucht und keine Vorteile für eine Anwendung bietet. Dieses Schema wird hier dennoch eingeführt, um die Ansätze dieser Arbeit an diesem Extremfall zu prüfen.

3.1.3 Schema 3: Jeder Prozess mit einer eigenen Frequenz

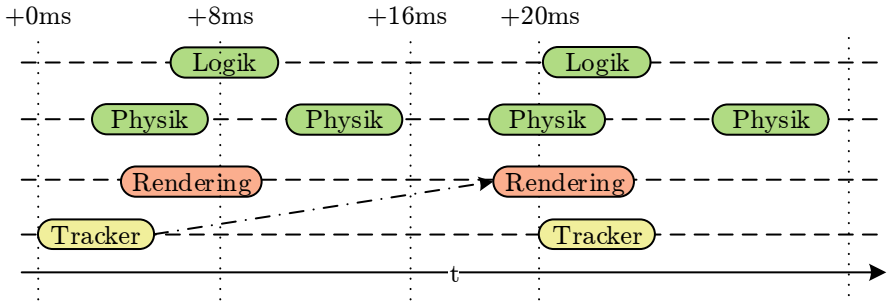


Abbildung 3.3: **Schema 3**: Jeder Prozess mit einer eigenen Frequenz

Aus dem vorherigen Schema wird klar, dass die Begrenzung der Frequenz sinnvoll

ist. Physik-Engines laufen häufig mit 120 Hz, um die Berechnungen numerisch stabil zu halten. Sub-Systeme für die Ausgabe sind in der Regel an die Frequenz des bedienten Kanals gebunden. So läuft ein Renderer in der Regel mit 60 Hz, da die angeschlossenen Projektoren und Monitore diese Frequenz haben. Da ein Benutzer Änderungen nicht schneller mitbekommen kann, als diese vom Monitor oder Projektor angezeigt werden, kann auch die Anwendungslogik auf 60 Hz begrenzt werden. Dieses Schema, welches durch Mönkkönen (2006) beschrieben wurde, ist in Abbildung 3.3 dargestellt.

3.1.4 Schema 4: Alle starten zum gleichen Zeitpunkt

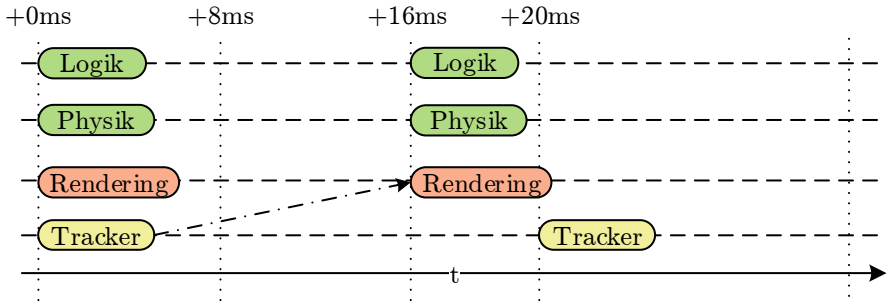


Abbildung 3.4: **Schema 4**: Alle starten zum gleichen Zeitpunkt

Um die Parallelität einer Anwendung zu erhöhen, können alle Sub-Systeme gleichzeitig starten, wie es in Abbildung 3.4 dargestellt wird. Dieses Schema erhöht die Parallelität, fügt jedoch eine Latenz hinzu.

3.1.5 Schema 5: Renderer an den Tracker binden

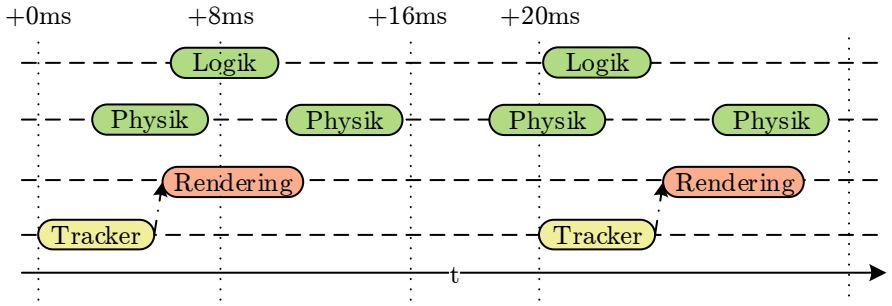


Abbildung 3.5: **Schema 5**: Renderer an den Tracker binden

Insbesondere im Anwendungsbereich VR ist es wünschenswert, eine möglichst niedrige Latenz zwischen Ein- und Ausgabe zu haben. Daher ist es häufig sinnvoll, den Renderer wie in Abbildung 3.5 gezeigt an die Tracking-Anbindung zu koppeln. Eine Latenz

zwischen der Physik und dem Renderer wird vom Nutzer nicht wahrgenommen, wohingegen eine Latenz zwischen Kopfbewegung und Anpassung der Projektionsmatrix sehr wohl durch einen Benutzer wahrgenommen wird. Andere Sub-Systeme laufen in diesem Schema mit ihrer eigenen Frequenz wie in **Schema 3**.

3.2 Grundelemente

Aus den vorher beschriebenen Schemata werden nun fünf Grundelemente des Synchronisationskonzeptes abgleitet. Jedes dieser Grundelemente ist auch immer ein Prozess, der fähig ist, Nachrichten an andere Prozesse zu senden und von diesen zu empfangen.

3.2.1 GE1: Unbewegter Beweger

Der Unbewegte Beweger ist ein Grundelement, welches nur ein einziges mal existiert und zu Beginn der Ausführung einer Anwendung, an eine Menge Prozesse je eine Nachricht versendet. Alle nachfolgenden Aktionen im System passieren immer aufgrund dieser initial versandten Nachrichten.

3.2.2 GE2: Sub-System

Ein Sub-System ist ein Prozess, der Berechnungen durchführt und anschließend die Ergebnisse zu anderen Prozessen kommuniziert oder mit EA-Geräten kommuniziert und diese Daten zur Verfügung stellt. Ein Sub-System kann zum Beispiel ein Renderer, eine Physik-Simulation, ein K.I.-System oder fein- und grobgranulare Teile dieser Sub-Systeme sein. Es repräsentiert die Definition eines Sub-Systems aus Kapitel 2 auf Seite 11 im Synchronisationskonzept für ein asynchrones Interaktives Echtzeitsystem. Ein Sub-System kann somit die folgenden drei Aufgaben durchführen:

1. Aktualisieren des internen Weltzustands auf Grundlage eingehender Nachrichten
2. Modifizieren des internen Weltzustands oder Nutzung von Ein- oder Ausgabe-Kanälen
3. Kommunikation des internen Weltzustands an andere Prozesse

Für diese Aufgaben benötigt ein Sub-System zwei verschiedene Arten von Nachrichten:

MT: Triggert einen für das Sub-System spezifischen Simulationsschritt

MS: Kommuniziert Teile eines neuen Weltzustandes

Ein vollständiger neuer Weltzustand wird von mehreren MT Nachrichten an andere Prozesse kommuniziert. Neben dem Empfang und dem Versand eines neuen Weltzustandes kann ein Sub-System auch andere Prozesse triggern, damit diese Prozesse einen Simulationsschritt durchführt. Die logischen Bedingungen und der Zeitpunkt wann dies erfolgt, sind sehr wichtig. Wir nennen diesen Mechanismus *Synchronisationspunkt*. An

einen Synchronisationspunkt ist eine Bedingung geknüpft, die irgendwann wahr werden kann. Wird diese Bedingung *wahr* wird eine Trigger-Nachricht versendet. Für grobgranulare Sub-Systeme gibt es zwei offensichtliche Synchronisationspunkte:

- *OnStepBegin*, wenn die Berechnungen für einen neuen Simulationsschritt beginnen
- *OnStepEnd*, wenn die Berechnungen für einen neuen Simulationsschritt enden

3.2.3 GE3: Frequenzbegrenzer

Eine Schleife wird in einem Message-Passing basierten System dadurch realisiert, dass sich ein Prozess selbst eine Nachricht schickt. Wie bereits bei der Vorstellung des **Schemas 2** gesagt, ist eine beliebig hohe Frequenz weder wünschenswert noch notwendig, da viele Simulationen nur bis zu einer bestimmten Frequenz stabil sind oder Ein- und Ausgabegeräte selbst mit einer bestimmten Frequenz laufen. Der *Frequenzbegrenzer* begrenzt die Frequenz, mit der ein Prozess sich selbst triggert. Wird also eine Nachricht an ihn gesendet, wird diese solange verzögert, bis so viel Zeit verstrichen ist, dass der Prozess nicht mit einer höheren Frequenz, als der gewünschten läuft.

3.2.4 GE4: Frequenz-Trigger

Ein Frequenzbegrenzer wie im vorherigen Abschnitt geht davon aus, dass ein Prozess sich am Ende selbst triggert. Innerhalb eines Interaktiven Echtzeitsystems gibt es jedoch auch Prozesse, die ständig mit einer eigenen Geschwindigkeit laufen. So laufen zum Beispiel viele Eingabegeräte wie Tracker mit einer eigenen Frequenz. Solch ein *Frequenz-Trigger* schickt nach seiner Aktivierung in einer festen Frequenz Nachrichten an einen anderen Prozess. Auch der Frequenz-Trigger kann unter bestimmten Umständen eine Konsistenz zwischen mehreren Sub-Systemen ermöglichen. Notwendig ist hierfür, dass der Frequenz-Trigger mehrere Sub-Systeme gleichzeitig startet und darüber hinaus, vor dem nächsten Triggern der Sub-Systeme, alle eingehenden Nachrichten über einen neuen Weltzustand verarbeitet wurden. Somit *kann* solch eine Konfiguration für Konsistenz sorgen, garantiert es jedoch *nicht*.

3.2.5 GE5: Barrier

Die vorherigen Elemente des Synchronisationskonzeptes beschäftigten sich mit dem Triggern von Prozessen und der Steuerung der Frequenz von Prozessen. Neben der Steuerung der Frequenz muss jedoch auch die Konsistenz sichergestellt werden. Dies wird durch eine Barrier erreicht. Eine Barrier hat eine Logik, die durch eingehende Nachrichten irgendwann wahr wird. Ist die Bedingung wahr, dann sendet die Barrier eine Nachricht und die Bedingung wird zurückgesetzt.

4 Verwendete Metriken

In diesem Kapitel werden drei Metriken zur Quantifizierung der Nebenläufigkeit, Latenz und Konsistenz beschrieben. Zur eindeutigen Spezifikation der Metriken wird zunächst ein Formalismus eingeführt.

Die hier vorgestellten Metriken erfüllen die in Kapitel 2.3.1 genannten Anforderungen **MR1** und **MR2**, weisen also eine niedrige Schwankung auf und sind nicht redundant. Die drei Metriken erfüllen jedoch nicht die Anforderung **MR3** (Vollständigkeit) in dem Sinne, dass sie alle Eigenschaften einer Software abbilden. Ein vollständiger Satz im Sinne der Anforderungen **MR1**–**MR3** wurden bereits in (Rehfeld et al., 2014) vorgestellt.

Die hier formal spezifizierten Metriken bilden die Grundlage für die Definition von Algorithmen, zu ihrer Berechnung aus typischen Daten wie sie durch Profiling bzw. durch Model Checking gewonnen werden, was in den Kapiteln 5 und 6 erfolgt.

4.1 Formalismus

Der hier verwendete Formalismus ist inspiriert durch (Lamport, 1978), wobei einige Operatoren eine andere Bedeutung haben. Lamport verwendet Kleinbuchstaben für Ereignisse und einfache Pfeile, wenn zwischen diesen Ereignissen ein kausaler Zusammenhang besteht. Der Ausdruck $a \rightarrow b$ bedeutet in Lamports Formalismus, dass Ereignis b kausal auf Ereignis a folgt. Lamports Formalismus ist somit sehr allgemein. Im Rahmen dieser Arbeit liegt der Fokus jedoch auf Message-Passing basierte Systeme, weshalb eine solch allgemeine Betrachtung nicht notwendig ist. Deshalb ist der folgende Formalismus stärker auf Message-Passing ausgelegt.

4.1.1 Prozesse, Nachrichten und Versand

Prozesse im Sinne des Formalismus sind Teile des Programmcodes welche nebenläufig zu anderen ausgeführt werden können. Diese Prozesse kommunizieren untereinander über *Nachrichten*. Nachrichten können hierbei Informationen von einem Prozess zum anderen kommunizieren und Berechnungen auslösen. Der Fall, dass der Prozess A die Nachricht m an Prozess B schickt, wird wie folgt geschrieben:

$$A \xrightarrow{m} B \tag{4.1}$$

Darüber hinaus sei auf Basis von Gleichung 4.1 angenommen, dass der Prozess B die Nachricht m irgendwann verarbeitet, diese also nicht unendlich lang unverarbeitet bleibt. Darüber hinaus sei im Allgemeinen angenommen, dass eine Nachricht nur ein einziges

Mal versandt werden kann. Werden mehrere Nachrichten versandt, wird dies wie gefolgt geschrieben:

$$A \xrightarrow{m} B \Rightarrow A \xrightarrow{n} B \tag{4.2}$$

Die Gleichung 4.2 bedeutet darüber hinaus, dass die Nachricht m vor der Nachricht n durch Prozess A versandt und durch Prozess B verarbeitet wird. Es wird also davon ausgegangen, dass die Reihenfolge von Nachrichten gewahrt bleibt.

Für jeden Prozess existieren darüber hinaus zwei Mengen, welche gesendete und verarbeitete Nachrichten enthalten. Die Menge A_s enthält alle Nachrichten, die von Prozess A versandt wurden. Die Menge A_r enthält alle Nachrichten, die von Prozess A verarbeitet wurden. Der Großbuchstabe entspricht also der Bezeichnung des Prozesses.

4.1.2 Funktionen

Für jede Nachricht existieren drei Zeitstempel:

- $t_s(m)$: Der Zeitpunkt, an dem die Nachricht m versandt wurde.
- $t_b(m)$: Der Zeitpunkt, an dem mit der Verarbeitung der Nachricht m begonnen wurde.
- $t_e(m)$: Der Zeitpunkt, an dem die Verarbeitung der Nachricht m endete.

Typen von Nachrichten werden über griechische Buchstaben ausgedrückt. Der Typ einer Nachricht wird über die Funktion T ermittelt.

$$T(m) = \alpha \tag{4.3}$$

Für einige Metriken müssen nur Nachrichten eines bestimmten Typs betrachtet werden. Die Menge A_r^α enthält alle Nachrichten vom Typ α, welche vom Prozess A empfangen wurden.

$$A_r^\alpha = \{m \in A_r | T(m) = \alpha\} \tag{4.4}$$

Häufig werden Daten durch mehrere Prozesse nacheinander verarbeitet. Solche Pfade von Daten werden geschrieben als $A \xrightarrow{m} B \xrightarrow{n} C \xrightarrow{o} D$. In solchen Fällen ist die Länge der Zeitspanne zwischen dem Senden von Nachricht m und der Verarbeitung von Nachricht o gegeben durch

$$|A \xrightarrow{m} B \xrightarrow{n} C \xrightarrow{o} D| = t_e(o) - t_s(m) \tag{4.5}$$

4.2 M1: Nebenläufigkeit

Um die Nebenläufigkeit zu betrachten, wird zunächst der *Grad der Parallelität* (engl. Degree of Parallelism) ermittelt. Der Grad der Parallelität gibt an, wie viele Prozesse zu einem Zeitpunkt t damit beschäftigt waren, eine Nachricht zu verarbeiten. Zur Berechnung sei Σ_r die Menge aller verarbeiteten Nachrichten.

$$\Sigma_r = A_r \cup B_r \cup \cdots \cup N_r \tag{4.6}$$

Somit ändert sich der Grad der Parallelität, sobald ein Prozess damit beginnt, eine Nachricht zu verarbeiten, oder die Verarbeitung beendet. Folglich muss aus der Menge Σ_r eine Menge Π generiert werden, die aus Tupeln besteht, welche den Beginn und das Ende der Verarbeitung einer Nachricht darstellen. Das Symbol $\nearrow$ bedeutet hierbei den Beginn der Nachrichtenverarbeitung und das Symbol $\searrow$ das Ende der Nachrichtenverarbeitung.

$$\Pi = \{(t, \nearrow) \,|t = t_b\,(m) \wedge m \in \Sigma_r\} \cup \{(t, \searrow) \,|t = t_e\,(m) \wedge m \in \Sigma_r\} \tag{4.7}$$

Es sei nun angenommen, dass über $\Pi\,(i)$ ein geordneter Zugriff auf die Elemente in Π möglich ist, wobei $\Pi_1\,(i)$ der Zeitstempel und $\Pi_2\,(i)$ die Information ist, ob es sich um den Beginn oder das Ende der Nachrichtenverarbeitung handelt. Darüber hinaus sei der Zugriff auf die einzelnen Elemente zeitlich geordnet, womit $\Pi_1\,(i) \leq \Pi_1\,(i+1)$ gilt. Somit kann die Anzahl der zum Zeitpunkt t parallel arbeitenden Prozesse über Gleichung 4.8 ermittelt werden.

$$d(t) = \begin{cases} 0, & \text{wenn } \Pi_1(1) > t \\ e(t, 2, 1), & \text{wenn } \Pi_1(1) \leq t \end{cases} \tag{4.8}$$

Die Gleichung 4.8 benötigt die rekursive Funktion $e(t, i, d)$ in Gleichung 4.9 als Hilfsfunktion.

$$e(t, i, d) = \begin{cases} d, & \text{if } \Pi_1(i) > t \\ e(t, i+1, d+1), & \text{wenn } \Pi_2(i) = \nearrow \\ e(t, i+1, d-1), & \text{wenn } \Pi_2(i) = \searrow \end{cases} \tag{4.9}$$

Neben dem Grad der Parallelität zu einem Zeitpunkt t, ist auch die *durchschnittliche Parallelität* über die gesamte Laufzeit einer Anwendung oder eines bestimmten Zeitraums relevant. Wie in Gleichung 4.10 dargestellt, wird zunächst die Summe des Produkts aus dem Grad der Parallelität und der Zeitspanne, in der dieser Grad der Parallelität herrschte, gebildet. Diese Summe wird geteilt durch die Laufzeit Δt der Anwendung.

$$\frac{\sum_{i=0}^{N-1} \{\Pi_1(i+1) - \Pi_1(i)\} * d(\Pi_1(i+1))}{\Delta t} \tag{4.10}$$

Da insbesondere kleinere Anwendungen ein System nicht vollständig auslasten, wodurch die durchschnittliche Parallelität leicht unter 1 fallen könnte, ergibt es im Kontext eines Interaktiven Echtzeitsystems Sinn, eine *normierte durchschnittliche Parallelität* zu

berechnen, wobei Zeiten, in denen der Grad der Parallelität null ist, aus dem Durchschnitt entfernt werden. Für diese Berechnung sei zunächst die Hilfsfunktion $e(t)$ in Gleichung 4.11 definiert, die 0 ist wenn am Zeitpunkt t der Grad der Parallelität nicht 0 ist, und 1 ist, wenn der Grad der Parallelität an t gleich 0 ist.

$$e(t) = \begin{cases} 1, & \text{wenn } d(t) = 0 \\ 0, & \text{wenn } d(t) \neq 0 \end{cases} \tag{4.11}$$

Mit der Hilfsfunktion $e(t)$ lässt sich nun die Zeitspanne Δt_0 berechnen, die angibt, wie lang ein Grad der Parallelität von 0 bestand.

$$\Delta t_0 = \sum_{i=0}^{N-1} \{\Pi_1(i+1) - \Pi_1(i)\} * e(\Pi_1(i+1)) \tag{4.12}$$

Das Δt_0 kann nun verwendet werden, um Gleichung 4.10 so anzupassen, dass diese die normierte durchschnittliche Parallelität berechnet.

$$\frac{\sum_{i=0}^{N-1} \{\Pi_1(i+1) - \Pi_1(i)\} * d(\Pi_1(i+1))}{\Delta t - \Delta t_0} \tag{4.13}$$

4.3 M2: Latenz

Grundsätzlich entspricht die Latenz der in Gleichung 4.5 dargestellten Zeitspanne. Hierbei ist jedoch zu beachten, dass $|A \xrightarrow{m} B \xrightarrow{n} C \xrightarrow{o} D|$ nicht das gleiche ist wie $|A \xrightarrow{m} B| + |B \xrightarrow{n} C| + |C \xrightarrow{o} D|$. Eine Zerlegung der gesamten Latenz in einzelne Abschnitte ist stark abhängig von der konkreten Implementierung des verwendeten RIS, jedoch für eine detaillierte Analyse unerlässlich.

4.4 M3: Konsistenz

Eine Untersuchung der Konsistenz ermöglicht es, ungewöhnliches Verhalten zu erklären und Fehler zu finden. Hierfür wird die Kommunikation zwischen zwei Prozessen $A \rightarrow B$ untersucht. Es sei angenommen, dass, wenn A einen neuen Weltzustand berechnet hat, dieser Prozess den neuen Weltzustand durch mehrere *update Nachrichten* an Prozess B sendet. Sei die erste Nachricht a und die letzte Nachricht z.

$$A \xrightarrow{a} B \Rightarrow A \xrightarrow{b} B \Rightarrow \cdots \Rightarrow A \xrightarrow{z} B \tag{4.14}$$

Ein Prozess arbeitet in der Regel dann auf Basis eines konsistenten Weltzustands, wenn zwischen der Verarbeitung der Nachrichten a–z, keine Nachricht durch B verarbeitet wurde, die dazu führt, dass der Prozess B einen weiteren Simulationsschritt ausführt. Sei α der Typ der Nachricht, der dazu führt, dass B einen Simulationsschritt ausführt. Dann arbeitet B auf einem konsistenten Weltzustand, wenn folgende Gleichung erfüllt ist.

$$\neg\exists m \in B_r^{\alpha} : t_b(a) < t_b(m) < t_b(z) \tag{4.15}$$

Dies muss für alle von B durchgeführten Simulationsschritte geprüft werden. Hieraus wird ermittelt, in wie viel Prozent der Fälle dies zutrifft.

5 Profiling und Benchmarking

Im vorherigen Kapitel wurden Metriken für die Nebenläufigkeit, Latenz und Konsistenz formal spezifiziert. Der verwendete Formalismus ermöglicht eine kompakte Darstellung, die zugleich jedoch auch sehr abstrakt ist. Als Zwischenschritt zwischen der sehr abstrakten Darstellung in Kapitel 4 und der Beschreibung der Implementierung in Kapitel 7.2.2, werden in diesem Kapitel die *Algorithmen* zur Berechnung der Metriken aus typischen Daten, die ein Profiler erzeugt, vorgestellt.

Für die Darstellung von Algorithmen wird häufig Pseudo-Code verwendet, um die Beschreibung frei von Implementierungsdetails eines konkreten Systems oder einer konkreten Programmiersprache zu halten. Lamport (2009) stellt jedoch fest, dass Pseudo-Code in der Regel nicht eindeutig spezifiziert ist und Autoren dazu neigen, bei Bedarf eigene Konstrukte einzuführen, für die ebenfalls keine eindeutige Spezifikation existiert. Hierdurch sind Beschreibungen von Algorithmen in Pseudo-Code als Spezifikation unbrauchbar, da diese in der Regel mehrdeutig sind.

Zur Lösung dieses Problems entwickelte (Lamport, 2009) die Sprache *PlusCal*. PlusCal ist eine formale Sprache zur Spezifikation von Algorithmen, die entwickelt wurde um Pseudo-Code zu ersetzen. Für PlusCal gibt es zwei verschiedene Syntax-Varianten. Zum einem der p-Syntax, wobei p für *prolix*, also weitschweifig steht, zum anderen der c-Syntax, wobei c für *compact* steht. Laut Lamport (2009) wirkt der c-Syntax für die meisten Programmierer vertrauter. In PlusCal spezifizierte Algorithmen können mittels Model Checking überprüft werden.

In diesem und dem nächsten Kapitel wird PlusCal für die eindeutige Spezifikation von Algorithmen verwendet. Darüber hinaus wird PlusCal in Kapitel 7.2.3 in einem anderen Kontext, nämlich als möglicher Kandidat für die Anbindung eines Models Checkers, evaluiert.

In diesem Kapitel werden zunächst Annahmen über die Struktur der vorliegenden Daten getroffen und in Form von zwei Datenstrukturen spezifiziert. Darüber hinaus werden eine Reihe von Hilfsfunktionen zur Filterung von Sequenzen spezifiziert. Auf dieser Grundlage werden die drei Algorithmen spezifiziert.

Dieses Kapitel ist stilistisch an Lamport (2002) angelehnt. Dies bedeutet, dass sich erklärender Text zu den Algorithmen und ihre Spezifikation in PlusCal im gemeinsamen Fließtext abwechseln. Die vollständigen Spezifikationen sind noch einmal zusammenhängend in Anhang 1 abgebildet.

5.1 Grundlagen

Für die in PlusCal definierten Algorithmen werden sowohl Datenstrukturen für die Repräsentation von Prozessen und Nachrichten benötigt, als auch einige Hilfsfunktionen zur

Filterung von Daten. Beides wird in einem TLA+ Modul mit dem Namen *ProfilingAlgorithmBase* definiert.

MODULE *ProfilingAlgorithmBase*

EXTENDS *Sequences*, *Naturals*, *TLC*

Dieses Modul definiert drei Abhängigkeiten zu anderen Modulen. *Sequences* wird benötigt, da angenommen wird, dass die Informationen über die gesendeten und empfangenen Nachrichten als Sequenz vorliegen. Da die IDs und Zeitstempel von Nachrichten und Prozessen als natürliche Zahlen repräsentiert werden, wird das Modul *Naturals* benötigt. Das Modul *TLC* enthält eine Sortierfunktion, die für einen der Algorithmen benötigt wird.

5.1.1 Eingabedaten

Informationen zu Nachrichten werden durch die Struktur *MessageRecord* dargestellt. Der Typ einer Nachricht ist durch das Feld *type* dargestellt, welches im Formalismus der Funktion $T(m)$ entspricht. Darüber hinaus verweist die Datenstruktur durch die Felder *sender* und *receiver* auf die IDs des Senders und des Empfängers der Nachricht. Die drei Zeitstempel $t_s(m)$, $t_b(m)$ und $t_e(m)$ werden durch die Felder *sent*, *beginProcessing* und *endProcessing* dargestellt.

$$
\begin{array}{ll}
\mathit{MessageRecord} \triangleq [& \\
\quad \mathit{type} : \mathit{Nat}, & \text{Typ der Nachricht} \\
\quad \mathit{sender} : \mathit{Nat}, & \text{Id des Senders} \\
\quad \mathit{receiver} : \mathit{Nat}, & \text{Id des Empfängers} \\
\quad \mathit{sent} : \mathit{Nat}, & \text{Zeitstempel des Versands} \\
\quad \mathit{beginProcessing} : \mathit{Nat}, & \text{Zeitstempel des Beginns der Verarbeitung} \\
\quad \mathit{endProcessing} : \mathit{Nat} & \text{Zeitstempel des Endes der Verarbeitung} \\
]
\end{array}
$$

Die Informationen zu einem Prozess werden über die Struktur *ProcessRecord* dargestellt. Im Formalismus aus Kapitel 4.1 kann jeder Prozess eindeutig durch den verwendeten Großbuchstaben identifiziert werden. In der Datenstruktur erfolgt die eindeutige Identifizierung über das Feld *id* vom *ProcessRecord*. Darüber hinaus hat die Datenstruktur zwei Felder mit den Namen *sent* und *received*. Diese entsprechen den Mengen, welche gesendete (A_s) und empfangene (A_r) Nachrichten beinhalten. Im Gegensatz zum Formalismus handelt es sich hier nicht um ungeordnete Mengen, sondern um geordnete Sequenzen. Hierbei wird angenommen, dass der Inhalt des Feldes *sent* aufsteigend nach dem im *MessageRecord#sent* abgelegten Zeitstempel geordnet ist, während *received* aufsteigend nach dem in *MessageRecord#beginProcessing* abgelegten Zeitstempel sortiert ist.

$$
\begin{array}{ll}
\mathit{ProcessRecord} \triangleq [& \\
\quad \mathit{id} : \mathit{Nat}, & \text{Eindeutige Id jedes Prozesses} \\
\quad \mathit{sent} : \mathit{Seq}(\mathit{MessageRecord}), & \text{Sequenz gesendeter Nachrichten} \\
\quad \mathit{received} : \mathit{Seq}(\mathit{MessageRecord}) & \text{Sequenz empfangener Nachrichten} \\
]
\end{array}
$$

Die Abbildung 5.1 zeigt noch einmal den Zusammenhang zwischen *ProcessRecord* und *MessageRecord*.

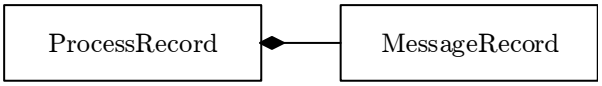


Abbildung 5.1: Die angenommene Datenstruktur für den Profiler.

Die Tabelle 5.1 zeigt noch einmal den Zusammenhang zwischen Elementen des Formalismus aus Kapitel 4.1 und den Strukturen *MessageRecord* und *ProcessRecord*. Hier ist zu beachten, dass die Felder für den Sender und den Empfänger im *MessageRecord* keine Entsprechung im Formalismus haben. Diese sind notwendig zur Spezifikation eines Algorithmus zur Berechnung der Latenz.

Beschreibung	Formalismus	Datenstruktur
Eindeutige Id jedes Prozesses	A	*ProcessRecord#id*
Gesendete Nachrichten	A_s	*ProcessRecord#sent*
Empfangene Nachrichten	A_r	*ProcessRecord#received*
Typ der Nachricht	$T(m)$	*MessageRecord#type*
Id des Senders einer Nachricht	k. E.	*MessageRecord#sender*
Id des Empfängers einer Nachricht	k. E.	*MessageRecord#receiver*
Zeitstempel des Versands	$t_s(m)$	*MessageRecord#sent*
Zeitstempel des Beginns der Verarbeitung	$t_b(m)$	*MessageRecord#beginProcessing*
Zeitstempel des Endes der Verarbeitung	$t_e(m)$	*MessageRecord#endProcessing*

Tabelle 5.1: Zusammenhang zwischen dem Formalismus aus Kapitel 4.1 und den Strukturen *MessageRecord* und *ProcessRecord*. Für einige Elemente gibt es keine Entsprechung (k. E.).

5.1.2 Filterfunktionen

Neben den vorher genannten Strukturen, definiert das Modul *ProfilingAlgorithmBase* Funktionen zur Filterung von Sequenzen von *MessageRecord*s. Diese werden in der Regel verwendet, um die Sequenzen in *ProcessRecord#sent* und *ProcessRecord#received* nach bestimmten Kriterien zu filtern. Die Funktion *FilterByType* filtert eine Sequenz von Nachrichten nach ihrem Typ, wodurch im Ergebnis nur noch *MessageRecord*s eines bestimmten Typs enthalten sind. Die Funktion nimmt zwei Parameter entgegen, die Sequenz *seq*, die gefiltert werden soll, und der Datentyp *type*, nach dem gefiltert werden soll. Zum Filtern der Sequenz wird die Funktion *SelectSeq* aus dem Modul *Sequences* verwendet. Die Funktion *SelectSeq* nimmt als *Parameter* die zu filternde Sequenz und eine Testfunktion mit einem Parameter entgegen. Jedes Element in *seq* wird an diese Testfunktion übergeben. Gibt diese *TRUE* zurück, ist das Element im Ergebnis, andernfalls nicht. Über *LET* wird die Vergleichsfunktion *Test* definiert, die den Parameter v dahingehend überprüft, dass es einen *MessageRecord* darstellt, der eine Nachricht repräsentiert, die dem durch *type* angegeben Typen entspricht. Diese Funktion wird wiederum im *IN*-Teil als Parameter für *SelectSeq* verwendet.

$$FilterByType(seq,\ type) \triangleq \text{LET } Test(v) \triangleq v.type = type \text{ IN } SelectSeq(seq,\ Test)$$

Neben der Filterung durch den Typen, werden drei Funktionen benötigt, um nach dem Zeitstempel zu filtern. Diese sind in ihrer Definition analog zur Funktion *FilterByType*. Die Funktion *FilterBySent* filtert eine Sequenz danach, ob sie an oder nach einem bestimmten Zeitstempel gesendet wurde. Die Funktion *FilterByBeginProcessing* filtert eine Sequenz danach, ob an oder nach einem bestimmten Zeitstempel mit der Verarbeitung der Nachricht begonnen wurde. *FilterBySpan* filtert eine Sequenz danach, ob die Nachrichten innerhalb einer gegebenen Zeitspanne verarbeitet wurden.

$$FilterBySent(seq,\ time) \triangleq \text{LET } Test(v) \triangleq v.sent \geq time \text{ IN } SelectSeq(seq,\ Test)$$

$$FilterByBeginProcessing(seq,\ time) \triangleq \text{LET } Test(v) \triangleq v.beginProcessing \geq time \text{ IN } SelectSeq(seq,\ Test)$$

$$FilterBySpan(seq,\ begin,\ end) \triangleq \text{LET } Test(v) \triangleq v.beginProcessing > begin \land v.beginProcessing < end \text{ IN } SelectSeq(seq,\ Test)$$

Neben der Filterung nach Typ und Zeitstempel, wird auch die Filterung nach dem Sender einer Nachricht benötigt. Hierfür wird angenommen, dass die ID des Prozesses als Parameter r übergeben wird.

$$FilterByReceiver(seq,\ r) \triangleq \text{LET } Test(v) \triangleq v.receiver = r \text{ IN } SelectSeq(seq,\ Test)$$

5.2 Transformation der Daten in Metriken

Auf dieser Grundlage werden nun Algorithmen zur Berechnung der Metriken **M1**–**M3** vorgestellt. Im Rahmen der Spezifikationen der Algorithmen ist es häufig notwendig, über alle Elemente einer Sequenz zu iterieren. PlusCal besitzt jedoch keine zählergesteuerte oder For-Each-Schleife. Stattdessen wird die Sequenz in eine Variable geschrieben und anschließend eine kopfgesteuerte Schleife solange ausgeführt, wie die betreffende Sequenz nicht leer ist. In dieser Schleife wird auf das aktuelle Element durch *Head(var)* zugegriffen. Am Ende der Schleife wird das letzte Element der Sequenz entnommen und die Schleife beginnt erneut.

while $var \neq \langle\rangle$ **do**
  Verarbeitung des aktuellen Elementes durch Head(var)
  $processes := Tail(var)$ **;**
end while ;

5.2.1 M1: Nebenläufigkeit

Der Algorithmus zur Berechnung der Nebenläufigkeit ist mit dem Modul *DegreeOfParallelism* spezifiziert. Um auf die Filterfunktionen und Strukturen für Nachrichten und Prozesse zugreifen zu können, erweitert dieses Modul das Modul *ProfilingAlgorithmBase*. Der Algorithmus benötigt zwei Konstanten, welche die Parameter des Algorithmus darstellen. Dies ist zum einen *SPAN*, welche die Laufzeit der Anwendung beinhaltet und zum anderen *PROCESSES*, welches die Menge aller Prozesse beinhaltet. *PROCESSES* entspricht der Menge $\Sigma = A \cup B \cup \cdots \cup N$ aus dem Formalismus. Im Algorithmus wird davon ausgegangen, dass *PROCESSES* eine Sequenz von *ProcessRecord*s ist. Darüber hinaus wird die Funktion *CompTimeStamp* definiert, die als Vergleichsfunktion bei der Sortierung von Tupeln verwendet wird.

MODULE *DegreeOfParallelism*

EXTENDS *ProfilingAlgorithmBase*
CONSTANTS *PROCESSES*, *SPAN*

ASSUME $PROCESSES \in Seq(ProcessRecord)$
ASSUME $SPAN \in Nat$

$CompTimeStamp(a,\ b) \triangleq a[1] < b[1]$

Der Algorithmus benötigt fünf Variablen zur Berechnung. In der Variable *events* werden zu jeder verarbeiteten Nachricht zwei Tupel angelegt, die jeweils den Beginn und das Ende der Verarbeitung repräsentieren. Der Inhalt entspricht nach dem Durchlauf des Algorithmus dem Ergebnis der Gleichung 4.7. In der Variable *dop* werden zu jedem Zeitpunkt, an dem sich der Grad der Parallelität ändert, der Zeitstempel und der Grad der Parallelität zu diesem Zeitpunkt abgelegt. Die Variable *d* wird während des Laufs des Algorithmus verwendet, um den aktuellen Grad der Parallelität zu speichern. Die Variable *processes* wird verwendet, um über alle Prozesse zu iterieren. Der Wert wird initial auf *PROCESSES* gesetzt. Dies ist notwendig, da *PROCESSES* im Kontext von PlusCal eine Konstante ist, die über die ganze Laufzeit des Algorithmus unverändert bleiben muss. Die Variable *messages* wiederum wird verwendet, um über alle empfangenen Nachrichten eines Prozesses zu iterieren.

--algorithm *DegreeOfParallelism*
variables
$events = \langle\rangle$,
$dop = \langle\rangle$,
$d = 0$,
$processes = PROCESSES$,
$messages = \langle\rangle$ **;**

In einer verschachtelten Schleife wird über alle Nachrichten aller Prozesse iteriert. Für jede dieser Nachrichten wird ein Tupel erstellt, welches den Beginn und das Ende der Verarbeitung repräsentieren. Diese Tupel bestehen aus zwei Elementen. Das erste

Element ist der Zeitstempel, während das zweite Element für die Art des Ereignisses steht. Hier wird *TRUE* verwendet, um den Beginn der Verarbeitung und *FALSE*, um das Ende der Verarbeitung zu markieren. Beide Tupel werden in der Sequenz *events* abgelegt.

```
begin
  while processes ≠ ⟨⟩ do
    messages := Head(processes).received ;
    while messages ≠ ⟨⟩ do
      events := Append(events, ⟨Head(messages).beginProcessing, TRUE⟩) ;
      events := Append(events, ⟨Head(messages).endProcessing, FALSE⟩) ;
      messages := Tail(messages) ;
    end while ;
    processes := Tail(processes) ;
  end while ;
```

Nach dem alle Tupel generiert wurden, werden diese aufsteigend nach ihrem Zeitstempel sortiert. Hierbei wird die Funktion *SortSeq* durch das Modul *TLC* bereitgestellt.

```
  events := SortSeq(events, CompTimeStamp) ;
```

Anschließend wird über alle sortierten Tupel iteriert. Wie bereits vorher beschrieben, wird die Variable d verwendet, um den aktuellen Wert des Grades der Parallelität zu ermitteln. Wenn im aktuellen Tupel der zweite Wert *TRUE* ist, dann wird d um 1 inkrementiert, andernfalls dekrementiert. Anschließend wird der Variable *dop* ein Tuple aus dem aktuellen Zeitstempel und dem Grad der Parallelität hinzugefügt.

```
  while events ≠ ⟨⟩ do
    if  Head(events)[2] then
      d := d + 1 ;
    else
      d := d − 1 ;
    end if ;
    dop := Append(dop, ⟨Head(events)[1], d⟩) ;
    events := Tail(events) ;
  end while ;
end
```

Nach dem Durchlauf des Algorithmus beinhaltet *dop* den Grad der Parallelität für jeden Zeitpunkt, an dem er sich geändert hat.

5.2.2 M2: Latenz

Der Algorithmus zur Berechnung der Latenz befindet sich im Modul *Latency*. Wie auch das vorherige Modul erweitert das Modul *Latency* das Modul *ProfilingAlgorithmBase*. Es benötigt als Eingabe lediglich den Pfad, den die Information, deren Latenz gemessen werden soll, nimmt. Dieser Pfad wird durch die Konstante PATH repräsentiert wird.

MODULE *Latency*

EXTENDS *ProfilingAlgorithmBase*
CONSTANTS *PATH*

In Kapitel 4.3 wurde die Latenzmetrik formal und abstrakt beschrieben. Hierbei wurde gesagt, dass sich die Latenz in eine Kette von einzelnen Abschnitten zerlegen lässt, wobei konkrete Abschnitte in der Regel abhängig von einer Implementierung sind. Die einzelnen Abschnitte seien exemplarisch anhand des folgenden Beispiels entwickelt.

Gegeben sei eine kleine Anwendung, in der die Position des Kopfes des Benutzers durch ein Tracking-System bestimmt wird und die Projektionsmatrix beim Renderer entsprechend angepasst wird. Die Kommunikation mit dem Tracker wird durch einen Prozesse, der Tracking-Anbindung, verwaltet, während ein weiterer Prozess die Umrechnung des Koordinatensystems des Tracking-System in das der Kamera des Renderers durchführt. Das Rendering wird durch einen dritten Prozess bewerkstelligt. Des Weiteren sei angenommen, dass die Tracker-Anbindung selbst Simulationsschritte durchführt, in denen sie die neuen Daten vom Tracker holt. Der Prozess zur Umrechnung arbeitet reaktiv bei neu eintreffenden Daten von der Tracking-Anbindung.

Die innerhalb der Software ermittelbare Latenz ist somit die Zeitspanne zwischen dem Zeitpunkt, an dem die Software frühestens Kenntnis von der Position des Kopfes erhalten kann, und dem Zeitpunkt, an dem das neue Bild für den Benutzer sichtbar ist. Hierbei muss jedoch der kausale Zusammenhang, der über die Umrechnung besteht, beachtet werden. Das heißt, dass wenn man den Beginn eines Simulationsschritts der Tracking-Anbindung findet, man nicht einfach das zeitlich nächste Ende eines Simulationsschritts des Renderers nehmen darf. Es muss sichergestellt werden, dass die Information aus der Tracking-Anbindung den Renderer bereits erreicht hat.

Diese gesamte Latenz des Beispiels kann in folgende Abschnitte zerlegt werden, welche auch noch einmal in Abbildung 5.2 dargestellt sind:

a.) Beginn des Simulationsschritts der Tracking-Anbindung, bis eine Nachricht an den Berechnungsprozess gesendet wird.

b.) Die Nachricht an den Berechnungsprozess wartet auf Verarbeitung.

c.) Die Nachricht beim Berechnungsprozess wird verarbeitet, die Daten für die Kamera werden berechnet und an den Renderer geschickt.

d.) Die Nachricht an den Renderer wartet auf Bearbeitung.

e.) Die Nachricht wurde vom Renderer verarbeitet und der Simulationsschritt wurde beendet.

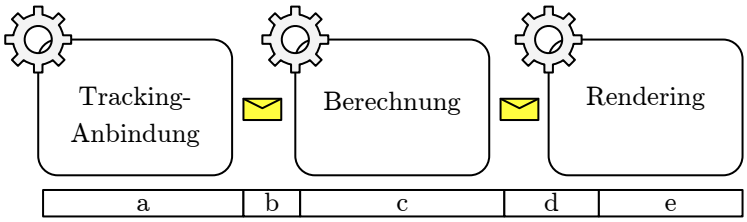


Abbildung 5.2: Abschnitte der Gesamtlatenz des Beispiels.

Durch diese einzelnen Abschnitte lässt sich die gesamte Latenz lückenlos in Abschnitte zerlegen, bzw. der Pfad, den eine Information nimmt, lückenlos beschreiben. In Tabelle 5.2 sind diese Abschnitte noch einmal abstrakter dargestellt. Neben den vorher beschriebenen Abschnitten ist noch ein weiterer Abschnitt notwendig, um alle Fälle abzudecken. Der im Folgenden beschriebene Algorithmus basiert auf den in Tabelle 5.2 gezeigten bzw. im Folgenden definierten Zeitabschnitten:

$BEGIN\_OF\_SIMULATION\_STEP\_TO\_MESSAGE\_IN\_MAILBOX \triangleq 1$
$MESSAGE\_WAITS\_IN\_MAILBOX \triangleq 2$
$BEGIN\_OF\_MESSAGE\_PROCESSING\_TO\_BEGIN\_OF\_SIMULATION\_STEP \triangleq 3$
$BEGIN\_OF\_MESSAGE\_PROCESSING\_TO\_MESSAGE\_IN\_MAILBOX \triangleq 4$
$FINAL\_MESSAGE\_PROCESSED \triangleq 5$
$FINAL\_SIMULATION\_STEP \triangleq 6$

Wie der letzten Spalte von Tabelle 5.2 zu entnehmen ist, kann nicht jeder Abschnitt auf jeden beliebigen anderen Abschnitt folgen. Darüber hinaus existieren zwei Abschnitte, die das Ende eines Pfades darstellen.

Id	Beschreibung der Zeitspanne	Formal	Mögliche Nachfolger
1	Beginn des Simulationsschritts bis eine Nachricht gesendet wurde	$t_s(u) - t_b(s)$	2
2	Nachricht wartet in der Mailbox	$t_b(u) - t_s(u)$	3, 4, oder 5
3	Beginn der Verarbeitung der Update-Nachricht bis die nächste Update-Nachricht versendet wurde	$t_s(u_2) - t_s(u_1)$	2
4	Beginn der Verarbeitung der Update-Nachricht bis zum Beginn des Simulationsschrittes	$t_b(s) - t_b(u)$	1 oder 6
5	Ende der letzten Zeitspanne bis die Update-Nachricht verarbeitet wurde	$t_e(u) - t_b(u)$	—
6	Ende der letzten Zeitspanne bis der Simulationsschritt beendet wurde	$t_e(s) - t_b(s)$	—

Tabelle 5.2: Zeitabschnitte zur Berechnung der Latenz.

Neben den in Tabelle 5.2 definierten Zeitabschnitten, wird auch eine Angabe benötigt, auf welchen Prozess und welche Nachrichtentypen er sich bezieht. Die Struktur *PathNode* beschreibt hierbei einen Abschnitt des Pfades. Hierbei beschreibt das Feld *state*, um welche Zeitspanne es sich handelt und entspricht somit einer der in Tabelle 5.2 definierten Abschnitte. Der Typ der Nachricht wird durch das Feld *messageType* definiert. Das Feld

proc enthält den *ProcessRecord* des betreffenden Prozesses. Es wird angenommen, dass es sich bei *PATH* um eine Sequenz von *PathNode*s handelt.

$$
\begin{array}{l}
PathNode \triangleq [\\
\quad state : Nat, \\
\quad messageType : Nat, \\
\quad proc : ProcessRecord \\
]
\end{array}
$$

ASSUME $PATH \in Seq(PathNode)$

Der Algorithmus zur Bestimmung der Latenz benötigt fünf Variablen. Die Variable *path* wird verwendet, um eine aus der Datenstruktur *PathNode* bestehende Sequenz zu durchlaufen. In der Variable *data* werden die Latenzen der einzelnen Durchläufe gespeichert. Die Variablen *buffer1* und *buffer2* werden durch den Algorithmus verwendet, um Sequenzen von *MessageRecord*s zu durchlaufen. Die Variable *endTime* wird verwendet, um aus einer Prozedur einen Wert zurückgeben zu können.

--algorithm *Latency*
variables
$path = \langle\rangle$,
$data = \langle\rangle$,
$buffer1 = \langle\rangle$,
$buffer2 = \langle\rangle$,
$endTime = 0$ **;**

Zur Verarbeitung der einzelnen Elemente des Pfades bietet sich eine rekursive Definition eines Algorithmus an. Bis auf das erste Element des Pfades, werden alle Elemente durch die rekursive Prozedur *calculateLatency* verarbeitet. Diese hat zwei Parameter. Der Parameter *lPath* enthält den Rest des Pfades, den eine Information gehen muss. Die Variable *start* gibt den Zeitstempel an, wann der letzte Abschnitt endete und ab wann folglich das nächste Ereignis gesucht werden soll. Die zwei Variablen *lBuffer1* und *lBuffer2* entsprechen in ihrer Bedeutung den Variablen *buffer1* und *buffer2*.

procedure *calculateLatency*(*lPath*, *start*)
variable
$lBuffer1 = \langle\rangle$ **;**
$lBuffer2 = \langle\rangle$ **;**

In der folgenden Beschreibung ist mit dem Begriff *aktueller Prozess* immer der Prozess gemeint, der an der aktuellen Stelle des Pfades in der Struktur *PathNode* steht. Mit dem Begriff *aktueller Nachrichtentyp* ist der durch den aktuellen *PathNode* angegebene Nachrichtentyp gemeint.

In der Prozedur *calculateLatency* muss nun einer der sechs definierten Zeitspannen verarbeitet werden. Der erste Fall ist, dass der Zeitabschnitt die Zeit zwischen dem Beginn des Simulationsschritts und dem Versand einer Update-Nachricht an den nächsten

Prozess darstellt. Dies stellt zugleich auch den komplexesten aller Fälle dar. Hierfür wird zunächst ein Simulationsschritt gesucht, der vom aktuellen Prozess nach dem Zeitpunkt in *start* begonnen wurde, um anschließend eine Nachricht zu suchen, die an den aktuellen Prozess des nächsten Pfadelements gesendet wurde und auch dem aktuellen Nachrichtentyp des nächsten Elements entspricht. Hierfür werden zunächst die vom aktuellen Prozess empfangenen Nachrichten nach ihrem Typ gefiltert. Anschließend werden diese nach dem Beginn ihrer Verarbeitung gefiltert, welche nach *start* liegen muss. Im Anschluss werde die vom Prozess gesendeten Nachrichten gefiltert. Hierzu wird zunächst nach Nachrichten gefiltert, die gesendet wurden, nachdem die Verarbeitung der im ersten Schritt gefundenen Nachricht begonnen wurde. Anschließend wird danach gefiltert, dass es sich um Nachrichten handelt, die dem Typ des nächsten Pfadelements entsprechen. Abschließend wird danach gefiltert, dass diese Nachricht an den Prozess des nächsten Pfadelements gesendet wurde. Anschließend wird die Prozedur *calculateLatency* mit dem Rest des Pfades, zum Zeitpunkt des Versands der zweiten Nachricht, aufgerufen.

```
begin
  if Head(lPath).state =
       BEGIN_OF_SIMULATION_STEP_TO_MESSAGE_IN_MAILBOX then
    lBuffer1 := FilterByType(Head(lPath).proc.received, Head(lPath).messageType);
    lBuffer1 := FilterByBeginProcessing(lBuffer1, start);
    lBuffer2 := FilterBySent(Head(lPath).proc.sent, Head(lBuffer1).beginProcessing);
    lBuffer2 := FilterByType(lBuffer2, lPath[2].messageType);
    lBuffer2 := FilterByReceiver(lBuffer2, lPath[2].proc.id)    ;
    call calculateLatency(
      Tail(lPath),
      Head(lBuffer2).sent
    );
    return;
```

Der nächste Abschnitt ist die Zeit, die eine Nachricht auf ihre Verarbeitung wartet. Hierzu wird nach der ersten Nachricht vom aktuellen Nachrichten gesucht, deren Bearbeitung nach dem in *start* stehenden Zeitstempel begann. Hierfür werden die durch den aktuellen Prozess empfangenen Nachrichten zunächst nach ihrem Typ gefiltert, anschließend wird geprüft, dass ihre Verarbeitung nach *start* begann. Die Prozedur *calculateLatency* wird anschließend mit dem Rest des Pfades und dem gefundenen Beginn der Nachrichtenverarbeitung für den Parameter *start* aufgerufen.

```
  elsif Head(lPath).state =
       MESSAGE_WAITS_IN_MAILBOX then
    lBuffer1 := FilterByType(Head(lPath).proc.received, Head(lPath).messageType);
    lBuffer1 := FilterByBeginProcessing(lBuffer1, start);
    call calculateLatency(
      Tail(lPath),
      Head(lBuffer1).beginProcessing
    );
```

Der nächste Fall besteht darin, dass der aktuelle Abschnitt, die Zeitspanne zwischen dem Beginn der Verarbeitung einer Update-Nachricht und dem Beginn des nächsten Simulationsschritts ist. Hierzu wird die erste Nachricht gesucht, die nach dem Zeitstempel in *start* vom aktuellen Prozess verarbeitet wurde und vom aktuellen Nachrichtentyp ist. Hierfür werden die Nachrichten des aktuellen Prozesses nach dem im Pfadelement spezifizierten Typ gefiltert. Im Anschluss werden die bereits gefilterten Nachrichten erneut gefiltert und zwar so, dass deren Verarbeitung nach dem in *start* abgelegten Zeitstempel begann. Anschließend wird *calculateLatency* rekursiv mit dem Rest des Pfades aus *lPath* aufgerufen, exklusive dem aktuellen Knoten, sowie dem Zeitpunkt am dem die Verarbeitung der gefundenen Nachricht begonnen wurde. Wenn die Funktion zurückkehrt, wird auch dieser Durchlauf beendet.

```
elsif Head(lPath).state =
  BEGIN_OF_MESSAGE_PROCESSING_TO_BEGIN_OF_SIMULATION_STEP then
 lBuffer1 := FilterByType(Head(lPath).proc.received, Head(lPath).messageType);
 lBuffer1 := FilterByBeginProcessing(lBuffer1, start);
 call calculateLatency(
   Tail(lPath),
   Head(lBuffer1).beginProcessing
 );
 return;
```

Als Nächstes wird die Berechnung in dem Fall, dass die Zeitspanne, die zwischen dem Beginn der Verarbeitung einer Nachricht und dem Versand der nächsten Update-Nachricht ist, gezeigt. Hierzu wird die erste Nachricht gesucht, die nach dem in *start* angegebenen Zeitstempel vom aktuellen Prozess gesendet wurde und dem aktuellen Nachrichtentyp entspricht. Hierfür werden die vom aktuellen Prozess gesendeten Nachrichten ebenfalls nach ihrem Typ gefiltert. Anschließend erfolgt jedoch eine Filterung nach dem Zeitstempel ihres Versands, der größer sein muss als der in *start*. Im Anschluss wird *calculateLatency* mit dem Rest des Pfades und dem ermittelten Zeitstempel des Versands der Nachricht aufgerufen.

```
elsif Head(lPath).state =
      BEGIN_OF_MESSAGE_PROCESSING_TO_MESSAGE_IN_MAILBOX then
 lBuffer1 := FilterByType(Head(lPath).proc.sent, Head(lPath).messageType);
 lBuffer1 := FilterBySent(lBuffer1, start);
 call calculateLatency(
   Tail(lPath),
   Head(lBuffer1).sent
 );
 return;
```

Wie in Tabelle 5.2 dargestellt, gibt es zwei mögliche Endelemente für einen Pfad. Der Fall *FINAL_MESSAGE_PROCESSED* repräsentiert den Fall, dass die letzte Nachricht eine Update-Nachricht war. Es wird der Zeitpunkt gesucht, wann die Verarbeitung dieser

Nachricht endete. Hierzu werden die Nachrichten zunächst nach ihrem Typ gefiltert. Anschließend nach dem Beginn der Verarbeitung. Die Prozedur *calculateLatency* wird nicht erneut aufgerufen, sondern das ermittelte Ende der Nachrichtenverarbeitung wird in die Variable *endTime* geschrieben und die Rekursion endet.

```
  elsif Head(lPath).state =
        FINAL_MESSAGE_PROCESSED then
   lBuffer1 := FilterByType(Head(lPath).proc.received, Head(lPath).messageType) ;
   lBuffer1 := FilterByBeginProcessing(lBuffer1, start) ;
   endTime := Head(lBuffer1).endProcessing ;
   return ;
```

Der Fall *FINAL_SIMULATION_STEP* repräsentiert den Fall, dass die letzte Nachricht eine Simulationsschrittnachricht war. Es wird der Zeitpunkt gesucht, wann die Verarbeitung dieser Nachricht endete. Hierzu werden die Nachrichten zunächst nach ihrem Typ gefiltert. Anschließend nach dem Beginn der Verarbeitung. Dies ist auch der letzte Fall der Prozedur *calculateLatency*. Auch hier wird der Zeitpunkt des ermittelten Endes in *endTime* geschrieben und die Rekursion endet.

```
  elsif Head(lPath).state =
        FINAL_SIMULATION_STEP then
   lBuffer1 := FilterByType(Head(lPath).proc.received, Head(lPath).messageType) ;
   lBuffer1 := FilterByBeginProcessing(lBuffer1, start) ;
   endTime := Head(lBuffer1).endProcessing ;
   return ;
 end if ;
 return ;
end procedure ;
```

Ein Pfad kann mit zwei verschiedenen Elementen beginnen. Entweder mit dem Beginn einer Simulationsschrittnachricht oder damit, dass eine Nachricht auf ihre Verarbeitung wartet. Der erste Fall entspricht im Wesentlichen bereits der Spezifikation wie in *calculateLatency*. Der Unterschied ist lediglich, dass nicht nach einem bestimmten Anfangszeitpunkt gefiltert wird. Alle gefundenen Simulationsschritte werden in einer Schleife durchlaufen und für jeden wird die Latenz ermittelt.

```
begin
 if Head(PATH).state =
    BEGIN_OF_SIMULATION_STEP_TO_MESSAGE_IN_MAILBOX then
   buffer1 := FilterByType(Head(PATH).proc.received, Head(PATH).messageType) ;
   buffer2 := FilterBySent(Head(PATH).proc.sent, Head(buffer1).beginProcessing) ;
   buffer2 := FilterByType(buffer2, PATH[2].messageType) ;
   buffer2 := FilterByReceiver(buffer2, PATH[2].proc.id)    ;
   while buffer2 ≠ ⟨⟩ do
     call calculateLatency(
       Tail(PATH),
```

```
      Head(buffer2).sent
    );
    data := Append(
      data,
      endTime − Head(buffer1).beginProcessing
    );
  end while ;
```

Auch der zweite Fall unterscheidet sich nur gering von dem in *calculateLatency* spezifizierten. Auch hier wird zwar nach Typ, nicht jedoch nach einem Versandzeitpunkt gefiltert. Auch hier wird für jede Nachricht der vollständige Pfad durchlaufen um die Latenz zu ermitteln.

```
else
  buffer1 := FilterByType(Head(PATH).proc.received, Head(PATH).messageType);
             while buffer1 ≠ ⟨⟩ do
    call calculateLatency(
      Tail(PATH),
      Head(buffer1).beginProcessing
    );
    data := Append(
      data,
      endTime − Head(buffer1).sent
   );
  end while ;
end if ;
end
```

Nach dem Durchlauf des Algorithmus befinden sich in *data* die Latenzen für alle identifizierten Fälle des Pfades.

5.2.3 M3: Konsistenz

Bei der Bestimmung der Konsistenz werden die Simulationsschritte zweier Prozesse und die Kommunikation zwischen diesen untersucht. Hierbei wird davon ausgegangen, dass der eine Prozess (Quelle), während eines Simulationsschritts, Daten in Form mehrerer Nachrichten über den neuen Weltzustand an einen anderen Prozess (Ziel) schickt. Das Ziel wiederum führt irgendwann selbst einen Simulationsschritt durch. Dieser Simulationsschritt ist auf Basis eines konsistenten Zustands in Bezug auf die Quelle durchgeführt worden, wenn das Ziel alle Nachrichten verarbeitet hat, die die Quelle während des eigenen Simulationsschritts gesendet hat und erst danach das Ziel mit dem eigenen Simulationsschritt beginnt, wobei diese noch nicht mit der unvollständigen Verarbeitung von Daten zum nächsten Simulationsschritt der Quelle begonnen hat.

Der Algorithmus benötigt als Eingangsdaten die *ProcessRecord*s der Quelle, als Konstante mit dem Namen *SOURCE*, und des Ziels, unter dem Namen *DESTINATION*, sowie den Typ der Simulationsschrittnachrichten *TRIGGERTYPE* und den Typ der Nachrichten zur Kommunikation des neuen Weltzustandes *UPDATETYPE*. Die Konstante *N* beinhaltet die Länge der Laufzeit der Anwendung.

MODULE *Consistency*

EXTENDS *ProfilingAlgorithmBase*
CONSTANTS *SOURCE*, *DESTINATION*, *TRIGGERTYPE*, *UPDATETYPE*, *N*

ASSUME $SOURCE \in ProcessRecord$
ASSUME $DESTINATION \in ProcessRecord$
ASSUME $TRIGGERTYPE \in Nat$
ASSUME $UPDATETYPE \in Nat$
ASSUME $N \in Nat$

Als Nächstes werden noch die Funktionen *Min* und *Max* benötigt, welche den kleineren bzw. größeren von zwei Werten zurückgeben.

$Min(a,\, b) \triangleq \text{IF } a < b \text{ THEN } a \text{ ELSE } b$
$Max(a,\, b) \triangleq \text{IF } a > b \text{ THEN } a \text{ ELSE } b$

Der Algorithmus benötigt acht Variablen. Während der Bestimmung der Konsistenz werden zwei Puffer (*buffer1* und *buffer2*) benötigt, in dem Sequenzen von *MessageRecord*s abgelegt werden. In der Variable *sourceProcessing* wird eine Sequenz von allen Nachrichten, die bei der Quelle zu einem Simulationsschritt geführt haben, welche also von *SOURCE* verarbeitet wurden und vom Typ *TRIGGERTYPE* sind, gesammelt. In der Variable *updateMessages* werden Nachrichten gesammelt, die von der Quelle ans Ziel gesendet wurden und vom Typ *UPDATETYPE* sind. In der Variable *targetProcessing* wird eine Sequenz von allen Nachrichten, die beim Ziel zu einem Simulationsschritt geführt haben, welche also von *DESTINATION* verarbeitet wurden und vom *TRIGGERTYPE* sind, gesammelt. Die Variablen *min* und *max* werden während der Ermittlung der Anzahl der inkonsistenten Weltzustände verwendet, um die Zeitspanne, in der der Zielprozess die Nachrichten zur Kommunikation des neuen Weltzustandes verarbeitet, zu ermitteln. In der Variable *inconsistenceCounter* wird die Anzahl der der inkonsistenten Weltzustände ermittelt.

--algorithm *Consistency*
variables
$buffer1 = \langle\rangle$,
$buffer2 = \langle\rangle$,
$sourceProcessing = \langle\rangle$,
$updateMessages = \langle\rangle$,
$targetProcessing = \langle\rangle$,
$min = N$,
$max = 0$,
$inconsistenceCounter = 0$;

Die Variable *sourceProcessing* ist der nach Datentyp *TRIGGERTYPE* gefilterte Inhalt der durch die Quelle empfangenen Nachrichten. Für die Variable *updateMessages* werden die durch die Quelle gesendeten Nachrichten einmal nach dem Typ *UPDATETYPE* und nach dem Ziel, welches die *DESTINATION* sein muss, gefiltert. Die Variable *targetProcessing* ist der nach Datentyp *TRIGGERTYPE* gefilterte Inhalt der durch das Ziel empfangenen Nachrichten.

begin
sourceProcessing := *FilterByType*(*SOURCE.received*, *TRIGGERTYPE*) ;
updateMessages := *FilterByReceiver*(*SOURCE.sent*, *DESTINATION.id*) ;
updateMessages := *FilterByType*(*updateMessages*, *UPDATETYPE*) ;
targetProcessing := *FilterByType*(*DESTINATION.received*, *TRIGGERTYPE*) ;

Im Folgenden wird aus den Daten in *sourceProcessing*, *updateMessages* und *targetProcessing* ermittelt, wie häufig der Zielprozess einen Simulationsschritt auf Basis eines inkonsistenten Weltzustandes durchgeführt hat. Zunächst wird der Variablen *buffer1* der Wert von *sourceProcessing* zugewiesen. Hierdurch repräsentiert jeder Eintrag in *buffer1* einen Simulationsschritt, der vom Quellprozess durchgeführt wurde. Für jeden dieser Simulationsschritte wird nun ermittelt, welche Nachrichten während der Durchführung dieses Schrittes an den Zielprozess gesendet wurden. Hierfür wird der Variablen *buffer2* der Inhalt von *updateMessages* zugewiesen. Die Variable *min* wird auf die maximale Laufzeit der Anwendung N und die Variable *max* auf 0 gesetzt. In diesen Variablen wird ermittelt, in welchem Zeitraum der Zielprozess mit der Verarbeitung des neuen Weltzustandes beschäftigt war, der im aktuellen Simulationsschritt erzeugt wurde. Hierfür wird für jeden Eintrag in *buffer2* ermittelt, ob die Nachricht gesendet wurde, während durch den Quellprozess die Nachricht aus *buffer1* verarbeitet wurde. Ist dies der Fall, wird die Variable *min* auf den Beginn der Verarbeitung der Nachricht gesetzt, falls dieser Wert kleiner ist als der aktuelle Wert. Die Variable *max* wird auf den Zeitpunkt des Endes der Verarbeitung der Nachricht gesetzt, falls dieser Wert größer ist als der aktuelle Wert von *max*. Wurde die Verarbeitung aller Einträge in *buffer2* abgeschlossen, liegt ein inkonsistenter Weltzustand vor, wenn in der Sequenz *targetProcessing* ein Eintrag für eine Nachricht existiert, die im Zeitraum zwischen *min* und *max* verarbeitet wurde.

*buffer*1 := *sourceProcessing* ;
while *buffer*1 $\neq \langle\rangle$ **do**
  *buffer*2 := *updateMessages* ;
  min := N ;
  max := 0 ;
  while *buffer*2 $\neq \langle\rangle$ **do**
    if *Head*(*buffer*2).*sent* > *Head*(*buffer*1).*beginProcessing*
      $\wedge$ *Head*(*buffer*2).*sent* < *Head*(*buffer*1).*endProcessing* **then**
      min := *Min*(*min*, *Head*(*buffer*2).*beginProcessing*) ;
      max := *Max*(*max*, *Head*(*buffer*2).*endProcessing*) ;
    end if ;
    *buffer*2 := *Tail*(*buffer*2) ;

```
    end while ;
    if FilterBySpan(targetProcessing, min, max) ≠ ⟨⟩ then
      inconsistenceCounter := inconsistenceCounter + 1 ;
    end if ;
    buffer1 := Tail(buffer1) ;
  end while ;
end
```

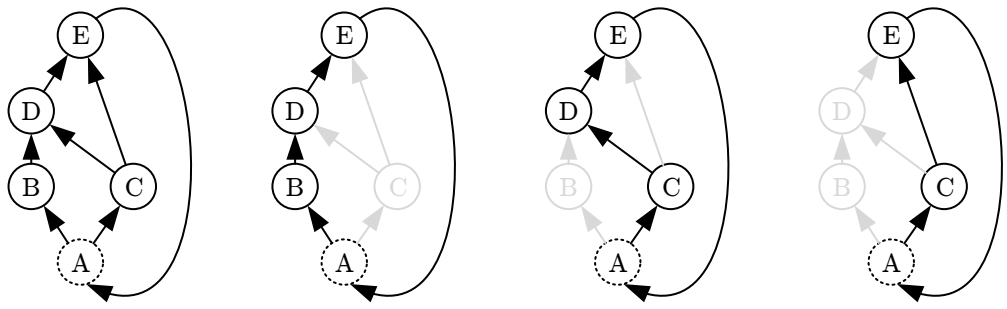
5.3 Verwendung für weitere Metriken

Die in diesem Kapitel beschriebenen Strukturen und Hilfsfunktionen können auch zur Spezifikation weiterer Metriken verwendetet werden. So sind in (Rehfeld et al., 2014) sechs weitere Metriken definiert. Diese benötigen für die Berechnung keine anderen Daten als die in *ProfilingAlgorithmBase* definierten Strukturen. Des Weiteren beschreiben die Strukturen allgemeine Daten aus einem Message-Passing-basierten System, wodurch auch Metriken außerhalb des Anwendungsbereichs der Interaktiven Echtzeitsystemen auf ihrer Basis und im gleichen Stil definiert werden können.

6 Model Checking

Zur Beantwortung der vierten Forschungsfrage dieser Arbeit, müssen die Metriken **M1**–**M3** auch aus den Daten, die ein Model Checker generiert, berechnet werden. Wie in Kapitel 2.3.3 beschrieben, erzeugt ein Model Checker einen gerichteten Graphen, welcher die möglichen Ausführungen eines spezifizierten Algorithmus oder Systems darstellt. Während also die Datenstrukturen aus dem vorherigen Kapitel einen möglichen Pfad darstellen, beinhaltet der durch den Model Checker erzeugte Graph alle Pfade. Solch ein Pfad durch den Graphen sei hier *Szenario* genannt.

Als Beispiel sei hier die Abbildung 6.1 angeführt. In Abbildung 6.1a ist eine State Space mit fünf Zuständen und sieben Kanten dargestellt. Dieser bildet in jedem Fall eine Schleife. Aus diesem State Space lassen sich drei Szenarien erzeugen.



(a) Ein kleiner State Space als Beispiel.
(b) Erstes Szenario (ABDE...).
(c) Zweites Szenario (ACDE...).
(d) Drittes Szenario (ACE...).

Abbildung 6.1: Ein kleiner State Space als Beispiel für die Szenarien. Aus dem State Space in Abbildung 6.1a lassen sich die in Abbildung 6.1b, 6.1c und 6.1d dargestellten drei Szenarien erzeugen.

Um also für die Spezifikation eines Interaktiven Echtzeitsystems die Metriken **M1**–**M3** zu berechnen, müssen diese für jedes Szenario berechnet werden. Eine neue Spezifikation für die Berechnung der Metriken ist jedoch nicht notwendig. Stattdessen können aus jedem Szenario *ProcessRecord*s und *MessageRecord*s, wie sie im vorherigen Kapitel definiert sind, erstellt werden. Auf dieser Grundlage können auch die im letzten Kapitel spezifizierten Algorithmen verwendet werden.

In diesem Kapitel wird folglich ein Algorithmus beschrieben, um aus einem Graphen, der einen State Space repräsentiert, die Szenarios zu ermitteln und für jedes Szenario Daten zu erzeugen, wie sie auch beim Profiling entstehen. Hierzu wird wie in Kapitel 5

die Sprache PlusCal verwendet.

6.1 Annahmen

Vor der Spezifikation des Algorithmus müssen einige Annahmen getroffen werden, welche Informationen in welcher Form im State Space enthalten sind.

6.1.1 Granularität

Zunächst einmal muss die *Granularität* der Informationen angenommen werden. So könnte bei einer sehr feingranularen Darstellung, eine Kante einen einzelnen Befehl, wie eine einzelne Zuweisung, eine Verzweigung oder ähnliches, repräsentieren. Bei der Verwendung einer imperativen Spezifikationssprache wäre dies beispielsweise eine Zeile mit einer Anweisung. Solch eine feingranulare Darstellung ist für den hier beschriebenen Zweck gar nicht notwendig. Wie bereits im vorherigen Kapitel beschrieben, sind lediglich Informationen über die Prozesse sowie versandte und verarbeitete Nachrichten notwendig. Somit ist es ausreichend, wenn eine Kante die Verarbeitung einer Nachricht durch einen Prozess repräsentiert.

6.1.2 Repräsentation der Zeit

Bei Interaktiven Echtzeitsystemen ist die Einhaltung zeitlicher Anforderungen wichtig. Folglich muss auch die Zeit im State Space repräsentiert werden. Khamespanah et al. (2012) zufolge gibt es hier zwei Ansätze, um die Zeit zu repräsentieren. In einem *time transition system* wird die aktuelle Zeit in jedem ermittelten State durch einen Wert repräsentiert, der in der Regel *now* oder *clock* genannt wird. In einem *floating time transition system* hingegen wird für jeden Prozess solch eine Variable geführt. Somit wird keine für alle Prozesse gültige Zeit benötigt, sondern die Uhr jedes Prozesses wird angepasst, wenn dieser eine Nachricht verarbeitet. Für den Zweck dieser Arbeit ist ein *floating time transition system* einfacher zu verwenden, weshalb davon ausgegangen wird, dass die Daten in diesem Format vorliegen.

6.2 Transformation der Daten

Der Algorithmus zur Ermittlung aller Szenarien eines State Spaces und zur Erzeugung von Daten, wie aus dem Profiling, für jedes dieser Szenarien ist im Modul *StateSpaceToRecords* definiert. Dieses Modul bindet das Modul *ProfilingAlgorithmBase* ein, welches im vorherigen Kapitel spezifiziert wurde.

MODULE *StateSpaceToRecords*

EXTENDS *ProfilingAlgorithmBase*

Der Algorithmus benötigt als Eingabe drei Konstanten. Die Konstante *START* beinhaltet den initialen Zustand. Alle möglichen Zustände des State Spaces sind als Sequenz

in der Konstante *STATES* abgelegt. *TRANSITIONS* beinhaltet alle Kanten des Graphen.

CONSTANTS *START*, *STATES*, *TRANSITIONS*

Im Folgenden wird nun die detaillierte Datenstruktur des State Spaces beschrieben. Nachrichten werden durch die Struktur *Message* repräsentiert. Jede Nachricht die von einem Prozess versandt wurde, hat eine eindeutige ID und einen eindeutigen Typ. Darüber hinaus besitzt sie einen Zeitstempel, wann sie versandt wurde, und die ID des sendenden Prozesses.

$$
\begin{array}{l}
Message \triangleq [\\
\quad id : Nat, \\
\quad type : Nat, \\
\quad sent : Nat, \\
\quad sender : Nat \\
]
\end{array}
$$

Die einzelnen Prozesse werden in Zuständen durch die Struktur *ProcessInfo* repräsentiert. Auch jeder Prozess besitzt einen eindeutige ID. Da wie vorab beschrieben von einem *floating time transition system* ausgegangen wird, wird für jeden Prozess auch eine eigene Uhr geführt. Darüber hinaus hat jeder Prozess auch eine Warteschlange von eingehenden Nachrichten.

$$
\begin{array}{l}
ProcessInfo \triangleq [\\
\quad id : Nat, \\
\quad clock \ : Nat, \\
\quad queue : Seq(Message) \\
]
\end{array}
$$

Ein Zustand wird durch die Struktur *State* repräsentiert. Diese sind somit die Knoten innerhalb des State Spaces. Jeder Zustand hat eine eindeutige ID und eine Liste von Informationen über die Prozesse.

$$
\begin{array}{l}
State \triangleq [\\
\quad id : Nat, \\
\quad processesInfo : Seq(ProcessInfo) \\
]
\end{array}
$$

Übergänge zwischen den einzelnen Zuständen werden durch Strukturen des Typs *Transition* dargestellt. Solch eine Kante ist immer gerichtet und beschreibt den Übergang eines Zustands in einen anderen. Die ID des ersten Zustandes wird im Feld *source* abgelegt, die des Zielzustandes im Feld *destination*. Darüber hinaus ist im Feld *processId* vermerkt, welcher Prozess eine Nachricht verarbeitet hat und im Feld *messageId*, welche Nachricht dies war. Das Feld *shift* beschreibt bei einer Schleife, wie viel Zeit auf die Uhren im Zielzustand aufaddiert werden müssen.

$$
\begin{array}{l}
Transition \triangleq [\\
\quad source : Nat, \\
\quad destination : Nat, \\
\quad processId \ \ : Nat, \\
\quad messageId : Nat, \\
\quad shift : Nat \\
]
\end{array}
$$

Die Abbildung 6.2 verbildlicht noch einmal den Zusammenhang zwischen *Message*, *ProcessInfo*, *State* und *Transition*.

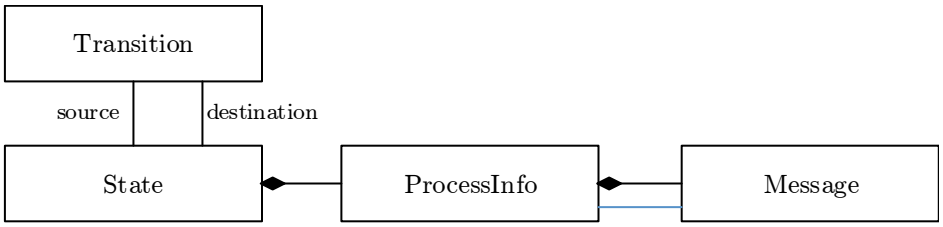


Abbildung 6.2: Die angenommene Datenstruktur aus dem Model Checker.

Vergleicht man diese Strukturen mit der Definition eines State Spaces aus Kapitel 2.3.3, repräsentieren Elemente des Typs *State* die Elemente der Menge S, Elemente des Typs *Transition* die Elemente der Menge T und Elemente der Typen *ProcessInfo* und *Message* den Inhalt der Menge L.

Auf Grundlage der nun definierten Datenstrukturen können auch die Annahmen zum Inhal der vorher deklarierten Konstanten getroffen werden.

$$
\begin{array}{l}
\text{ASSUME } START \in State \\
\text{ASSUME } STATES \in Seq(State) \\
\text{ASSUME } TRANSITIONS \in Seq(Transition)
\end{array}
$$

Der Algorithmus benötigt eine Reihe von Hilfsfunktionen für die Berechnung. Die erste Hilfsfunktion mit dem Namen *Reachable* ermittelt für die ID eines Zustands die ausgehenden Kanten, also somit indirekt die erreichbaren Zustände. Hierzu wird wie bereits im letzten Kapitel die Funktion *SelectSeq* zusammen mit einer Testfunktion verwendet.

$$
\begin{array}{l}
Reachable(source) \triangleq \text{LET } Test(v) \triangleq v.source = source \\
\qquad\qquad\qquad\qquad\ \ \text{IN } \ SelectSeq(TRANSITIONS,\ Test)
\end{array}
$$

Das TLA+ Modul *Sequences* bietet über die Funktion *Head* die Möglichkeit, das erste Element einer Sequenz zu erhalten. Analog hierzu wird die Funktion *Last* definiert, die das letzte Element einer Sequenz zurückgibt.

$$Last(s) \triangleq s[Len(s)]$$

Bei der Ermittlung der einzelnen Szenarien ist es notwendig herauszufinden, ob ein bestimmter Zustand innerhalb dieses Szenarios bereits zugänglich ist. Ist dies nämlich

der Fall, so wurde eine Schleife gefunden und das Szenario ist vollständig. Hierzu wird geschaut, ob es in der übergebenen Sequenz eine *Transition* gibt, die zu dem Zustand mit der übergebenen ID führt.

$$ContainsDestination(seq,\ id) \triangleq \exists\, n \in 1\,..\,Len(seq) : seq[n].destination = id$$

Darüber hinaus muss ermittelt werden, ob ein Element nicht in einer Sequenz ist. Hierzu dient die Funktion *NotContains*.

$$NotContains(seq,\ e) \triangleq \neg\exists\, n \in 1\,..\,Len(seq) : seq[n] = e$$

Eine weitere Funktion ist die Ermittlung des Indexes eines Elements einer Sequenz, wobei die ID des Elements als Suchkriterium verwendet wird.

$$IndexOf(seq,\ id) \triangleq \text{CHOOSE } n \in 1\,..\,Len(seq) : seq[n].id = id$$

Darüber hinaus wird noch eine Funktion benötigt, die einem aus einer Sequenz das Element mit der entsprechenden ID zurückgibt.

$$ElementById(seq,\ id) \triangleq seq[IndexOf(seq,\ id)]$$

Als letztes wird eine Funktion definiert, die die Uhr eines Prozesses von einem bestimmten Zustand zurückgibt.

$$\begin{array}{l} ClockOfProcessInState(processId,\ stateId) \triangleq \\ \quad ElementById(\\ \quad\quad ElementById(STATES,\ stateId).processesInfo, \\ \quad\quad processId \\ \quad).clock \end{array}$$

Der Algorithmus benötigt zur Umformung eines State Spaces, in Daten wie sie ein Profiler erzeugt, zehn Variablen. Die Variablen *buffer1* und *buffer2* werden zum Durchlaufen von Sequenzen und zum Aufbau temporärer Datenstrukturen an verschiedenen Stellen des Algorithmus benötigt. Die Variable *current* wird während der Ermittlung der einzelnen Szenarios benötigt. Die Variable *scenarios* beinhaltet während des Durchlaufs des Algorithmus alle Szenarien als Sequenz von *Transition*s, während die Variable *scenario* während der Erzeugung von *MessageRecord*s und *ProcessRecord*s verwendet wird. In *messageIds* und *processIds* werden die Ids von gefundenen Nachrichten und Prozessen abgelegt. Die erzeugten *ProcessRecord*s werden in der Variable *processRecords* als Sequenz abgelegt, während die erzeugten *MessageRecord*s in der Variable *messageRecords* abgelegt werden. Das Ergebnis wird in der Variable *results* abgelegt, die am Ende eine Sequenz beinhaltet, wobei jedes Element dieser Sequenz für ein Szenario steht und eine Sequenz der *ProcessRecord*s dieses Szenarios ist.

--algorithm *ProfilingAlgorithmBase*

variables
$buffer1 = \langle\rangle$,

$buffer2 = \langle\rangle$,
$current = \langle\rangle$,
$scenario = \langle\rangle$,
$scenarios = \langle\rangle$,
$processIds = \langle\rangle$,
$messageIds = \langle\rangle$,
$processRecords = \langle\rangle$,
$messageRecords = \langle\rangle$,
$results = \langle\rangle$;

Im ersten Schritt müssen alle Szenarien aus dem State Space ermittelt werden. Zur Erinnerung: Ein Szenario ist ein möglicher Weg durch den State Space, der zumindest im Teil eine Schleife bildet. Hier werden zunächst die vom Startzustand abgehenden Kanten ermittelt und in der Variable *buffer2* gespeichert. Im Anschluss wird für jedes Element in *buffer2* eine Sequenz mit diesem Element als einzigen Eintrag erzeugt und in der Variable *buffer1* abgelegt. Die Variable *buffer1* beinhaltet nach diesem Schritt also eine Sequenz von Sequenzen, wobei die inneren Sequenzen je eine *Transition* als Element haben.

begin
$buffer2 := Reachable(START.id)$;
while $buffer2 \neq \langle\rangle$ **do**
$buffer1 := Append(buffer1, \langle Head(buffer2)\rangle)$;
$buffer2 := Tail(buffer2)$;
end while ;

Auf dieser Grundlage werden nun alle Szenarien ermittelt. Hierzu werden die Elemente in *buffer1* durchlaufen. Da der aktuellen Sequenz weitere Elemente hinzugefügt werden müssen, wird der Wert in der Variable *current* gespeichert. In der Variable *current* wird somit das aktuelle Szenario weiter aufgebaut.

while $buffer1 \neq \langle\rangle$ **do**
$current := Head(buffer1)$;

Hierzu wird ermittelt welche ausgehenden *Transitions* von dem Zustand abgehen, auf welche der letzte Zustand im aktuellen Szenario verweist. Diese Sequenz wird in der Variable *buffer2* abgelegt.

$buffer2 := Reachable(Last(current).destination)$;

Im Anschluss wird jede in *buffer2* abgelegte Kante ausgewertet. Wenn der aktuelle Eintrag in *buffer2* auf einen Zustand verweist der bereits durch eine Kante die in *current* steht zugänglich ist, wurde eine Schleife gefunden und das Szenario ist vollständig. In diesem Fall wird die aktuelle Sequenz in *current* als Element der Sequenz in *scenarios* hinzugefügt. Andernfalls wird das aktuelle Element in *buffer2* dem aktuellen Szenario in *current* abgehangen und das Ergebnis wird der Variable *buffer1* abgehangen, damit das Szenario später weiter untersucht werden kann.

```
  while buffer2 ≠ ⟨⟩ do
    if ContainsDestination(current, Head(buffer2).destination) then
      scenarios := Append(scenarios, current) ;
    else
      buffer1 := Append(buffer1, Append(current, Head(buffer2))) ;
    end if ;
    buffer2 := Tail(buffer2) ;
  end while ;
```

In einem Durchlauf werden für *genau ein* Szenario alle weiterführenden Kanten betrachtet. Abschließend wird das erste Element aus *buffer1* entfernt, da auf seiner Grundlage ja ein fertiges Szenario gefunden wurde oder weitere zur Untersuchung der gleichen Variable hinzugefügt wurden.

```
    buffer1 := Tail(buffer1) ;
  end while ;
```

Nach Durchlauf dieser Schleife befinden sich in der Variable *scenarios* alle ermittelten Szenarios in der Form von Sequenzen von *Transition*s. Im nächsten Schritt werden diese durchlaufen, um für jedes dieser Szenarien *ProcessRecord*s und *MessageRecord*s zu erzeugen. Hierzu wird zunächst das aktuelle Szenario in der Variable *scenario* gespeichert und der Inhalt der Variable *processRecords* auf eine leere Sequenz zurückgesetzt. Anschließend wird der Variable *buffer1* die Sequenz aller *ProcessInfo*s des Startzustandes zugewiesen.

```
  while scenarios ≠ ⟨⟩ do
    scenario := Head(scenarios) ;
                processRecords := ⟨⟩ ;
    buffer1 := START.processesInfo ;
```

Dies ist notwendig, um für jeden existierenden Prozess den *ProcessRecord* zu erzeugen. Dieser *ProcessRecord* erhält die ID des Prozesses und darüber hinaus eine leere Sequenz für gesendete und empfangene Nachrichten. Des Weiteren wird der Sequenz in der Variable *processIds* die ID des aktuellen Prozesses angehängt.

```
    while buffer1 ≠ ⟨⟩ do
      processRecords := Append(
                          processRecords,
                          [
                            id ↦ Head(buffer1).id,
                            sent ↦ ⟨⟩,
                            received ↦ ⟨⟩
                          ]
                        ) ;
      processIds := Append(processIds, Head(buffer1).id) ;
      buffer1 := Tail(buffer1) ;
    end while ;
```

Im nächsten Schritt werden die *MessageRecord*s für die einzelnen Nachrichten erzeugt. Hierzu wird zunächst der Wert der Variable *messageRecords* auf eine leere Sequenz zurückgesetzt. Anschließend werden die Einträge in der Variable *scenario* einzeln durchlaufen.

```
messageRecords := ⟨⟩ ;
while scenario ≠ ⟨⟩ do
```

Anschließend werden die *ProcessInfo*s des Zustandes, von dem die aktuelle Kante in der Variable *scenario* ausgeht, in *buffer1* gespeichert. Diese *ProcessInfo*s werden durchlaufen und die Warteschlange eingehender Nachrichten, welche aus *Message*s besteht, wird in der Variable *buffer2* abgelegt. Nun wird auch der Inhalt der Variable *buffer2* durchlaufen.

```
  buffer1 := ElementById(STATES, Head(scenario).source).processesInfo ;
  while buffer1 ≠ ⟨⟩ do
    buffer2 := Head(buffer1).queue ;
    while buffer2 ≠ ⟨⟩ do
```

Wenn die ID dieser *Message* noch nicht in der Variable *messageIds* abgelegt ist, wurde für diese Nachricht noch kein *MessageRecord* erzeugt. In diesem Fall wird der *MessageRecord* mit dem aus dem *Message*-Element bekannten Daten angelegt. Anschließend wird die ID der Variable *messageIds* hinzugefügt.

```
      if NotContains(messageIds, Head(buffer2).id) then
        messageRecords := Append(
                            messageRecords,
                            [
                              id ↦ Head(buffer2).id,
                              sent ↦ Head(buffer2).sent,
                              sender ↦ Head(buffer2).sender,
                              receiver ↦ Head(buffer1).id,
                              type ↦ Head(buffer2).type,
                              beginProcessing ↦ 0,
                              endProcessing ↦ 0
                            ]
                          ) ;
        messageIds := Append(messageIds, Head(buffer2).id) ;
      end if ;
```

Am Ende der jeweiligen Schleife werden die ersten Elemente aus den Sequenzen in *buffer1* und *buffer2* entfernt, um die weitere Abarbeitung der Schleife zu gewährleisten.

```
      buffer2 := Tail(buffer2) ;
    end while ;
    buffer1 := Tail(buffer1) ;
  end while ;
```

Die nun in *messageRecords* abgelegten *MessageRecord*s beinhalten die Information, wann eine Nachricht von wem an wen versandt wurde. Es fehlt somit noch die Information, wann die Verarbeitung dieser Nachricht begann und wann sie endete. Diese Information ergibt sich aus der *Transition*, in der abgelegt ist, welcher Prozess welche Nachricht verarbeitet hat. Um also diese Information zu erzeugen, wird den Feldern *beginProcessing* und *endProcessing* der Wert der Uhr des Prozesses, der eine Nachricht verarbeitet hat, aus den beiden verbundenen Zuständen zugewiesen.

```
  messageRecords := [
    messageRecords EXCEPT
      ![IndexOf(messageRecords, Head(scenario).messageId)] =
      [@ EXCEPT
       !.beginProcessing =
         ClockOfProcessInState(Head(scenario).processId, Head(scenario).source),
       !.endProcessing =
         ClockOfProcessInState(Head(scenario).processId, Head(scenario).destination)
      ]
    ];
```

Abschließend wird das erste Element aus der Variable *scenario* entfernt, damit die nächste Kante des Szenarios abgearbeitet werden kann.

```
  scenario := Tail(scenario);
end while ;
```

Wurden alle *MessageRecord*s erzeugt, müssen diese noch in die richtigen *ProcessRecord*s eingetragen werden. Für jeden Prozess wurde bereits vorab ein *ProcessRecord* in der Variable *processRecords* erzeugt. Nun müssen noch in die Felder *sent* und *received* die richtigen *MessageRecord*s eingetragen werden.

Hierzu werden alle in *messageIds* abgelegten IDs durchlaufen. Zuerst wird ermittelt, welcher Prozess der sendende Prozess dieser Nachricht ist. Beim entsprechenden *ProcessRecord* wird dann dieser *MessageRecord* dem Feld *sent* hinzugefügt.

```
while messageIds ≠ ⟨⟩ do
  processRecords := [
    processRecords EXCEPT
      ![
        IndexOf(
          processRecords,
          messageRecords[IndexOf(messageRecords, Head(messageIds))].sender
        )
      ] = [@ EXCEPT !.sent = Append(@, messageRecords[Head(messageIds)])];
```

Anschließend wird ermittelt, welcher Prozess die Nachricht empfangen hat. Dem *ProcessRecord* dieses Prozesses wird der *MessageRecord* dem Feld *received* hinzugefügt.

```
  processRecords := [
```

```
      processRecords EXCEPT
        ![
          IndexOf(
            processRecords,
            messageRecords[IndexOf(messageRecords, Head(messageIds))].receiver
          )
        ] = [@ EXCEPT !.received = Append(@, messageRecords[Head(messageIds)])] ;
    messageIds := Tail(messageIds) ;
  end while ;
```

Nach dem Durchlauf dieser Schleife befinden sich für das aktuelle Szenario alle *ProcessRecord*s in der Variable *processRecords*. Darüber hinaus verweisen alle *ProcessRecord*s auf die *MessageRecord*s von Nachrichten, die sie gesendet oder empfangen haben. Dieses Ergebnis wird somit der Variable *results* als ein Szenario hinzugefügt. Anschließend wird der Inhalt der Variable *processIds* zurückgesetzt. Damit beim nächsten Schleifendurchlauf das nächste Szenario verarbeitet werden kann, wird das erste Element aus der Variable *scenarios* entfernt.

```
  results := Append(results, processRecords) ;
  processIds := ⟨⟩ ;
  scenarios := Tail(scenarios) ;
end while ;

end
```

Nach dem Durchlauf des Algorithmus befinden sich in der Variable *results* für jedes Szenario die *ProcessRecord*s, welche wiederum auf die *MessageRecord*s verweisen, wie sie auch durch das Profiling erzeugt werden. Die in Kapitel 5 beschriebenen Algorithmen können somit verwendet werden, um für ein Szenario die Metriken **M1**–**M3** zu berechnen.

7 Prototypische Implementierung für Simulator X

In diesem Kapitel wird eine prototypische Implementierung der vorgestellten Konzepte beschrieben. Das Kapitel beginnt mit einer Darstellung des Simulator X Frameworks, welches als Grundlage für die Umsetzung dient. Im Anschluss wird die Umsetzung einer DSL für die Steuerung der Nebenläufigkeit und Synchronisation vorgestellt. Daraufhin wird die Implementierung des Profilers dargestellt. Das Kapitel schließt mit der Implementierungsbeschreibung der Model-Checking-Anbindung ab.

7.1 Das Simulator X Framework

Die in den vorherigen Kapiteln vorgestellten Konzepte wurden auf Basis von Simulator X (Latoschik und Tramberend, 2011b,a) umgesetzt. Simulator X ist ein asynchrones Interaktives Echtzeitsystem, welches eine Kommunikation über Prozessvariablen realisiert. Es verwendet somit sowohl das richtige Modell (vgl. Tabelle 2.1 von Seite 26), als auch die richtige Synchronisationstechnik (vgl. Tabelle 2.2 von Seite 33). Simulator X ist das einzige System mit dieser Kombination und damit, neben einer aufwendigen Neuimplementierung, alternativlos. In diesem Abschnitt erfolgt eine kurze Einführung in die Grundkonzepte von Simulator X.

7.1.1 Architektur

Simulator X ist ein in Scala (Odersky, 2011) implementiertes asynchrones Interaktives Echtzeitsystem, das auf Hewitts Actor Model (Hewitt et al., 1973), also einem Spezialfall des *Message-Passing*, basiert. Im Actor Model ist ein System aus nebenläufigen Aktoren aufgebaut. Jeder dieser Aktoren hat seinen eigenen Speicherbereich, den nur er selbst manipulieren kann. Die Kommunikation mit anderen Aktoren findet ausschließlich über Nachrichten statt. Ein Aktor kann hierfür jedem anderen beliebigen Aktor eine Nachricht schicken. Ein Aktor verarbeitet zu einem Zeitpunkt nur eine Nachricht. Während der Verarbeitung einer Nachricht kann ein Aktor seinen eigenen Zustand (Speicher) verändern, sein Verhalten ändern, anderen Aktoren, inklusive sich selbst, eine Nachricht schicken und neue Aktoren erzeugen.

Komponenten

Auf dieser Grundlage führt Simulator X das Konzept der *Komponenten* ein. Die Komponente entspricht im Kontext von Simulator X den in Kapitel 2.1.2 definierten Begriff

des Sub-Systems. Komponenten kapseln somit Ein- und Ausgabegeräte oder führen Berechnungen für die Simulation durch. Eine Komponente kann hierbei aus einem Aktor oder mehreren Aktoren bestehen. Der erste Ansatz wird üblicherweise gewählt, wenn ein externes Sub-System monolithisch gekapselt und innerhalb von Simulator X zur Verfügung gestellt werden soll. Zwei Beispiele hierfür sind die mit Simulator X ausgelieferten Anbindung an die Physik-Engine JBullet (JBullet, 2016) und die Geräteschnitstelle VRPN (Taylor et al., 2001). Der zweite Ansatz, also mehrere Aktoren pro Komponente, kann verwendet werden wenn Komponenten vollständig selbst implementiert oder mehrere Instanzen benötigt werden. Dieser Ansatz wurde z. B. bei der Rendering-Anbindung zum Renderer JVR (Roßbach, 2010) in Simulator X verwendet um ein Rendering auf mehreren Knoten eines Clusters zur ermöglichen (Rehfeld et al., 2013). Abbildung 7.1 zeigt für dieses Beispiel, welcher Aktor welchen anderen erzeugt.

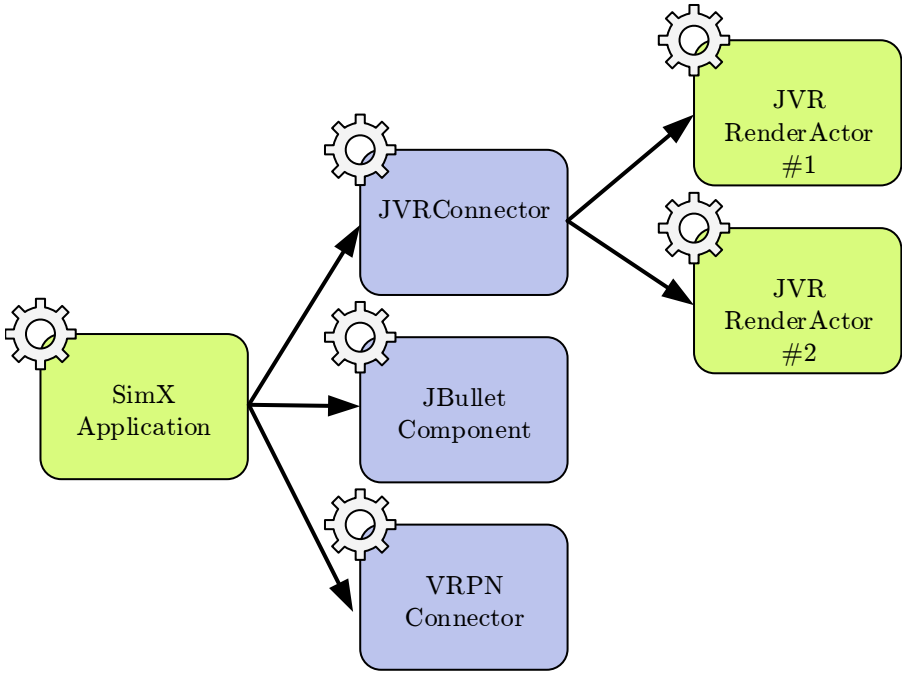


Abbildung 7.1: Exemplarische Darstellung der Erzeugung verschiedener Aktoren. Der Aktor *SimXApplication* ist für das Bootstrapping der Anwendung verantwortlich. Er erzeugt drei Komponenten. Die *JBulletComponent* und der *VRPNConnector* stellen monolithische Komponenten dar. Der *JVRConnector* erzeugt fürs Rendering jedoch zwei weitere Aktoren.

Abbildung 7.2 zeigt einen Auszug aus der Klassenhierarchie der Implementierung der Komponente. Hier ist zu erkennen, dass eine Komponente immer fähig ist Entitäten, die im nächsten Abschnitt beschrieben werden, zu verwalten, was durch den Trait `EntityConfigLayer` realisiert ist. Darüber hinaus ist eine Komponente auch immer ein Aktor, der Zustandsvariablen verwalten kann, was in der Klasse `SVarActor` implementiert ist.

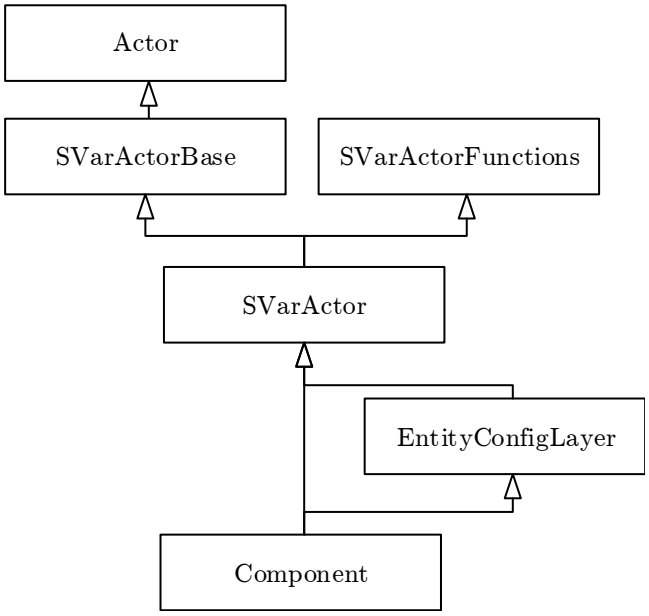


Abbildung 7.2: Ausschnitt aus der Klassenhierarchie einer Komponente in Simulator X vor der Umsetzung der Konzepte dieser Arbeit.

Entitäten

Eingabegeräte und simulierte Objekte werden in Simulator X in Form von *Entitäten* (engl. *Entities*) repräsentiert. Eine Entität stellt hierbei eine Abstraktion eines Objektes dar, welche für mehrere Komponenten existieren kann und übersetzt die komponentenspezifische Darstellung in eine unabhängige Form. Ein Beispiel hierfür ist die Transformation, die sowohl für die Physik-Simulation, als auch für den Renderer relevant sind. Technisch gesehen sind Entitäten eine Abbildung von Ontologieeinträgen nach Zustandsvariablen. Die Abbildung 7.3 zeigt als Beispiel eine Entität mit drei Eigenschaften. Die Transformation gibt hierbei die Position und Ausrichtung des Objekts an, während die Eigenschaften *velocity* und *torque* die Richtung- und Rotationsgeschwindigkeit des Objekts darstellen.

Die Erzeugung und Verwaltung der Entitäten ist in der Klasse `EntityConfigLayer` implementiert. Das Protokoll hierfür wurde durch Wiebusch (2015) beschrieben.

Zustandsvariablen

Zustandsvariablen (engl. State Variables) sind in Simulator X die feingranularste Struktur. Sie repräsentieren getypt einen einzelnen Wert und stellen im Kontext einer Entität eine einzelne Eigenschaft dar. Hierbei beinhalten die Zustandsvariablen nicht selbst einen Wert, sondern stellen lediglich ein einheitliches Interface dar, welches folgende Operationen erlaubt:

- Schreiben

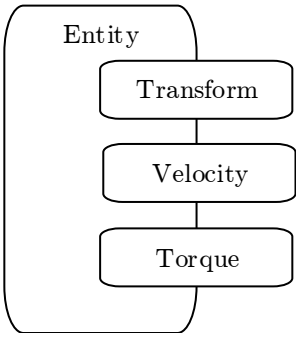


Abbildung 7.3: Eine Entität mit den drei Eigenschaften Transformation (Transform), Geschwindigkeit (Velocity) und Rotationsgeschwindigkeit (Torque).

- Lesen
- Beobachten

Beim Beobachten einer Zustandsvariable wird der Beobachter bei einer Änderung des Werts automatisch informiert. Hierbei ist jede Zustandsvariable genau einem Aktor zugewiesen, der sie verwaltet. Beim Lesen wird der Wert einer Zustandsvariable asynchron angefragt und vom verwaltenden Aktor zurückgesandt. Beim Schreiben wird ein neuer Wert an den verwaltenden Aktor gesendet. Wie dieser jedoch darauf reagiert, hängt vom verwaltenden Aktor ab.

Die Abbildung 7.4 zeigt hier ein vollständiges Beispiel. Eine exemplarische Entität besteht aus drei Eigenschaften, die durch Zustandsvariablen abgebildet werden. Alle drei Zustandsvariablen werden durch die Physik-Simulation geschrieben. Die zwei Rendering-Aktoren beobachten eine dieser Zustandsvariablen, da nur die aktuelle Transformation für sie relevant ist. Abbildung 7.4 zeigt auch, dass es sich bei Entitäten und Zustandsvariablen um rein logische Konstrukte handelt. Die Kommunikation verläuft, wie durch die gestrichelte Linie gezeigt wird, direkt zwischen den Aktoren.

7.2 Implementierung eigener Konzepte

In diesem Abschnitt wird die Implementierung der vorher vorgestellten Konzepte beschrieben. Dies beginnt mit der Vorstellung der Implementierung einer Domain Specific Language (DSL), in der die im Kapitel 3 vorgestellten Schemata mittels der dort definierten Grundelemente im Simulator X umgesetzt werden können.

Daraufhin wird die Implementierung des eines Profilers für Simulator X vorgestellt. Anschließend wird die Implementierung der Model-Checking-Anbindung beschrieben.

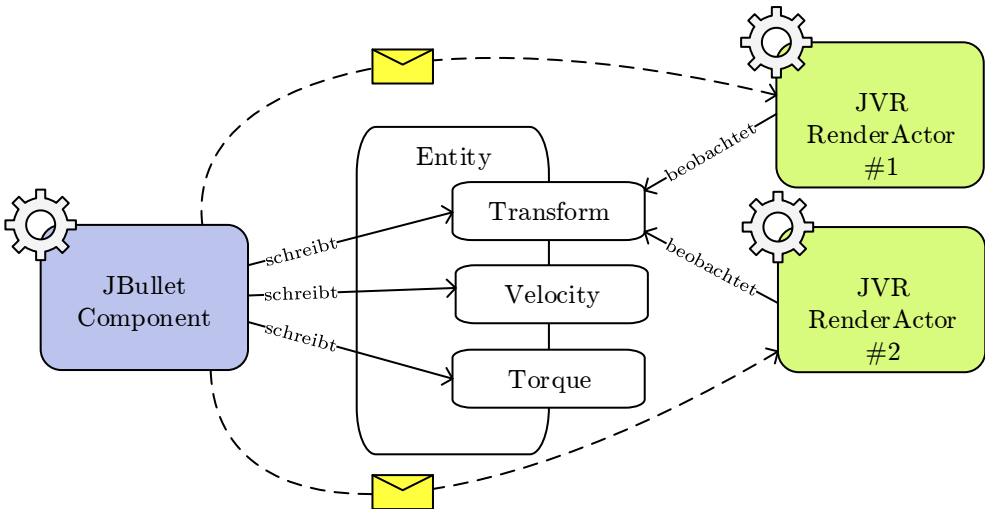


Abbildung 7.4: Ein Beispiel einer Entität mit drei Eigenschaften, welche durch Zustandsvariablen abgebildet werden. Eine der Zustandsvariablen wird durch zwei andere Aktoren beobachtet. Die Kommunikation erfolgt jedoch direkt zwischen den Aktoren, da es sich bei Entitäten und Zustandsvariablen um rein logische Konstrukte handelt.

7.2.1 Nebenläufigkeit

Grundelemente

Im Folgenden wird aufgeführt, in welcher Form die Grundelemente aus Kapitel 3.2 in Simulator X implementiert wurden.

GE1: Unbewegter Beweger Der Unbewegte Beweger stellt in der Implementierung innerhalb von Simulator X kein eigenständiges Element dar. Stattdessen werden initiale Nachrichten zum Start der Anwendung nach der Erzeugung aller Komponenten und deren Konfiguration mit der im Abschnitt 7.2.1 beschriebenen DSL versandt.

GE2: Sub-System Die Repräsentation eines Sub-Systems ist die Komponente aus Simulator X. In einer früheren Implementierung hatte jede Komponente ihre eigene Nachricht, die zu einem neuen Simulationsschritt führte. Diese Nachrichten wurden durch die Nachricht `PerformSimulationStep` ersetzt. Die Entgegennahme dieser Nachricht findet durch den Trait `ExecutionStrategySupport` statt. Empfängt ein Aktor, der diesen Trait verwendet, eine Nachricht vom Typ `PerformSimulationStep`, wird zunächst die interne Methode `startSimulation` aufgerufen. In dieser wird der aktuelle Zeitstempel gespeichert. Anschließend wird an alle anderen Aktoren, die beim Beginn des Simulationsschrittes (OnStepBegin) getriggert werden sollen, eine `PerformSimulationStep` Nachricht gesendet. Im Anschluss wird die abstrakte Methode `performSimulationStep` aufgerufen, welche durch die Implementierungen der Komponenten definiert wird.

Bei einer grobgranularen Komponente, die beispielsweise eine externe Physik-Engine kapselt, würde innerhalb dieser Methode der Simulationsschritt ausgeführt werden. Bei feingranularen Komponenten könnten in dieser Methode komponentenspezifische Nachrichten an andere Aktoren versandt werden, welche dann die Berechnung durchführen. Um dem letzteren Szenario Rechnung zu tragen, ist der Simulationsschritt nicht zu Ende, wenn die Methode `performSimulationStep` beendet wurde, sondern wenn die Komponente die Methode `simulationCompleted` aufruft.

Wird die Methode `simulationCompleted` aufgerufen, wird zunächst an alle Aktoren eine `PerformSimulationStep`-Nachricht geschickt, die am Ende des Simulationsschritts getriggert werden sollen (OnStepEnd). Im Anschluss hängt es von der konfigurierten Frequenz der Komponente ab, was passiert. Ist keine Begrenzung eingestellt (Unbound), schickt der Aktor sich selbst eine Nachricht vom Typ `PerformSimulationStep`. Ist der Prozess so eingestellt, dass er nur von außen getriggert werden kann (Triggered), wird keine Aktion durchgeführt. Ist seine Frequenz begrenzt, werden die im nächsten Abschnitt beschriebenen Aktionen durchgeführt.

GE3: Frequenzbegrenzer Der Frequenzbegrenzer wird im Gegensatz zum Konzept aus Kapitel 3.2 nicht als eigener Aktor abgebildet, sondern ist Teil der Komponente in Form eines Features vom Trait `ExecutionStrategySupport`. Für jede Komponente kann über die in Abschnitt 7.2.1 beschriebene DSL eine Obergrenze in der Frequenz festgelegt werden. Zur Feinabstimmung kann noch das Verhalten konfiguriert werden: In dem Fall, dass der Simulationsschritt zu lange dauerte, um die Frequenz zu halten:

1. Ein Frame wird ausgelassen und die `PerformSimulationStep`-Nachricht wird so versandt, dass sie möglichst in den regelmäßigen Takt passt. Sind z. B. bei einem Renderer 60 Hz eingestellt, würde so die Frequenz auf 30 Hz sinken.
2. Der Aktor sendet sich selbst ohne Verzögerung eine `PerformSimulationStep`-Nachricht, wodurch der nächste Simulationsschritt möglichst bald ausgelöst wird.
3. Der Aktor ruft selbst noch einmal die `startSimulation`-Methode auf, wodurch der nächste Simulationsschritt sofort erzwungen wird.

Der Unterschied zwischen dem zweiten und dem dritten Verhalten ist, dass der Aktor beim zweiten Verhalten noch Nachrichten abarbeitet, die während des Simulationsschrittes eingetroffen sind und darüber hinaus die Kontrolle an den Scheduler der Aktor-Implementierung abgibt, wodurch auch andere Aktoren eine Aktion durchführen können. Beim dritten Verhalten hingegen werden keine eingetroffenen Nachrichten verarbeitet und der Aktor gibt die Kontrolle nicht wieder an den Scheduler zurück.

GE4: Frequenz-Trigger Wie bereits vorab beschrieben, steht der Frequenz-Trigger für Prozesse die eine eigene Frequenz haben, die jedoch nicht unter der Kontrolle der Anwendung liegen. Ein Frequenz-Trigger existiert somit nicht als eigenständige Implementierung, sondern existiert in der Form von Komponenten die EA-Geräte, wie z. B. ein Tracking-System, repräsentieren.

GE5: Barrier Die Barrier wird durch einen eigenständigen Aktor präsentiert. Bei der verwendeten Aktor Bibliothek Akka wird das Verhalten eines Aktors durch eine *Partial Function* bestimmt. Die erstellte Implementierung einer Barrier ist von `ParticalFunction` abgeleitet, wodurch eine Instanz als Verhalten für einen Aktor verwendet werden kann. Darüber hinaus wurde eine DSL implementiert, über die die Barrier konfiguriert werden kann.

Listing 7.1 zeigt das Beispiel einer Barrier. Diese Barrier sendet, nachdem sie zwei Nachrichten vom Typ `PerformSimulationStep` erhalten hat, je eine Nachricht vom Typ `PerformSimulationStep` an die beiden Aktoren *physics* und *gfx*.

Listing 7.1: Einfaches Beispiel einer Barrier.

```
1 class BarrierActor extends Actor {
2   def receive = Barrier afterReceived
3     2 messageOfType
4     classOf[PerformSimulationStep] sendMessageOfType
5     classOf[PerformSimulationStep] to Set( physics, gfx )
6 }
7
8 val barrier = context.actorOf(Props( new BarrierActor ))
```

Die Barrier erlaubt es auch, komplexere Bedingungen mit *und* sowie *oder* Verknüpfung zu erzeugen, sowie bei einer Nachricht nicht nur nach dem Typ, sondern auch nach dem Sender der Nachricht einzuschränken. Beides wird in Listing 7.2 gezeigt. Dort wird eine Barrier erzeugt, die eine Nachricht vom Typ `PerformSimulationStep` an die Aktoren *actor1* und *actor2* schickt, wenn sie eine Nachricht vom Typ `MessageType1` von einem beliebigen Aktor bekommt oder wenn sie zwei Nachrichten vom Typ `MessageType2` vom Aktor *anActor* bekommt.

Listing 7.2: Beispiel einer Barrier, die auf zwei verschiedenen Typen von Nachrichten reagiert, wobei bei einem Typ auch der Sender der Nachricht ausgewertet wird.

```
1  class BarrierActor extends Actor {
2    def receive = Barrier afterReceived
3      1 messageOfType
4      classOf[MessageType1] or orAfterReceived
5                   2 messageOfType
6                   classOf[MessageType2] sentBy anActor
7                   sendMessageOfType
8      classOf[PerformSimulationStep] to Set( actor1, actor2 )
9  }
10
11 val barrier = context.actorOf(Props( new BarrierActor ))
```

DSL zur Konfiguration

Um die Konfiguration einer Anwendung möglichst einfach und an einer zentralen Stelle zu ermöglichen, wurde eine DSL erstellt, über die sich Frequenzen und Trigger-Reihenfolgen einstellen lassen. Die DSL wird über die Klasse `ExecutionStrategy` bereitgestellt. Das Listing 7.3 zeigt hier als einfaches Beispiel das **Schema 1** aus Kapitel 3.1.1.

Eine neue Ausführungsstrategie wird durch `ExecutionStrategy` begonnen. Nach dem Bezeichner `where` werden zunächst alle Komponenten aufgeführt und konfiguriert, ob und wie deren Frequenz begrenzt werden soll. Das **Schema 1** sieht keine Begrenzung der Frequenz vor und die Komponenten sollen nacheinander ausgeführt werden. Somit werden beide Komponenten so konfiguriert, dass sie nur einen Simulationsschritt durchführen, wenn sie von außen getriggert werden.

Nachdem die Frequenz aller Komponenten eingestellt ist, wird konfiguriert, in welcher Reihenfolge zu welchem Zeitpunkt sich diese gegenseitig triggern sollen. Im Listing 7.3 triggern sich die Physik und der Renderer gegenseitig, nachdem sie einen Simulationsschritt abgeschlossen haben.

Zum Abschluss wird noch festgelegt welche Komponenten initial beim Start getriggert werden sollen.

Listing 7.3: Das **Schema 1** aus Kapitel 3.1.1 konfiguriert mit der DSL in Simulator X.

```
1 val es = ExecutionStrategy where
2   physics runs Triggered() and
3   gfx runs Triggered()
4   where
5   gfx isTriggeredBy physics onStepComplete() and
6   physics isTriggeredBy gfx onStepComplete() startWith(
7           Set() + gfx )
8   )
```

Wie das **Schema 2** aus Kapitel 3.1.2 erzeugt wird, ist in Listing 7.4 dargestellt. Im Gegensatz zum vorherigen Schema werden hier die Komponenten nicht getriggert, sondern sie laufen ohne Limitierung der Frequenz, was durch die Bezeichnung *Unbound* eingestellt wird. Folglich muss auch keine Reihenfolge des Triggerings festgelegt werden. Darüber hinaus werden Komponenten, die keine Begrenzung in der Frequenz haben, automatisch gestartet, müssen also nicht noch einmal explizit zum Start der Anwendung getriggert werden.

Listing 7.4: Das **Schema 2** aus Kapitel 3.1.2 konfiguriert mit der DSL in Simulator X.

```
1 val es = ExecutionStrategy where
2   physics runs Unbound() and
3   gfx runs Unbound()
```

Da wie vorab dargestellt eine unbegrenzte Frequenz weder wünschenswert noch sinnvoll ist, zeigt das Listing 7.5 wie das **Schema 3** aus Kapitel 3.1.3 umgesetzt wird. Hierbei wird für jede Komponente festgelegt, mit welcher maximalen Frequenz diese laufen soll. Auch

in diesem Fall werden die Komponenten automatisch gestartet, weshalb keine Reihenfolge des Triggerings festgelegt werden muss und keine initiale Trigger-Nachricht versandt werden muss.

Listing 7.5: Das **Schema 3** aus Kapitel 3.1.3, konfiguriert mit der DSL in Simulator X.

```
1 val es = ExecutionStrategy where
2   physics runs Soft(120) and
3   gfx runs Soft(60)
```

Komplexer ist die Umsetzung von **Schema 4** aus Kapitel 3.1.4. Hierzu wird, wie in Listing 7.6 dargestellt, zunächst eine Barrier benötigt, die erst dann eine Nachricht vom Typ `PerformSimulationStep` an die Physik und den Renderer schickt, wenn sie vorher zwei Nachrichten vom selben Typ erhalten hat.

Aus dieser Barrier wird im nächsten Schritt ein Aktor gemacht. Dieser Aktor wiederum wird in der Ausführungsstrategie verwendet. Sowohl die Physik als auch der Renderer werden so eingestellt, dass sie nur einen Simulationsschritt durchführen, wenn sie getriggert werden. Darüber hinaus werden beide Komponenten so eingestellt, dass die nach Abschluss des Simulationsschritts die Barrier triggern. Initial wird sowohl an den Renderer als auch an die Physik eine Trigger-Nachricht geschickt.

Listing 7.6: Das **Schema 4** aus Kapitel 3.1.4 konfiguriert mit der DSL in Simulator X.

```
1  class BarrierActor extends Actor {
2    def receive = Barrier afterReceived
3    2 messageOfType
4    classOf[PerformSimulationStep] sendMessageOfType
5    classOf[PerformSimulationStep] to Set( physics, gfx) )
6  }
7
8  val barrier = context.actorOf(Props( new BarrierActor ))
9  val es = ExecutionStrategy where
10   physics runs Triggered() and
11   gfxName runs Triggered()
12   where
13   barrier isTriggeredBy physics onStepComplete() and
14   barrier isTriggeredBy gfx onStepComplete() startWith Set(
15           gfx, physics )
16         )
```

Das Listing 7.7 zeigt wie das **Schema 5** aus Kapitel 3.1.5 umgesetzt wird. Hierbei wird zunächst die Tracking-Anbindung so konfiguriert, dass sie den Renderer triggert. Da die Frequenz der Tracking-Anbindung nicht durch die Applikation beeinflussbar ist, wird sie in der Ausführungsstrategie nicht erwähnt. Dort wird lediglich eingestellt, dass die Physik mit 120 Hz laufen soll und der Renderer getriggert wird.

Listing 7.7: Das **Schema 5** aus Kapitel 3.1.5 konfiguriert mit der DSL in Simulator X.

```
1 VRPNConnector.toTrigger = Some( gfx )
```

```
2 val es = ExecutionStrategy where
3   physics runs Soft(120) and
4   gfx runs Triggered() )
```

Der Trait `ExecutionStrategyHandling` stellt, wie in Listing 7.8 dargestellt, die Funktion `start` zur Verfügung. An diese wird die erstellte `ExecutionStrategy` übergeben. Diese teilt allen Komponenten ihre konfigurierte Frequenz mit. Im Anschluss werden die initialen Trigger-Nachrichten gesendet, wodurch die Anwendung gestartet ist.

Listing 7.8: Ist eine `ExecutionStrategy` erstellt, wird die Anwendung über die Funktion `start` gestartet.

```
1 val es = ExecutionStrategy where ...
2
3 start( es )
```

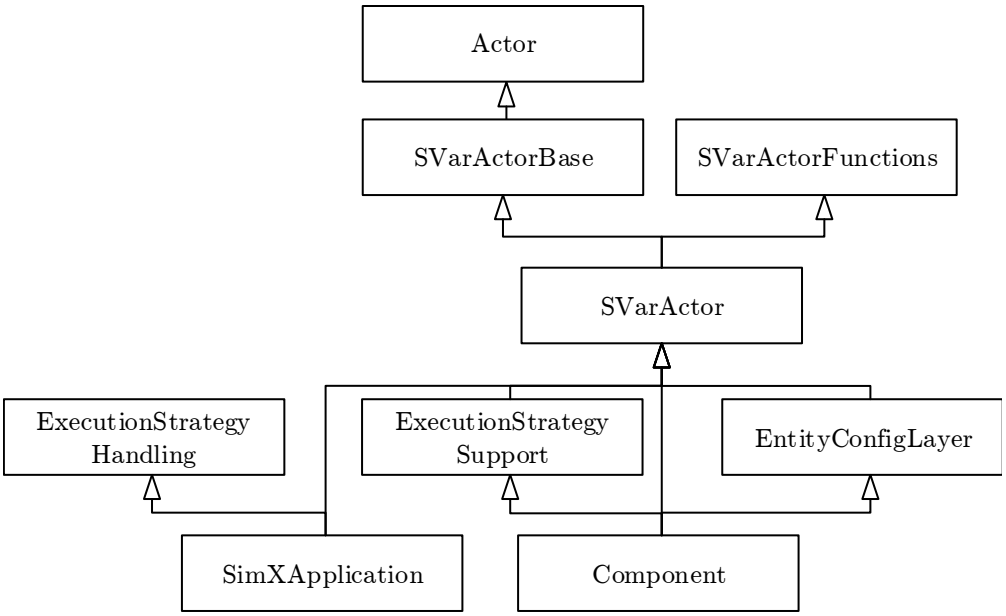


Abbildung 7.5: Die Implementierung der Ausführungsstrategie innerhalb von Simulator X. Der hier zu sehende Ausschnitt der Klassenhierarchie zeigt denselben Ausschnitt wie Abbildung 7.2, jedoch nach dem Einbau der hier vorgestellten Konzepte. Alle Konzepte wurden innerhalb von `SimXApplication`, `ExecutionStrategySupport` und `ExecutionStrategyHandling` umgesetzt.

Die Abbildung 7.5 zeigt, wie sich die Implementierung in die bestehende Struktur von Simulator X einfügt. Der Komponente wurde der Trait *ExecutionStrategySupport*, die die Konfiguration der Komponente anhand der empfangenen Strategie vornimmt. Der Klasse *SimXApplication* wurde der Trait `ExecutionStrategyHandling` hinzugefügt, welcher die oben beschriebene Methode zum Starten der Komponenten hat.

7.2.2 Profiler

In diesem Abschnitt wird die Implementierung des Profilers beschrieben. Dies beginnt mit einer Beschreibung der Implementierung der Instrumentierung für die Sammlung der Daten. Aus diesen Daten werden die in Kapitel 4 definierten Metriken mit Hilfe der in Kapitel 5 spezifizierten Algorithmen berechnet.

Im zweiten Teil dieses Abschnitts werden die erarbeiteten Darstellungen für diese Metriken vorgestellt.

Datensammlung

Grundlegend müssen die in Kapitel 5.1.1 durch die Strukturen *MessageRecord* und *ProcessRecord* abgebildeten Daten gesammelt werden. Da sich die Metriken **M1**–**M3** ausschließlich auf die Schicht des Message-Passings beziehen und keine Zähler der Prozessoren benötigt werden, wurde ein reiner Instrumentierungsansatz verfolgt. Darüber erlaubt die Metrik **M3** nicht die Verwendung von Daten, die durch Sampling erhoben werden.

In Abschnitt 7.1 wurde beschrieben, dass Simulator X in Scala implementiert wurde, wobei der Scala-Compiler Byte-Code für die Java Virtual Machine (JVM) erzeugt. Da somit Simulator X und die damit erstellen Anwendungen innerhalb der JVM laufen, bietet es sich folgerichtig an, Funktionen der JVM für die Instrumentierung zu verwenden. In Kapitel 2.3.2 wurde beschrieben wie in Java die Instrumentierung über Java Agents und Transformatoren erfolgt.

Daten die den Strukturen *MessageRecord* und *ProcessRecord* entsprechen, lassen sich am einfachsten durch die Instrumentierung der verwendeten Aktor-Bibliothek gewinnen. Dies hat darüber hinaus den Vorteil, dass die Benutzung der Instrumentierung und des Profilers nicht auf Simulator X beschränkt ist. Für die Manipulation des Byte Codes stehen eine Reihe von Bibliotheken zur Verfügung. Diese sind jedoch eher darauf ausgelegt, einzelne Zähler in Funktionen einzubauen. Für die Instrumentierung im Rahmen dieser Arbeit ist es jedoch notwendig, Datenstrukturen wie durch *MessageRecord* und *ProcessRecord* beschrieben abzulegen.

Es wurde somit ein alternativer Ansatz gewählt. Da der Quelltext der verwendeten Aktor-Bibliothek offenliegt, wurde dieser manuell instrumentiert. Die Kompilate der modifizierten Klassen werden im Anschluss durch einen veränderten Class Loader geladen.

Bei der Instrumentierung stellt sich wiederum die Frage, ob diese eine Abfrage der Daten in Echtzeit erlaubt oder, ob diese abschließend in eine Datei geschrieben werden und im Nachhinein ausgewertet und visualisiert werden. Wie bereits in Kapitel 2.3.2 dargestellt, erlauben Profiler für Java und die Profiler für die Engines Unity 3D und Unreal das Sammeln und Visualisieren von Daten einer laufenden Anwendung. Das Problem ist jedoch, dass Interaktive Echtzeitanwendungen die Ressourcen des Systems bereits stark in Anspruch nehmen. Eine Instrumentierung erhöht dies naturgemäß weiter. Würde die Auswertung und Visualisierung der Daten zusätzlich auf dem gleichen System erfolgen, würde dies das Laufzeitverhalten der Anwendung noch stärker verändern. Gleiches gilt auch, sollten die Daten über eine Netzwerkverbindung übertragen werden, da je nach Architektur der Hardware, EA-Operationen einen erheblichen Einfluss auf alle System-

komponenten, auch die CPU, haben.

Die hier vorgestellte Instrumentierung legt zunächst alle gesammelten Daten im Arbeitsspeicher ab. Erst wenn die Anwendung beendet wird, werden diese Daten auf die Festplatte geschrieben. Für Operationen die nach dem Beenden der Applikation ausgeführt werden sollen, bietet die Java Virtual Machine die Möglichkeit *shutdown hooks* zu registrieren. Somit wird ein shutdown hook registriert, der die gesammelten Daten nach dem Beenden der Anwendung auf die Festplatte schreibt.

Die Instrumentierung schreibt nach dem Beenden der Anwendung zwei für diese Arbeit wichtige Dateien. Eine Datei beinhaltet die folgenden Informationen über gesendete Nachrichten:

- Zeitstempel des Sendevorgangs
- Eindeutige ID des Senders
- Eindeutige ID des Empfängers
- Datentyp der Nachricht
- Eindeutige ID der Nachricht

Die Abbildung 7.6 zeigt hier Beispieldaten in dem vorher definierten Format.

sending time	from	to	message type	message id
6,21254E+14	akka://ExampleApplication/temp/$a	akka://ExampleApplication/system/log1-L	class akka.event.Logging$InitializeLog	1083750747
6,21371E+14	null	akka://ExampleApplication/system/log1-L	class akka.event.Logging$Info	1339155973
6,21371E+14	null	akka://ExampleApplication/system/deadL	class akka.actor.DeadLetter	1615004858
6,21371E+14	null	akka://ExampleApplication/system/deadL	class akka.actor.DeadLetter	1340833973
6,21371E+14	null	akka://ExampleApplication/system/deadL	class akka.actor.DeadLetter	1296849568
6,21259E+14	akka://ExampleApplication/user/$a	akka://ExampleApplication/user/$a	class simx.core.entity.component.Get	555258148
6,21259E+14	akka://ExampleApplication/user/$a	akka://ExampleApplication/user/$a	class simx.core.entity.component.Get	618738072
6,21259E+14	akka://ExampleApplication/user/$a	akka://ExampleApplication/user/$a	class simx.core.entity.component.Get	1014957330

Abbildung 7.6: Tabellarische Darstellung der durch die Instrumentierung generierten CSV Datei mit den Informationen zu den versandten Nachrichten. Die erste Spalte enthält den Zeitstempel des Versands, die zweite die ID des Senders, die dritte Spalte die ID des Empfängers, die vierte Spalte den Datentyp der Nachricht und die fünfte Spalte eine eindeutige ID der Nachricht.

Neben dieser Information über den Versand wird noch eine weitere Datei geschrieben, die folgende Informationen über die Verarbeitung von Nachrichten beinhaltet:

- Zeitstempel des Beginns der Verarbeitung
- Zeitstempel des Endes der Verarbeitung
- Datentyp der Nachricht
- Eindeutige ID der Nachricht

- Eindeutige ID des Senders
- Eindeutige ID des Empfängers

Die Abbildung 7.7 zeigt hier Beispieldaten in dem vorher definierten Format.

start	end	message type	id	sender	receiver
6,2137E+14	6,2137E+14	class akka.actor.Terminated	269468393	akka://ExampleAppli	akka://ExampleApplication/
6,2137E+14	6,2137E+14	class akka.actor.Terminated	598531729	akka://ExampleAppli	akka://ExampleApplication/system
6,2125E+14	6,2125E+14	class akka.event.Logging$InitializeLogger	1083750747	akka://ExampleAppli	akka://ExampleApplication/system/log
6,2137E+14	6,2137E+14	class akka.event.Logging$Info	1339155973	akka://ExampleAppli	akka://ExampleApplication/system/log
6,2137E+14	6,2137E+14	class akka.actor.DeadLetter	1615004858	akka://ExampleAppli	akka://ExampleApplication/system/de
6,2126E+14	6,2126E+14	class simx.core.entity.component.GetDepe	555258148	akka://ExampleAppli	akka://ExampleApplication/user/$a
6,2126E+14	6,2126E+14	class simx.core.entity.component.GetDepe	618738072	akka://ExampleAppli	akka://ExampleApplication/user/$a
6,2126E+14	6,2126E+14	class simx.core.entity.component.GetDepe	1014957330	akka://ExampleAppli	akka://ExampleApplication/user/$a
6,2126E+14	6,2126E+14	class simx.core.entity.component.GetDepe	532347596	akka://ExampleAppli	akka://ExampleApplication/user/$a

Abbildung 7.7: Tabellarische Darstellung der durch die Instrumentierung generierten CSV Datei mit den Informationen zu den verarbeiteten Nachrichten. Die erste Spalte beinhaltet den Zeitstempel des Beginns der Verarbeitung, die zweite Spalte den Zeitstempel des Endes der Verarbeitung, die dritte Spalte den Typ der Nachricht, die vierte Spalte die eindeutige ID der Nachricht und die letzten Spalten die ID des Senders und des Empfängers der Nachricht.

Die durch die Instrumentierung erzeugten Daten entsprechen somit nicht exakt denen von *MessageRecord* und *ProcessRecord*, können jedoch in dieses Format umgeformt werden.

Um die Kosten der Instrumentierung zu messen, wurde ein kleiner Micro-Benchmark entwickelt. Dieser Micro-Benchmark besteht aus zwei Aktoren, die sich eine Nachricht 1.000.000 mal hin- und her schicken. Der Test wurde auf einem ThinkPad W520 mit einem Intel Core i7-2670QM mit 2,2 GHz und 8 GB RAM auf dem Betriebssystem Windows 10 ausgeführt. Dieser Test wurde sowohl mit als auch ohne die oben beschriebene Instrumentierung ausgeführt. Der Durchlauf ohne Instrumentierung dauerte im Schnitt 0,79 Sekunden. Ein Durchlauf mit Instrumentierung dauerte im Schnitt 2,72 Sekunden. Der Aufwand steigt durch die Instrumentierung somit um den Faktor 3,44.

Aus den gesammelten Daten werden die in Kapitel 4 definierten Metriken mit Hilfe der in Kapitel 5 spezifizierten Algorithmen berechnet.

Visualisierung

In diesem Abschnitt wird beschrieben, wie die drei Metriken im Profiler visualisiert werden.

M1: Nebenläufigkeit Die Metrik **M1** (Nebenläufigkeit) wird im Profiler auf drei verschiedene Arten dargestellt. Zum einen erfolgt, wie in Abbildung 7.8 gezeigt, eine numerische Darstellung der durchschnittlichen Parallelität über die gesamte Laufzeit der Messung. Da beobachtet wurde, dass gerade bei einfachen Anwendungen der Prozess

die meiste Zeit inaktiv (Idle) ist, wird die durchschnittliche Parallelität einmal mit und einmal ohne diese aktive Zeit berechnet. Die Parallelität ohne inaktive Zeit kann somit niemals kleiner als 1.0 sein.

Abbildung 7.8: Darstellung der Metrik **M1** (Nebenläufigkeit) in numerischer Form. Hierbei wird einmal die durchschnittliche gemessene Parallelität gemäß Gleichung 4.10 von Seite 55 sowie die normierte durchschnittliche Parallelität aus Gleichung 4.13 von Seite 56 gezeigt.

Zur besseren Beurteilung der Parallelität wird, inspiriert durch Intel VTune Amplifier, die Parallelität als Histogramm wie in Abbildung 7.9 gezeigt. Hierbei wird dargestellt, wie viele Millisekunden lang wie viele Aktoren gleichzeitig aktiv waren. In der Abbildung 7.9 waren z. B. 3.500 ms lang zwei Aktoren aktiv und ca. 200 ms lang vier Aktoren aktiv.

Neben den vorherigen Darstellungen lässt sich auch der Grad der Parallelität über der Zeit darstellen. Hierbei wird, wie in Abbildung 7.10 gezeigt, der Verlauf des Grades der Parallelität einer Anwendung gezeigt, die zwei Minuten lang lief. Deutlich ist hier die Startphase innerhalb der ersten 30 Sekunden zu erkennen, in der primär Assets von der Festplatte in den Speicher der Grafikkarte geladen werden.

Die Anzeige erlaubt es, den zeitlichen Ausschnitt zu verkleinern, wie in Abbildung 7.11 gezeigt wird. Hierdurch lässt sich genau untersuchen, zu welchem Zeitpunkt welcher Grad der Parallelität in der Anwendung herrschte.

M2: Latenz Wie bereits in Kapitel 5.2.2 beschrieben, wird zur Ermittlung der Latenz zunächst der Pfad einer Information durch die Anwendung benötigt. Abbildung 7.12 zeigt hier den grafischen Editor zur Konfiguration eines solchen Pfades. Hierbei wird zunächst ein Aktor ausgewählt, der eine Information erzeugt. Wenn dieser Prozess einen Simulationsschritt durchführt, kann dieser ebenfalls konfiguriert werden. Darüber hinaus kann konfiguriert werden, über welche Nachricht diese Information an welchen anderen Aktor weitergeleitet wird.

Ist der Pfad vollständig konfiguriert, kann über die Betätigung des Knopfes *Discover Latency* die Latenz auf diesem Pfad ermittelt werden. Hierfür wird der im Kapitel 5.2.2 spezifizierte Algorithmus verwendet. Das Ergebnis wird in einem eigenen Fenster wie in Abbildung 7.13 dargestellt.

In diesem Fenster wird zunächst die Latenz auf diesem Pfad numerisch dargestellt. Hierbei wird sowohl die ermittelte minimale und maximale Latenz genannt, als auch der Durchschnitt und der Median. Da insbesondere das Laden von Assets zu längeren Verzögerungen in der Latenz führt, ist der Median der aussagekräftigere Wert. Neben diesen numerischen Informationen wird in einem Graphen dargestellt, wie sich die Latenz über die Zeit verändert hat. In der Abbildung 7.13 ist beispielsweise zu sehen, dass die Latenz nach 25 Sekunden für die nächsten 10 Sekunden stark ansteigt und schwankt.

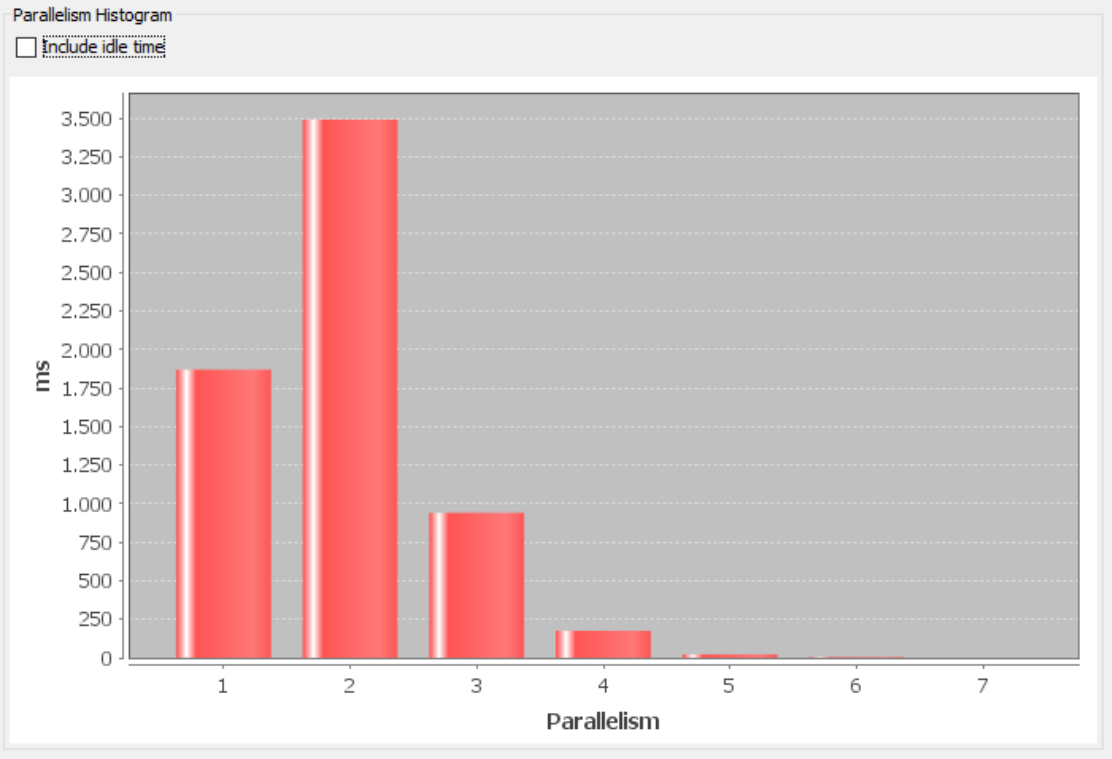


Abbildung 7.9: Darstellung der Metrik **M1** (Nebenläufigkeit) in Form einer Histogramms. Hierbei wird dargestellt, wie lange der jeweilige Grad der Parallelität („Parallelism“) vorlag. In dieser Abbildung herrschte in der gemessenen Anwendung für ca. 3,5 Sekunden ein Grad der Parallelität von 2.

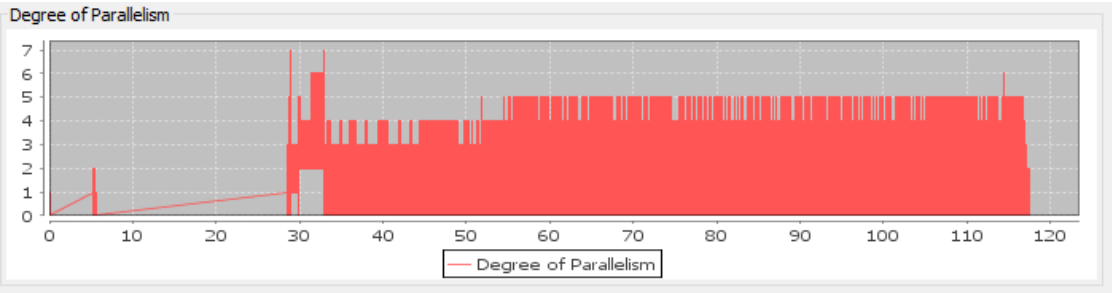


Abbildung 7.10: Darstellung der Metrik **M1** (Nebenläufigkeit) als Grad der Parallelität in Abhängigkeit von der Zeit (in Sekunden). Die dargestellte Anwendung lief 120 Sekunden, wobei der Grad der Parallelität in der Zweit zwischen 0 und 7 schwankte.

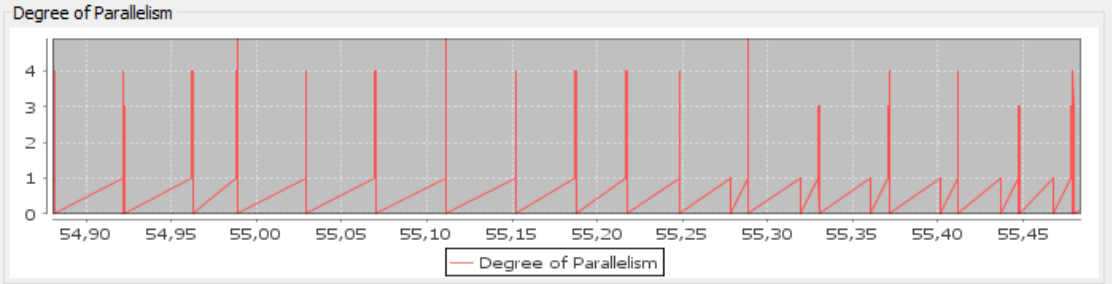


Abbildung 7.11: Darstellung des Grades der Parallelität über der Zeit (in Sekunden), wie in Abbildung 7.10, jedoch ein kleinerer Zeitausschnitt.

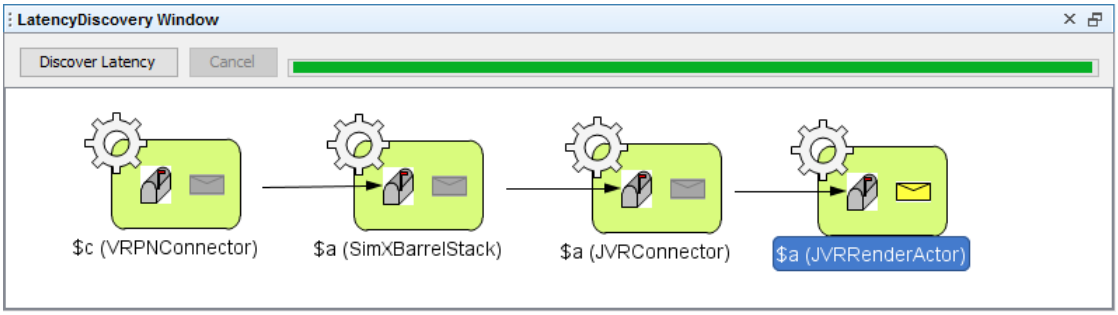


Abbildung 7.12: Darstellung des Fensters, in dem der Pfad einer Information, zur Bestimmung der Latenz, beschrieben wird. Der hier dargestellte Pfad stellt die Latenz zwischen einer von der VRPN-Anbindung bereitgestellten Kopfposition des Benutzers bis zum Abschluss des Renderings eines neuen Frames dar.

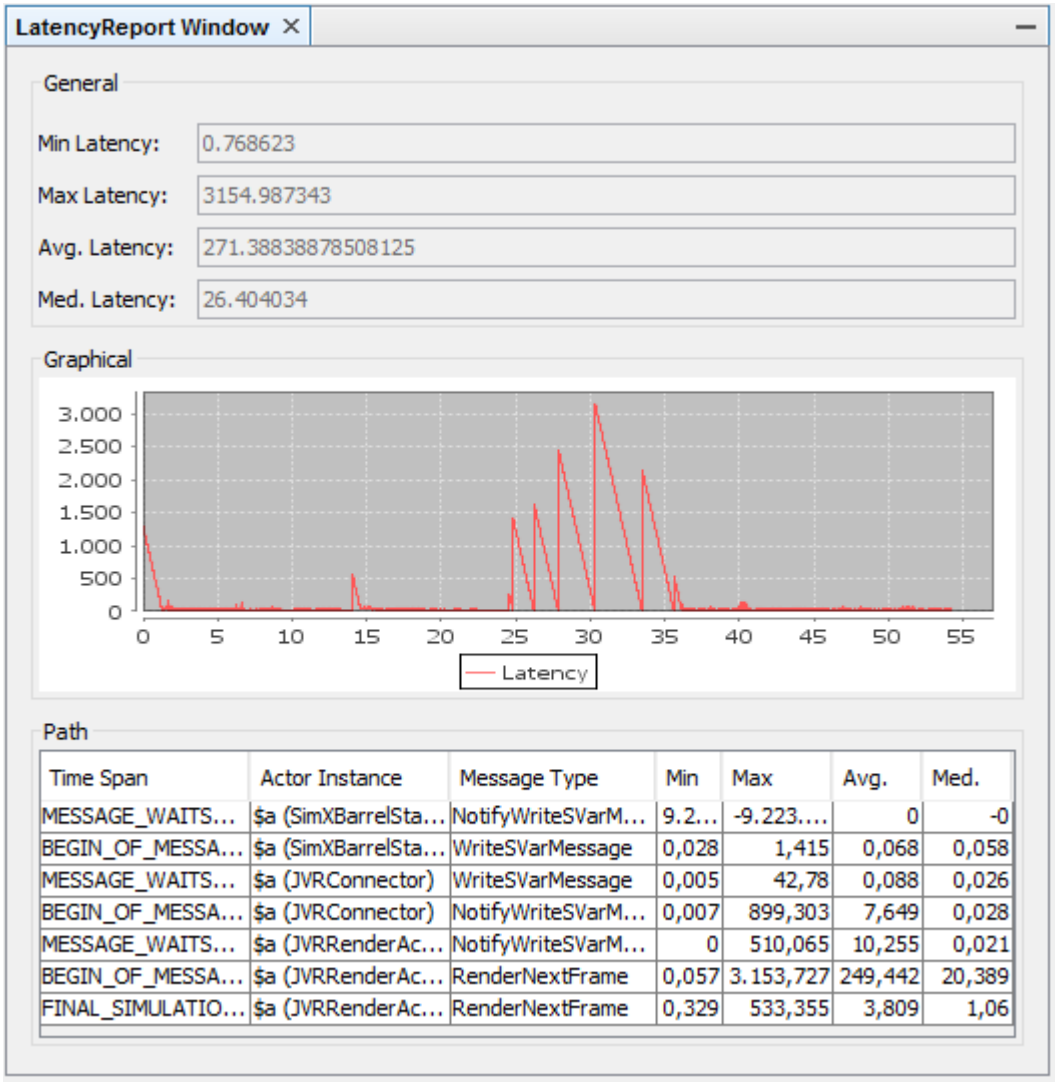


Abbildung 7.13: Darstellung der Metrik **M2** (Latenz). Die Latenz wurde aus Messdaten und entlang eines Pfades, wie in Abbildung 7.12, ermittelt. Zuerst werden die minimale, maximale und durchschnittliche Latenz sowie der Median angegeben. Der darunterliegende Graph bildet die Änderung der Latenz über die Laufzeit der Anwendung ab. Die Tabelle zeigt die Latenz der einzelnen Abschnitte des Pfades gemäß der Identifizierten Zeitabschnitte aus Tabelle 5.2 von Seite 66.

Abschließend wird in einer Tabelle dargestellt, welche einzelnen Abschnitte des Pfades mit wie viel Latenz zum gesamten Pfad beitragen. In der ersten Spalte wird der Typ der Zeitspanne, wie in Kapitel 7.13 spezifiziert, genannt. Anschließend wird die konkrete Instanz eines Aktors genannt, bei dem diese Latenz auftritt. In der nächsten Spalte wird der Typ der Nachricht genannt, um die es geht. In den letzten vier Spalten wird die minimale, maximale sowie der Durchschnitt und der Median der Latenz dieses Abschnitts genannt. Insbesondere diese Tabelle ist hilfreich, um Latenzen im Detail ausfindig zu machen.

M3: Konsistenz Für die Ermittlung der Konsistenz werden zwei Aktoren benötigt. Sind diese zwei Aktoren ausgewählt, werden darüber hinaus der Typ der Simulationsschrittnachricht des ersten Aktors benötigt, der Datentyp der Nachricht, der die Information zum zweiten Aktor transportiert, und der Typ der Simulationsschrittnachricht des zweiten Aktors. Die Konfiguration erfolgt über die in Abbildung 7.14 gezeigte Tabelle.

Anschließend kann über den Knopf *Discover Consistency* die Untersuchung der Konsistenz gestartet werden. Hierbei wird der im Abschnitt 5.2.3 spezifizierte Algorithmus ausgeführt. Das Ergebnis wird, wie in Abbildung 7.14 gezeigt, in der letzten Spalte der Tabelle dargestellt.

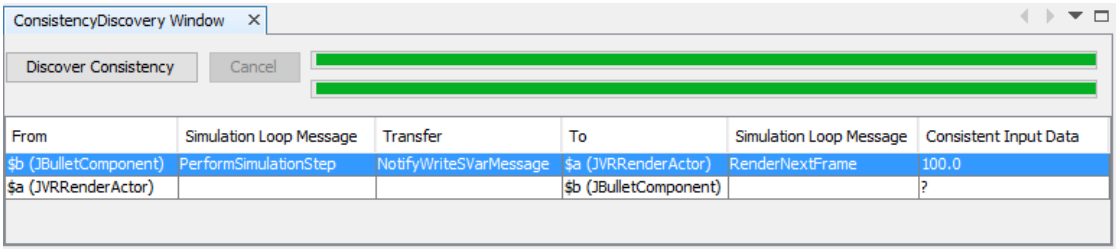


Abbildung 7.14: Darstellung der Metrik **M3** (Konsistenz). In diesem Beispiel wird die Konsistenz zwischen Physik-Simulation und Rendering untersucht. Hierbei wird konfiguriert, welche Nachricht beim jeweiligen Aktor einen neuen Weltzustand erzeugt und durch welche Nachricht ein neuer Weltzustand übertragen wird. In der letzten Spalte wird angegeben, in wie viel Prozent der Fälle der Renderer mit einem konsistenten Weltzustand der Physik-Simulation gearbeitet hat.

7.2.3 Model Checker

In diesen Abschnitt wird die Implementierung des Model Checking Tools beschrieben. Zunächst wird der Auswahlprozess zum verwendeten Model Checker dargestellt. Anschließend wird die erstellte Toolchain beschrieben.

Auswahl des Model Checking Tools

Im folgenden werden neun formale Spezifikationssprachen und die dazugehörigen Model Checker verglichen. Ziel ist es, eine formale Spezifikationssprache zu finden, mit deren Hilfe aus einer Spezifikation Vorhersagen zu den Metriken **M1**–**M3** getroffen werden können.

Kriterien Für den Vergleich werden zwei verschiedene Gruppen von Kriterien angewandt. Die spezifischen Kriterien beziehen sich explizit auf den hier definierten Anwendungsfall. Die allgemeinen Kriterien behandeln Sachverhalte, die die Benutzung der formalen Spezifikationssprache und dessen Model Checker einfacher machen. Die spezifischen Kriterien erhalten in der Endauswertung eine doppelte Gewichtung gegenüber den allgemeinen Kriterien. Die Ergebnisse des Vergleichs sind in Tabelle 7.1 auf Seite 109 zusammengefasst.

Allgemeine Kriterien

AK1: Dokumentation Eine gute Dokumentation vereinfacht die Verwendung der formalen Spezifikationssprache und des dazugehörigen Model Checkers. Wenn zu einer formalen Spezifikationssprache ein Buch existiert, erhält diese die Bewertung ++. Dagegen erhält sie die Bewertung +, wenn mehrere Publikationen von verschiedenen Autoren existieren. Die Bewertung 0 wird vergeben, wenn lediglich Publikationen vom ursprünglichen Autor der Spezifikationssprache existieren.

AK2: Verbreitung Wenn eine formale Spezifikationssprache eine weite Verbreitung hat, deutet dies darauf hin, dass sie und der dazu gehörige Model Checker einen reifen und stabilen Zustand erreicht haben, der die Verwendung in echten Projekten ermöglicht. Darüber hinaus kann angenommen werden, dass bei einer hohen Verbreitung Vorlagen für übliche Probleme existieren. Die Bewertung ++ wird vergeben, wenn die formale Sprache in Projekten außerhalb des akademischen Forschungsumfeldes eingesetzt wurde. Die Bewertung + wird vergeben, wenn die formale Spezifikationssprache von verschiedenen Forschungsgruppen an mehreren Instituten eingesetzt wurde. Die Bewertung −− wird vergeben, wenn lediglich der Autor der Sprache sie verwendet hat.

AK3: Implementierung Dieses Kriterium bildet ab, ob eine öffentlich zugänglich Implementierung eines Model Checkers verfügbar ist. Die Bewertung ++ wird vergeben, wenn eine Open-Source-Implementierung verfügbar ist. Dagegen wird ein + vergeben, wenn eine proprietäre Implementierung existiert. Ist keine öffentlich zugängliche Implementierung verfügbar, gilt als Wertung −−.

Spezifische Kriterien

SK1: Echtzeit Um das Echtzeitverhalten eines Interaktiven Echtzeitsystems abbilden zu können, ist es vorteilhaft, wenn die Spezifikation von Echtzeitsystemen von einer formalen Spezifikationssprache möglichst gut unterstützt wird. Die Bewertung ++ wird vergeben, wenn die Sprache selbst Konstrukte für die Spezifikation von Echtzeitsystemen beinhaltet. Dagegen wird die Bewertung + vergeben, wenn dies Teil einer Standardbibliothek der Sprache ist. Müssen Konstrukte für Echtzeitsysteme vollständig selbst spezifiziert werden, erhält die Sprache die Bewertung 0.

SK2: Message-Passing Die gewählte Zielplattform Simulator X ist vollständig auf Message-Passing basierend. Daher ist es von Vorteil, wenn die formale Spezifikationssprache bereits die Spezifikation von Message-Passing-basierten Systemen unterstützt. Die Bewertung ++ wird vergeben, wenn dies Bestandteil der Sprache selbst ist. Dagegen wird die Bewertung + vergeben, wenn dies nicht direkt Bestandteil der Sprache ist, jedoch wiederverwendbare Spezifikationen existieren. Die Bewertung 0 wird vergeben, wenn eine Message-Passing Spezifikation erstellt werden kann. Wenn es bekannte Probleme mit der Spezifikation von Message-Passing Systemen gibt, erhält die Sprache die Bewertung −−.

SK3: Integration in andere Tools Um Model Checking in den Prozess der Anwendungsentwicklung von Simulator X zu integrieren, muss sich der Model Checker in die aktuelle Toolchain einfügen lassen. Wie bereits in Abschnitt 7.1 beschrieben, wurde Simulator X in Scala implementiert. Scala erzeugt hierbei Byte-Code für die Java Virtual Machine. Von daher wird die Bewertung ++ vergeben, wenn der Model Checker in Java geschrieben ist und 0 wenn er in einer anderen Programmiersprache implementiert ist.

Kandidaten Im folgenden werden die neun Kandidaten fürs Model Checking vorgestellt.

TLA+ TLA+ (Temporal Logic of Actions) ist eine formale Sprache für die Spezifikation von nebenläufigen Systemen (Lamport, 2002). Sie basiert grundlegend auf Temporaler Logik, bietet darüber aber auch die Möglichkeit, Aktionen zu definieren. Ein System wird in TLA+ als eine Menge von Konstanten, Variablen und Aktionen definiert. Die Beschreibung einer Aktion beinhaltet die Vorbedingungen, die für die Ausführung der Aktion erfüllt sein müssen, sowie die Änderungen am Zustand des Systems durch diese Aktion. Das Gesamtsystem mit all seinen Aktionen wird in der Regel unter der Bezeichnung *Spec* zusammengefasst. Für die Überprüfung des Modells wird ein `THEOREM` definiert, das üblicherweise aus der Spezifikation folgt, dass eine bestimmte temporale Logik erfüllt ist. Aktionen werden in TLA+ nicht imperativ sondern deklarativ beschrieben.

TLA+ ist als Sprache gut dokumentiert, unter anderem durch ein Buch (Lamport, 2002), einer Online-Dokumentation (Lamport, 2015) und mehreren Veröffentlichungen.

TLA+ ist weit verbreitet, was unter anderem daran zu erkennen ist, dass spezielle Veranstaltungen zu TLA+ auf Konferenzen stattfinden, wie das *TLA+ Community Event* auf der *ABZ 2014* oder der *International Workshop on the TLA+ Method and Tools* auf der *FM 2012*. Darüber hinaus wird TLA+ in Entwicklungsprozessen bei *Amazon Web Services* verwendet (Newcombe et al., 2015).

TLA+ besitzt von sich selbst aus keine Semantik für Message-Passing-basierte Systeme. Diese muss vollständig selbst spezifiziert werden. In TLA+ ist dies jedoch nicht vollständig unabhängig vom konkreten Anwendungsszenario möglich, wodurch in TLA+ selbst keine abstrakte wiederverwendbare Spezifikation eines Message-Passing Systems möglich ist. Aus diesem Grund existieren auf TLA+ aufbauende Systeme wie cTLA und ReActor. Die Spezifikation eines Algorithmus in einer Message-Passing Umgebung ist somit mit erheblichen Aufwand verbunden (Burmeister, 2013; Burmeister und Helke, 2012).

Für TLA+ steht ein Model Checker mit dem Namen TCL zur Verfügung. Dieser unterstützt hierbei TLA+ nicht vollständig (Lamport, 2002), jedoch ausreichend, um für den praktischen Einsatz verwendet werden zu können. TLC ist Bestandteil einer „Toolbox“, die neben dem Model Checker noch einen Editor und weitere Tools beinhaltet (Lamport, 2016).

TLC ist in Java implementiert und quelloffen. Der Model Checker selbst lässt sich in der Kommandozeile starten oder auch in eigene Tools integrieren.

Für die Spezifikation von Echtzeitsystemen existieren neben eines Beispiels von Lamport (2002) erweiterte Konzepte wie von Zhang et al. (2010). Dass TLA+ zur Spezifikation und Überprüfung umfangreicher Echtzeitsysteme geeignet ist, wurde von Faria (2008) beschrieben, wo Aspekte eines nebenläufigen Echtzeitbetriebssystems in TLA+ spezifiziert und mit TLC überprüft wurden.

PlusCal Die Sprache PlusCal wurde bereits zu Beginn des Kapitels 5 kurz beschrieben. Dort wurde es verwendet, um Algorithmen zur Berechnung von Metriken eindeutig zu spezifizieren. In diesem Abschnitt wird PlusCal als möglicher Kandidat für die Anbindung eines Model Checkers evaluiert.

PlusCal setzt auf TLA+ auf, somit können in PlusCal spezifizierte Algorithmen durch in TLA+ geschriebene Ausdrücke erweitert werden. Darüber hinaus können durch einen Konverter in PlusCal geschriebene Spezifikationen in eine TLA+ Spezifikation überführt werden. Somit können in PlusCal spezifizierte Algorithmen durch TLC auf ihre Richtigkeit überprüft werden.

PlusCal ist wie TLA+ sehr gut dokumentiert. Für beide Syntaxvarianten existieren umfangreiche Dokumentationen (Lamport, 2012, 2013). Darüber hinaus wird die Anwendung von PlusCal in technischen Berichten beschrieben (Lamport, 2006b).

Wie TLA+ wird PlusCal in der kommerziellen Software-Entwicklung eingesetzt (Newcombe et al., 2015).

Wie auch TLA+ verfügt PlusCal über keine vorgefertigte Spezifikation von Message-Passing, was somit selbst spezifiziert werden muss. Da in PlusCal TLA+ Konstrukte verwendet werden können, ist auch die Verwendung von bestehenden Spezifikationen für Echtzeitsysteme möglich. Der Konverter von PlusCal in TLA+ ist in Java implementiert und quelloffen, lässt sich somit also leicht an andere Tools anbinden.

cTLA+ Compositional TLA+ (cTLA+) ist eine auf TLA+ aufbauende Sprache. Sie erweitert TLA+ um die explizite Definition von Prozessen (Graw et al., 2000). Der

Aufbau einer in cTLA+ erstellten Spezifikation eines Prozesses lehnt sich hierbei am Konzept der objektorientierten Programmierung an. Zu cTLA+ existieren eine Reihe von Veröffentlichungen aus den 1990er und frühen 2000er Jahren. Eine Implementierung steht nicht öffentlich zur Verfügung. Somit kann auch keine Anbindung an andere Tools erfolgen. Die Strukturierung durch cTLA+ würde eine Spezifikation im Anwendungsfall dieser Arbeit u. U. erleichtern, jedoch müsste ein Message-Passing immer noch vollständig selbst entwickelt werden. Die Spezifikation für Echtzeitverhalten setzt auf das Beispiel aus TLA+ auf (Herrmann und Krumm, 1997).

ReActor ReActor wurde aus der Erkenntnis heraus entwickelt, dass die Spezifikation von Message-Passing-basierten Algorithmen in TLA+ aufwendig ist und dass das Erstellen einer vom konkreten Anwendungsbeispiel unabhängigen Spezifikation eines Message-Passing Systems nicht möglich ist (Burmeister, 2013; Burmeister und Helke, 2012). Hierfür wird TLA+ erweitert, um Prozesse zu spezifizieren, zu instanziieren und Nachrichten zu versenden. Aus einer ReActor Spezifikation wird durch einen Konverter eine TLA+ Spezifikation erzeugt. Diese lässt sich wiederum mit TLC überprüfen. Eine Implementierung von ReActor ist nicht öffentlich verfügbar. Von daher hat ReActor keine Verbreitung. Ebenso ist keine Anbindung an andere Tools möglich. Als Dokumentation stehen nur einige wenige Veröffentlichungen zur Verfügung. Eine Unterstützung von Spezifikation von Echtzeitsystemen wird nicht explizit genannt. Wäre eine Implementierung von ReActor verfügbar, wäre eine Spezifikation für den definierten Anwendungsfall weniger aufwendig als direkt in TLA+.

Alloy Alloy (Jackson, 2002) ist eine deklarative Sprache zur Modellierung von Systemen. Im Gegensatz zu TLA+ und Derivaten ist die Syntax nicht an die Mathematik sondern an objektorientierte Programmiersprachen angelehnt. Durch den Syntax sollten drei verschiedene Verständnisebenen ermöglicht werden. Die höchste Abstraktion ist es, ein Modell in Alloy als OOP Modell zu verstehen. Hierdurch soll es auch ohne tieferes Verständnis von Alloy möglich sein, ein Modell zumindest in Grundzügen zu verstehen. In der mittleren Ebene wird ein Alloy Modell über die Mengenlehre aufgefasst. Laut Dokumentation sollte dies die bevorzugte Ebene sein, in der Anwender von Alloy denken sollten. Auf unterster Abstraktionsebene besteht ein Modell aus atomaren Entitäten und den Beziehungen dieser Entitäten untereinander.

Zu Alloy gibt es zahlreiche wissenschaftliche Publikationen, ein Buch (Jackson, 2012b) sowie Online-Tutorials (Jackson, 2012c). Alloy wurde durch eine Vielzahl von Projekten zur Verifikation verwendet (Bajić-Bizumić et al., 2013; Newcombe, 2011; Wegmann et al., 2007). Sowohl Message-Passing als auch Echtzeitverhalten sind nicht standardmäßig enthalten und müssen vollständig selbst spezifiziert werden.

Ein implementierter Model Checker ist frei verfügbar. Dieser ist vom API her so ausgelegt, dass er an andere Tools angebunden werden kann, wofür auf der Homepage von Alloy auch Beispielcode gezeigt wird (Jackson, 2012a).

Rebeca Rebeca (Sirjani et al., 2004) wurde mit dem Ziel entwickelt, ein praxistaugliches Tool zu bieten, was die Lücke zwischen formalen Methoden und dem Software Engineering schließt (Reynisson et al., 2014). Im Gegensatz zu TLA+, cTLA+ und ReActor erfolgt die Spezifikation von Systemen imperativ. Der Syntax ist an Java angelehnt. Mit dem Schlüsselwort `reactiveclass` werden Vorlagen für Prozesse definiert. Hierbei wird die maximale Anzahl der wartenden Nachrichten in der Mailbox angegeben. Eine `reactiveclass` enthält einen Abschnitt `knownrebecs`, der Verweise auf andere Prozesse enthält. Im Abschnitt `statevars` werden Variablen des Prozesses definiert. Mit dem Schlüsselwort `msgsrv` werden Nachrichten definiert, die an diesen Prozess gesendet werden können. Diese verändern üblicherweise den internen Zustand eines Prozesses. Neben der Definition der `reactiveclass`es enthält eine Rebeca-Spezifikation eine main-Definition, in der aus den Klassen die Definition konkreter Prozesse erfolgt und darüber hinaus die Kommunikation zwischen den Prozessen deklariert wird.

Zu Rebeca existiert ein Referenzhandbuch (Hojiat und Kaviani, 2012), welches jedoch veraltet ist. Neben dem Referenzhandbuch existieren eine Vielzahl von Veröffentlichungen, die den aktuellen Syntax von Rebeca darstellen. Durch die an Java angelehnte Syntax ist die Verwendung jedoch einfach.

Die Spezifikation des Anwendungsfalls ist mit Rebeca leicht möglich, da grundlegende Konzepte ähnlich wie in der gewählten Plattform sind.

Es existieren mehrere erweiterte Derivate von Rebeca. *TimedRebeca* (Reynisson et al., 2014) erweitert Rebeca um drei weitere Kommandos zur Spezifikation von Echtzeitsystemen. *ProbabilisticRebeca* erweitert Rebeca um die Möglichkeit, für Variablen keine konkreten Werte sondern Wahrscheinlichkeiten für Werte anzugeben. *ProbabilisticTimedRebeca* (Jafari et al., 2014) kombiniert beide Derivate.

Mit RMC (Rebeca Model Checker) (Khamespanah, 2014) existiert ein in Java implementierter quelloffener Model Checker. RMC erzeugt aus der in Rebeca geschriebenen Spezifikation den C++ Quelltext eines Model Checkers für diese Spezifikation. RMC ist fähig, ein in Rebeca spezifiziertes System auf in LTL (Pnueli, 1977) definierte Eigenschaften zu prüfen. Darüber hinaus kann RMC den State Space als XML exportieren. Eine Anbindung von RMC ist somit leicht möglich.

Promela Promela (Process Meta Language) (Holzmann, 1997) ist eine imperative Sprache zur Beschreibung von nebenläufigen Systemen. In Promela beschriebene Systeme werden mit dem dazugehörigen Model Checker Spin überprüft. Promela ist syntaktisch an die Programmiersprache C angelehnt, besitzt aber eine Vielzahl von Erweiterungen für das Model Checking. Nebenläufige Prozesse werden mit dem Schlüsselwort `proctype` definiert. Ein Prozess kann globale und lokale Variablen lesen und schreiben sowie andere Prozesse erzeugen. Promela besitzt einige syntaktische Besonderheiten, um einen Prozess auf Wertänderungen von Variablen warten zu lassen. Promela und Spin wurden für eine Reihe von Projekten verwendet, insbesondere in der Raumfahrt (Schneider et al., 1998; Holzmann, 2014).

Promela ist sehr gut durch wissenschaftliche Veröffentlichungen (Holzmann, 1997), Bücher (Holzmann, 2003; Ben-Ari, 2008) und eine Online-Dokumentation sowie Tutorials

dokumentiert. In PROMELA kann die Kommunikation von spezifizierten Prozessen über Message-Passing erfolgen. Die Spezifikation von Echtzeitsystemen wird in PROMELA und SPIN selbst nicht unterstützt (Ruys und Holzmann, 2004). SPIN ist quelloffen verfügbar. Die Anbindung von SPIN an andere Tools ist mit erhöhtem Aufwand verbunden, da die Ausgaben von SPIN nicht wohldefiniert und leicht maschinenlesbar sind (Ben-Ari, 2010).

Uppaal Im Gegensatz zu den vorher vorgestellten Model Checkern wird bei UPPAAL (Larsen et al., 1997) das Modell eines nebenläufigen Systems über einen grafischen Editor definiert. Durch diesen grafischen Editor werden Zustände und Übergänge definiert. Neben der grafischen Definition werden in einer imperativen Sprache Variablen und Funktionen in einer C-ähnlichen Syntax definiert. Diese Funktionen können zu Kanten der grafischen Definition zugeordnet werden. UPPAAL ist fähig, das Modell auf bestimmte Eigenschaften zu überprüfen und den State Space schrittweise zu durchlaufen. UPPAAL ist nicht quelloffen. Für akademische Nutzung wird keine Lizenzgebühr erhoben.

UPPAAL ist durch ein umfangreiches Tutorial dokumentiert (Behrmann et al., 2006). Die Definition von Echtzeitsystemen wird unterstützt. Message-Passing hingegen muss selbst definiert werden, was nicht trivial ist, da die Message Queue für asynchrones Message-Passing leicht zu einer Explosion des State Spaces führen kann (Kristinsson et al., 2013). Bei der Implementierung ist der Editor vom Model Checker getrennt. Der Model Checker läuft in einem eigenen Prozess und kommuniziert über TCP/IP mit dem grafischen Editor. Aufgrund der geschlossenen Quellen ist eine Anbindung an andere Tools schwierig.

McErlang McErlang (Fredlund und Svensson, 2007) ist ein Model Checker für in Erlang (Larson, 2008; Armstrong und Ab, 1997) geschriebene Software. Im Gegensatz zu den anderen vorgestellten Systemen werden bei McErlang die zu überprüfenden Spezifikationen nicht in einer separaten Spezifikationssprache, sondern in Erlang selbst geschrieben. McErlang unterstützt hierbei einen Großteil der Sprachkonstrukte aus Erlang. McErlang selbst besteht im Endeffekt aus einer modifizierten Erlang-Run-Time, welche eine Erlang-Anwendung nicht ausführt, sondern ein Model Checking durchführt.

Zu McErlang existiert ein Handbuch (Fredlund und Earle, 2011) und ein Tutorial (Earle und Fredlund, 2010). Mit Hilfe von McErlang wurden einige verteilte Systeme überprüft (Earle und Fredlund, 2009; Fredlund und Sánchez Penas, 2007). Message-Passing ist ein integraler Bestandteil der Programmiersprache Erlang und wird somit durch McErlang unterstützt. Auch die Überprüfung von Echtzeitverhalten ist mit McErlang möglich (Earle und Fredlund, 2012). Eine Implementierung von McErlang ist quelloffen verfügbar. Eine Anbindung an andere Tools ist möglich, wie in (Kristinsson et al., 2013) gezeigt wurde.

Auswertung und Auswahl Tabelle 7.1 fasst die Ergebnisse des Vergleichs nach den oben genannten Kriterien zusammen. Wie erkennbar ist, eignet sich für den definierten Anwendungsfall die Sprache *Rebeca* in Kombination mit dem Model Checker *RMC* am besten. Während also zur Spezifikation der Berechnungsalgorithmen in den Kapiteln 5

und 6 PlusCal verwendet wurde, eignet sich diese Sprache nicht als Grundlage für die Implementierung der Model-Checking-Anbindung.

Tabelle 7.1: Vergleich von neun formalen Spezifikationssprachen und ihren Model Checkern für die Schätzung der Nebenläufigkeit, Latenz und Konsistenz in eine asynchronen Interaktiven Echtzeitsystem.

	TLA+	PlusCal	cTLA	ReActor	Alloy	Rebeca	Promela	Uppaal	McErlang
Typ	dec.	dec.	dec.	dec.	dec.	imp.	imp.	imp.	dec.
AK1: Dokumentation	++	++	+	0	++	+	++	++	+
AK2: Verbreitung	++	++	−−	−−	++	+	++	++	+
AK3: Implementierung	++	++	−−	−−	++	++	++	+	++
SK1: Echtzeit	+	+	++	−−	0	++	−−	++	++
SK2: Message-Passing	−−	−−	+	++	0	++	++	0	++
SK3: Integration	++	++	−−	−−	++	++	0	++	0
Punkte	8	8	-1	-4	10	16	6	3	12

Erweiterungen an RMC

Im Rahmen dieser Arbeit wurden Erweiterungen an RMC, dem Rebeca Model Checker, vorgenommen. Diese werden in diesem Abschnitt beschrieben.

Wahrung der Reihenfolge von Nachrichten in TimedRebeca Rebeca behandelt die Verarbeitung von Nachrichten als atomare Operation. Folglich stellt jede Verarbeitung einer Nachricht eine Kante im State Space dar. Sind in der Mailbox eines Prozesses mehrere Nachrichten, stehen mehrere Nachrichten für die Verarbeitung zur Auswahl. Somit gibt es von einem Zustand, in dem ein Prozess mehrere Nachrichten in der Mailbox hat, mehrere Kanten zu weiteren Zuständen.

In TimedRebeca verfügt jeder Prozess über eine eigene Uhr und jede Nachricht hat darüber hinaus einen Zeitstempel, die den Zeitpunkt seiner Ankunft darstellt. Dies ist in der Regel die aktuelle Zeit des versendenden Prozesses zum Zeitpunkt des Versands.

Hat ein Prozess mehrere Nachrichten in der Mailbox, behandelt RMC diese unterschiedlich, je nachdem ob deren Ankunft vor oder nach der aktuellen Zeit des Prozesses liegt. Ist der Zeitstempel der Nachricht größer als der aktuelle Wert der Uhr des Prozesses, ist die Reihenfolge der Nachrichten gewahrt. Liegen hingegen die Zeitstempel der Nachricht in der Vergangenheit, wodurch der Wert der Uhr also größer als der Zeitstempel der Nachrichten ist, behandelt RMC dies so, als wären Nachrichten in der Mailbox mögliche Alternativen in der Verarbeitung.

Dies entspricht jedoch nicht dem Verhalten der durch Simulator X verwendeten Aktor-Implementierung. Diese wahrt immer die zeitliche Reihenfolge von Nachrichten. Darüber hinaus wird der State Space durch diese Behandlung der Nachrichten unnötig groß.

Zur Lösung wurde RMC durch eine Option erweitert, damit der Model Checker in jedem Fall die zeitliche Reihenfolge der Nachrichten beachtet und somit nur noch Nachrichten mit gleichem Zeitstempel als Alternativen ansieht.

Export der Verarbeitungszeit als Information einer Transition im State Space Wie bereits im Abschnitt über Rebeca beschrieben exportiert RMC den generierten State Space als XML-Datei. Für diesen Export wird die Zeitdauer der Verarbeitung der Nachricht zu den Kanten hinzugefügt.

Toolchain

Rebeca ist zwar syntaktisch an Java angelehnt, verfügt jedoch nicht über Abstraktionstechniken wie Vererbung oder Generics. Bei der Erzeugung eines Rebecs können einzig Parameter eines Konstruktors verwendet werden, um das Verhalten zu beeinflussen. Darüber hinaus müssen bei der Erzeugung des Rebecs bereits alle Kommunikationspartner des Rebecs bekannt sein.

In Abschnitt 3.2 wurden einige Grundelemente eines Synchronisationskonzeptes definiert. Hierbei können jedoch nicht alle Kommunikationspartner vorher bekannt sein. Um dieses Problem zu lösen, wurde die in Abbildung 7.15 dargestellte Toolchain implementiert, die im folgenden beschrieben wird.

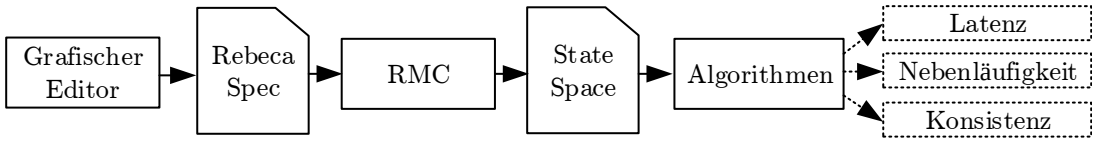


Abbildung 7.15: Toolchain fürs Model Checking.

Grafischer Editor Um die fehlenden Abstraktionsmechanismen von Rebeca auszugleichen und dem Endanwender eine einfache Erstellung von Synchronisationsschemata zu ermöglichen, die aus den in Kapitel 3.2 vorgestellten Grundelementen bestehen, wurde ein grafischer Editor entwickelt. Dieser in Abbildung 7.16 gezeigte Editor wurde auf Basis von NetBeans RCP (NetBeans, 2016) entwickelt.

Der Benutzer kann aus einer Tabelle die vorher vorgestellten Grundelemente auswählen. Anschließend kann er die Kommunikation zwischen den einzelnen Elementen konfigurieren und das Verhalten jedes einzelnen Elements genau einstellen.

Rebeca Spezifikation Aus dem grafischen Modell wird eine Spezifikation in Rebeca generiert. Im folgenden werden Beispiele zu den einzelnen Grundelementen erläutert.

GE1: Unbewegter Beweger Der Unbewegte Beweger hat wie vorher beschrieben lediglich die Aufgabe, initial eine Nachricht an eine Menge von anderen Prozessen, in

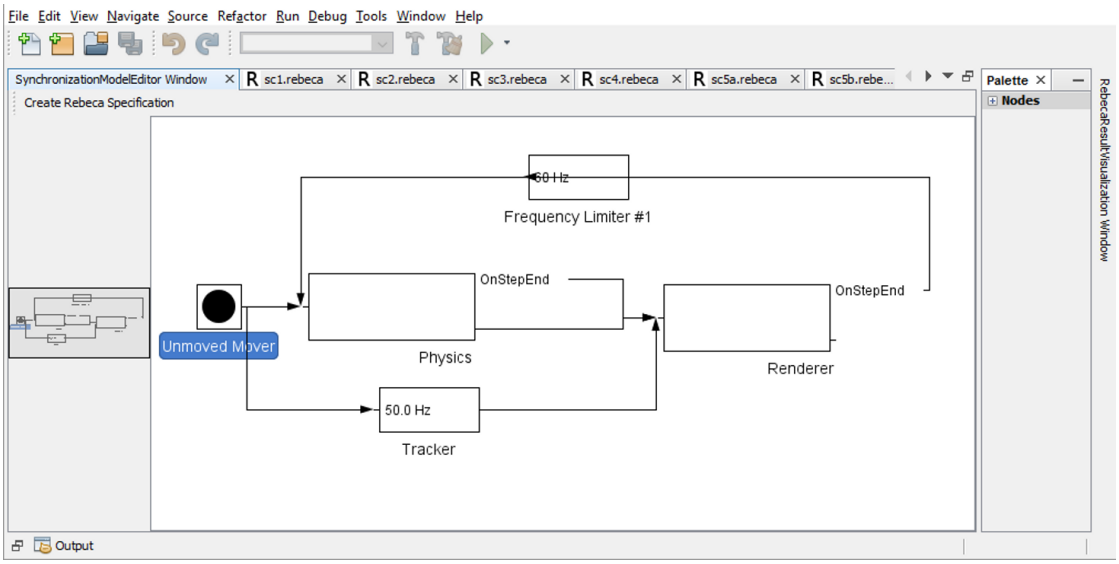


Abbildung 7.16: Grafischer Editor zur Erstellung und Konfiguration von Synchronisationsschemata. In dieser Darstellung ist das **Schema 1** zu sehen.

diesem Fall anderen Rebecs, zu schicken. Wie schon gesagt muss die Menge der bekannten Kommunikationspartner vorab feststehen, weshalb in Listing 7.9 der Inhalt des Abschnitts `knownrebes` und der Code im Message Server generiert sind.

Listing 7.9: Rebeca-Spezifikation für **GE1** (Unbewegter Beweger)

```
1  reactiveclass UnmovedMover( 1 ) {
2          knownrebecs {
3                  Physics physics;
4                  Renderer renderer;
5          }
6          statevars {
7                  int rebectype;
8          }
9          UnmovedMover() {
10                 rebectype = 0;
11                 self.Simulate();
12         }
13         msgsrv Simulate() {
14                 physics.Simulate();
15                 delay( 1 );
16                 renderer.Simulate();
17                 delay( 1 );
18         }
19 }
```

GE2: Sub-System Ein Sub-System führt Simulationsschritte aus, kommuniziert neue Weltzustände und pflegt neue Informationen in den eigenen Weltzustand ein. Jedes Sub-System hat somit eine Simulationsschrittnachricht, welche in Listing 7.10 *Simulate* heißt. In dieser werden Simulationsschrittnachrichten an andere Rebecs versandt und darüber hinaus Nachrichten zu einem neuen Weltzustand verschickt, welche in diesem Beispiel *NotifyWrite* heißen. Hierbei wird ein Zähler mitgeschickt, der angibt, auf welche Entität sich die Nachricht bezieht. Andere Sub-Systeme pflegen Zähler, um festzustellen, welches die letzte Entität war zu der sie eine Nachricht erhielten.

Listing 7.10: Rebeca-Spezifikation für **GE2** (Sub-System)

```
reactiveclass SubSystem( 10 ) {
        knownrebecs {
                Peer a;
                Peer b;
        }
        statevars {
                int rebectype;
                int aCounter;
                int aCounterExpected;
                int bCounter;
                int bCounterExpected;
                boolean dataMessageTypeNotifyWrite;
                boolean simulationStepMessageTypeSimulate;
                boolean namePhysics;
        }
        SubSystem() {
                rebectype = 5;
                aCounter = 0;
                aCounterExpected = 9;
                bCounter = 0;
                bCounterExpected = 5;
                dataMessageTypeNotifyWrite = true;
                simulationStepMessageTypeSimulate = true;
                namePhysics = true;
        }
        msgsrv Simulate() {
                a.Simulate() // OnStepBegin
                for( int i = 0; i < 10; i++ ) {
                        delay( 200 );
                        b.NotifyWrite( i );
                        delay( 1 );
                }
                b.Simulate(); // OnStepEnd
        }

        msgsrv NotifyWrite( int c ) {
                if( sender == a ) aCounter++;
                if( sender == b ) bCounter++;
```

```
39          }
40  }
```

GE3: Frequenzbegrenzer Wie vorab beschrieben ist eine extrem hohe Frequenz weder notwendig noch wünschenswert. Listing 7.11 zeigt die Spezifikation eines Frequenzbegrenzers. Der Frequenzbegrenzer benötigt zwei Variablen. In der Variable *triggered* wird abgelegt, ob ein anderer Prozess bereits eine Trigger-Nachricht gesendet hat. In der Variable *selftriggered* wird abgelegt, ob der Frequenzbegrenzer eine Nachricht von sich selbst erhalten hat.

Wenn beide Variablen *wahr* sind, schickt der Frequenzbegrenzer eine Nachricht an das Ziel und schickt sich darüber hinaus selbst eine Nachricht, wobei diese jedoch durch das Schlüsselwort `after` zeitlich verzögert wird.

Listing 7.11: Rebeca-Spezifikation für **GE3** (Frequenzbegrenzer)

```
1  reactiveclass FrequencyLimiter( 10 ) {
2          knownrebecs {
3                  Peer a;
4          }
5          statevars {
6                  int rebectype;
7                  boolean triggered;
8                  boolean selftriggered;
9          }
10         FrequencyLimiter() {
11                 rebectype = 4;
12                 triggered = false;
13                 selftriggered = true;
14         }
15         msgsrv Simulate() {
16                 if( sender == self ) selftriggered = true;
17                 else triggered = true;
18                 if( triggered && selftriggered ) {
19                         triggered = false;
20                         selftriggered = false;
21                         a.Simulate();
22                         self.Simulate() after( 16666 );
23                 }
24         }
25 }
```

GE4: Frequenz-Trigger Auf einen ähnlichen Mechanismus setzt der in Listing 7.12 gezeigte Frequenz-Trigger auf. Der Unterschied ist, dass dieser lediglich einmal eine Nachricht von außen benötigt und im Anschluss sich selbst eine Nachricht schickt, diese jedoch verzögert. Jedes Mal wenn dieser Prozess nun eine Nachricht von sich selbst erhält, schickt er sich wieder selbst eine und darüber hinaus eine an das Ziel.

Listing 7.12: Rebeca-Spezifikation für **GE4** (Frequenz-Trigger)

```
reactiveclass FrequencyTrigger( 10 ) {
        knownrebecs {
                Peer a;
        }
        statevars {
                int rebectype;
                boolean started;
        }
        FrequencyTrigger() {
                rebectype = 3;
                started = false;
        }
        msgsrv Simulate() {
                if( !started ) {
                        started = true;
                        self.Simulate() after( 20000 );
                } else if( sender == self ) {
                        a.Signal();
                        self.Simulate() after( 20000 );
                }
        }
}
```

GE5: Barrier Eine Barrier, wie in Listing 7.13 beschrieben, beschränkt das Triggern nicht nach zeitlichen, sondern nach logischen Kriterien. Dafür pflegt die Barrier Zähler für eingehende Nachrichten unterschiedlichen Typs und unterschiedlicher Quelle. Haben die Zähler den geforderten Stand erreicht, verschickt die Barrier eine Nachricht und setzt die Zähler zurück.

Listing 7.13: Rebeca-Spezifikation für **GE5** (Barrier)

```
reactiveclass Barrier( 10 ) {
        knownrebecs {
                Peer a;
        }
        statevars {
                int rebectype;
                int counterSimulate;
        }
        Barrier() {
                rebectype = 2;
                counterSimulate = 0;
        }
        msgsrv Simulate() {
                counterSimulate++;
                if( counterSimulate >= 3 ) {
                        a.Simulate();
```

```
17                              counterSimulate = 0;
18                      }
19              }
20      }
```

RMC Aus einer generierten Spezifikation erzeugt RMC den C++-Quelltext für einen Model Checker. Dieser wird mit GCC übersetzt und ausgeführt. Hierbei wird bereits überprüft, ob ein Deadlock vorkommen oder die eingehende Warteschlange eines Rebecs volllaufen kann. Dieser Model Checker erzeugt eine XML mit dem State Space.

State Space XML Nach dem der Model Checker fertig ist, wird diese State Space geparst. Dieser enthält die gefundenen Zustände sowie die Übergänge zwischen diesen Zuständen. In jedem Zustand wird darüber hinaus die Uhr jedes Prozesses abgelegt. Diese Datei kann mehrere GB groß werden.

Algorithmen Aus diesem State Space werden wie in Kapitel 6 beschrieben die einzelnen Szenarien erzeugt. Anschließend werden mit Hilfe der in Kapitel 5 beschriebenen Algorithmen für jedes Szenario die Metriken **M1**–**M3** berechnet.

In der Implementierung weichen der Algorithmus zur Ermittlung der Szenarien von dem in der Spezifikation dahingehend ab, dass der Algorithmus in Kapitel 6 bei der Erstellung eine Breitensuche nach Szenarien durchführt. Dies stellte sich in der Praxis jedoch als untauglich heraus, da die State Spaces so groß werden können, dass der Arbeitsspeicher der verwendeten Maschine nicht ausreichte. In der Implementierung wurde stattdessen eine Tiefensuche verwendet, die darüber hinaus viel sprachspezifische Optimierung benötigte, um nicht die gegebenen Ressourcen des verwendeten Computers zu sprengen.

Verbesserung der Genauigkeit der Vorhersagen

Es wurden zwei Maßnahmen vorgenommen, um die Präzision der Vorhersagen zu verbessern. Diese werden im folgenden beschrieben.

Bootstrapping der Anwendung Je nach Laufzeit einer Applikation macht das Bootstrapping, also die Zeitspanne in der die Sub-Systeme der Anwendung initialisiert werden und Assets von der Festplatte geladen werden, eine erhebliche Zeitspanne aus. Dies beeinflusst insbesondere die Parallelität. In Abbildung 7.17 ist deutlich zu erkennen, wie im Zeitraum zwischen Sekunde 10–17 der Laufzeit der Applikation der Grad der Parallelität deutlich erhöht ist. In diesem erkennbaren Zeitraum hat die Anwendung Assets von der Festplatte geladen.

Simulator X verwendet drei Prozesse zum Laden von Assets. Diese drei Prozesse wurden neben den Sub-Systemen in die Schemata eingefügt um die erhöhe Parallelität während des Bootstrappings abzubilden.

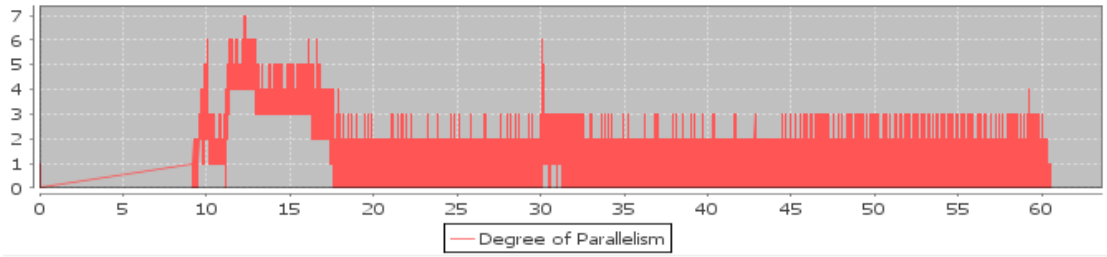


Abbildung 7.17: Veränderung des Grades der Parallelität einer Simulator X-Anwendung die 60 Sekunden lang lief. Deutlich ist der erhöhte Grad der Parallelität im Zeitraum 10–17 Sekunden zu sehen, wo drei Prozesse Assets von der Festplatte laden.

Scheduling der Aktor-Implementierung Die im Rahmen dieser Arbeit gewonnen Testdaten wurden auf einem Windows-System erstellt. Die von Simulator X verwendete Aktor-Implementierung hat auf Windows-Systemen die Eigenschaft, das Scheduling in 10 ms Intervallen durchzuführen. Gerade dieses Scheduling-Intervall hat einen erheblichen Einfluss auf die Latenz. Hierdurch wird es notwendig, zumindest diese Eigenschaft abzubilden.

Die Abbildung dieses Verhaltens ist zumindest in Bezug auf die Analyse der Latenz zwischen einem Tracker und einem Renderer einfach. Sei angenommen, dass der Tracker neue Daten zur Kopfposition mit der Nachricht `NewHeadPosition` an den Renderer überträgt. Sei darüber hinaus angenommen, dass die kleinste Zeiteinheit im Model Checking µs sind. Wie in Listing 7.14 zu sehen ist, muss lediglich in den Message Server für die Nachricht eine entsprechende Verzögerung zum nächsten vollen 10 ms Intervall eingebaut werden.

Listing 7.14: Abbildung des 10 ms Scheduling-Intervalls von Simulator X Windows für die Modellierung der Latenz zwischen Tracker und Renderer

```
1           ...
2           msgsrv NewHeadPosition() {
3                   delay( 10000 - (now\%10000) );
4           }
5   ...
6   }
```

Visualisierung

Die Abbildung 7.16 auf Seite 111 zeigte bereits den grafischen Editor für die Synchronisationsschemata. Wie darüber hinaus in Abbildung 7.15 auf Seite 110 dargestellt ist, wird im Lauf der Verarbeitung eine XML Datei mit dem State Space erzeugt. Hier setzt die Visualisierung an.

Zunächst wird wie in Abbildung 7.18 gezeigt diese XML Datei geladen. Hierbei wird ausgegeben, wie viele States und wie viele Transitions in der Datei gefunden wurden.

Darüber hinaus wird angegeben, welche Zeitspanne in diesem State Space repräsentiert wird und ob dieser eine Schleife bildet. Durch den Knopf „*Analyze*“ wird mit der Analyse begonnen und die Metriken **M1**–**M3** werden berechnet.

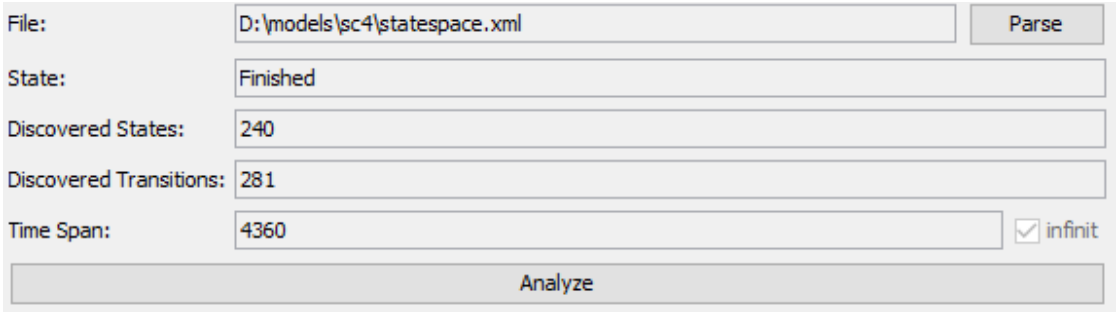


Abbildung 7.18: Laden eines erstellten State Spaces. Oben wird die XML-Datei mit dem State Space angegeben. Während des Ladens wird die Anzahl der gefundenen Zustände, Übergänge sowie die durch den State Space dargestellte Zeitspanne angezeigt.

Für die Berechnung der Metriken müssen jedoch die Szenarien ermittelt werden. Die geschieht parallel zur Berechnung der Metriken. Die Abbildung 7.19 zeigt, dass während der Berechnung die Anzahl der ermittelten und bereits analysierten Szenarien ausgegeben wird.

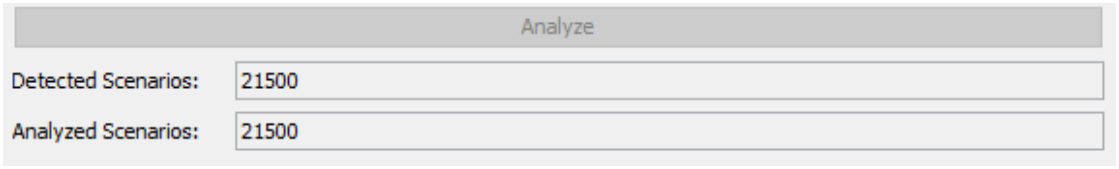


Abbildung 7.19: Während der Analyse des State Spaces wird ausgegeben, wie viele Szenarien gefunden wurden und wie viele bereits analysiert wurden.

M1: Parallelität Für die Metrik **M1** (Parallelität) wird wie im Profiler ausgegeben, wie hoch diese ohne Idle-Zeit ist. Die Abbildung 7.20 zeigt hier die numerische Darstellung der Parallelität. Diese wird einmal als minimale und maximale durchschnittliche Parallelität ausgegeben. Hierbei wird immer die durchschnittliche Parallelität über ein komplettes Szenario berechnet.

M2: Latenz Für die Ermittlung der Latenz muss wie vorab beschrieben zunächst konfiguriert werden, zwischen welchen Prozessen die Latenz ermittelt werden soll. Im Model Checker ist dies wesentlich einfacher gehalten als im Profiler, da das Modell in der Regel viele Implementierungsdetails außen vor lässt. Die Abbildung 7.21 zeigt die Konfiguration. Hier wird zunächst ausgewählt, welcher Prozess eine Information erzeugt. Im nächsten Schritt wird ausgewählt, durch welche Nachricht der Prozess diese Nachricht

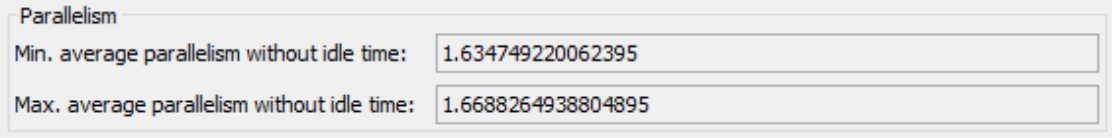


Abbildung 7.20: Darstellung der Metrik **M1** (Nebenläufigkeit) in numerischer Form. Hierbei wird einmal die durchschnittliche gemessene Parallelität gemäß Gleichung 4.10 von Seite 55 sowie die normierte durchschnittliche Parallelität aus Gleichung 4.13 von Seite 56 gezeigt.

kommuniziert. Im Anschluss wird ausgewählt, welcher Prozess diese Information verarbeitet. Abschließend wird eingestellt, welche Nachricht dazu führt, dass der Benutzer die Information sehen kann.

Im Beispiel von Abbildung 7.21 ist dies ein Tracker, der durch die Nachricht *NewHeadPosition* diese an den Renderer kommuniziert und diese Information für den Benutzer sichtbar ist, wenn der Renderer die Nachricht *PerformSimulationStep* ausgeführt hat.

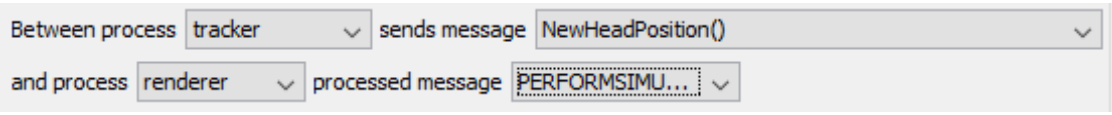


Abbildung 7.21: Konfiguration des Pfades, den eine Information im Modell nimmt.

Das Ergebnis wird numerisch dargestellt, wie in Abbildung 7.22 zu sehen ist. Es wird die minimale, maximale und durchschnittliche sowie der Median der Latenz dargestellt, wie diese über alle Szenarien ermittelt wurde.

Abbildung 7.22: Darstellung der ermittelten Latenz aus allen Szenarien, entsprechend des konfigurierten Pfades der Information. Es wird sowohl die minimale, maximale, durchschnittliche sowie das Median der Latenz angezeigt.

M3: Konsistenz Für die Konsistenz wird ermittelt welche Sub-Systeme untereinander kommunizieren und ob sich diese, zum Zeitpunkt eines Simulationsschrittes, in einem konsistenten Zustand befanden. Abbildung 7.23 zeigt hier, wie die Ergebnisse der Analyse dargestellt werden. In dieser Tabelle wird für jedes Sub-System, dargestellt mit welchem anderen Sub-System kommuniziert wurde und wie häufig auf einem Konsistenten Weltzustand gearbeitet wurde.

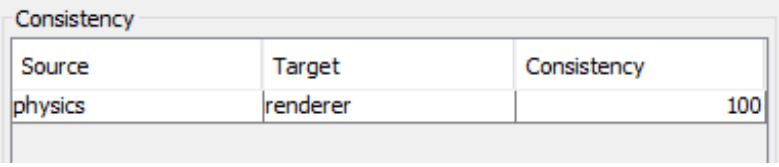


Abbildung 7.23: Darstellung der Metrik **M3** (Konsistenz) als Tabelle. Die erste Spalte gibt ein Sub-System an, welches eine Information erzeugt, die zweite Spalte ein Sub-System, das diese Information verarbeitet. In der dritten Spalte wird angegeben, wie viel Prozent der durchgeführten Simulationsschritte des zweiten Sub-Systems auf Grundlage von konsistenten Informationen der ersten Komponente erfolgten.

8 Erprobung des Konzepts

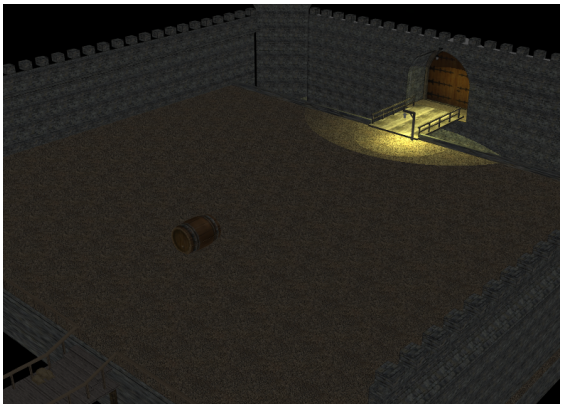
In diesem Kapitel wird beschrieben, wie die DSL dazu verwendet wurde die fünf identifizierten Nebenläufigkeits- und Synchronisationsschemata auf eine bestehende Applikation anzuwenden, um anschließend diese einmal mit dem Profiler und einmal mit dem Model Checker auf ihre Nebenläufigkeit, Latenz und Konsistenz zu untersuchen. Diese Präzision der Vorhersagen des Model Checkers werden mit den Ergebnissen aus einer älteren Version verglichen. Darüber hinaus wird beschrieben, wie der Profiler zur Analyse einer durchs Scheduling verursachten Latenz verwendet wurde. Abschließend werden noch einige Erkenntnisse in Bezug auf die temporale Granularität eines Modells und State Space Explosions aufgeführt.

8.1 Barrelstack Benchmark

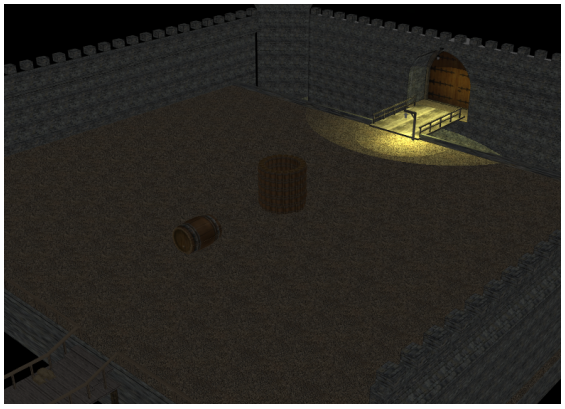
Der Barrelstack Benchmark ist eine kleine Anwendung, die auf der Demo-Anwendung *SiXton's Curse* (Fischbach et al., 2011) basiert und mit der gezielt Kommunikationslast in Simulator X erzeugt werden kann. Sie besteht, wie in Abbildung 8.1a zu sehen ist, aus einem großen Fass, welches für Kollisionen verwendet werden kann. Vor diesem Fass kann ein Stapel aus beliebig vielen Fässer erzeugt werden, wie diese z. B. in Abbildung 8.1b und 8.1e gezeigt werden. Die Erzeugung der Fässer erfolgt durch Tastendruck oder Zeitablauf. Ebenfalls kann eine Kraft auf das große Fass ausgeübt werden, welches hierdurch mit dem Stapel aus Fässern kollidiert, wie es in Abbildung 8.1c und 8.1d gezeigt wird. Wie der Vergleich von den Abbildungen 8.1a, 8.1d und 8.1f zeigt, lassen sich verschiedene Einstellungen in Bezug auf das Rendering vornehmen. So lässt sich die Umgebung und der Schattenwurf abschalten, um den Aufwand des Renderings zu verringern.

An die Anwendung lässt sich ein Tracker via VRPN anbinden, der die Kopfposition des Benutzers vorgibt. Für Testzwecke wurde ein Dummy-Server implementiert, der einen sich im Kreis bewegenden Körper simuliert. So können Benchmarks auch ohne eine konkrete Tracking-Hardware erfolgen.

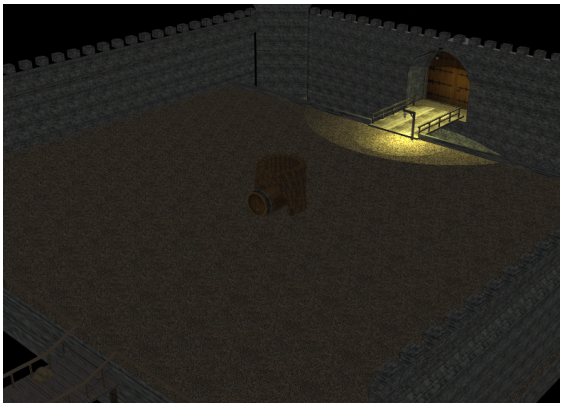
Der Barrelstack Benchmark ist für die Erprobung des Konzeptes gut geeignet, da er ein deterministisches und skalierbares Szenario repräsentiert, das in beliebiger Größenordnung Kommunikationslast in Simulator X erzeugen kann. Die Kommunikationslast wird durch eine Erhöhung oder Verringerung der Anzahl der Fässer gesteuert. Da es auf der Demo-Anwendung *SiXton's Curse* (Fischbach et al., 2011) basiert, verwendet es die typischen Komponenten von Simulator X. Darüber hinaus wurde der Barrelstack Benchmark bereits erfolgreich für die Leistungsmessung von Simulator X' Cluster-Subsystem verwendet (Rehfeld et al., 2013).



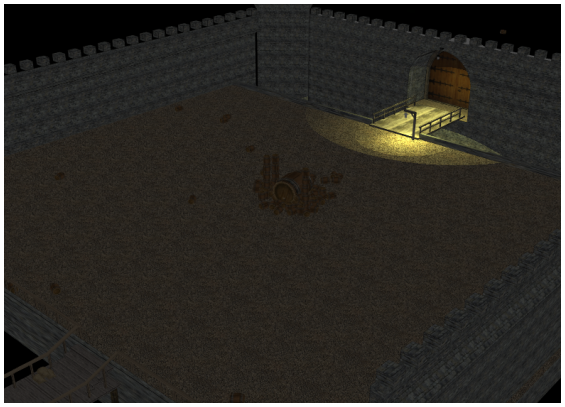
(a) Großes Fass für die Kollision in Umgebung.



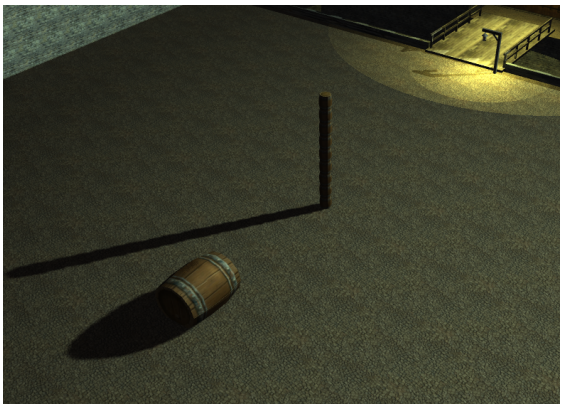
(b) Erstellter Stapel von kleinen Fässern.



(c) Großes Fass kollidiert mit Stapel von Fässern.



(d) Weiter fortgeschrittene Kollision.



(e) Stapel mit neun Fässern um eine niedrigere Last auf dem System zu erzeugen.

(f) Stapel und großes Fass für die Kollision, wobei die Umgebung deaktiviert wurde.

Abbildung 8.1: Diverse Screenshots aus dem Barrelstack Benchmark von Simulator X.

8.2 Versuchsaufbau

Für die Messungen mit dem Profiler wurde der Barrelstack Benchmark auf den in Tabelle 8.1 beschriebenen Systemen durchgeführt.

Tabelle 8.1: Verwendeter Computer für alle in diesem Kapitel beschriebenen Messungen

Komponente	Beschreibung
Typ	Thinkpad W520
CPU	Intel Core i7-2670QM 2.20 GHz
RAM	8 GB
GPU	NVIDIA Quadrio 1000M
Betriebssystem	Windows 8.1 und Windows 10

Der Barrelstack Benchmark wurde so konfiguriert, dass das große Fass beim Start der Anwendung erstellt wird. Nach zehn Sekunden Laufzeit, wird ein Stapel von Fässern erzeugt. Weitere 35 Sekunden später wird das große Fass in den Stapel gestoßen. Nach 60 Sekunden Laufzeit beendet sich die Anwendung selbst.

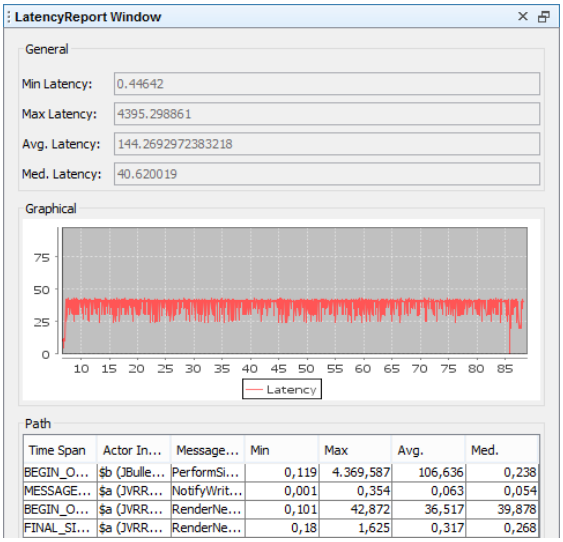
8.3 Ergebnisse

8.3.1 Durch das Scheduling erzeugte Latenz

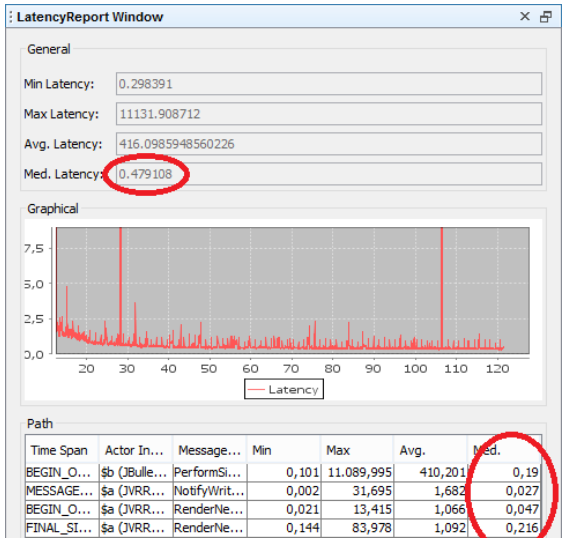
Durch eine mit dem Profiler durchgeführte Analyse wurde eine durch das Scheduling der verwendeten Aktor-Implementierung erzeugte Latenz aufgedeckt. Bei der Untersuchung der Kommunikation zwischen Physik-Simulation und Renderer fiel auf, dass die Latenz zwischen Physik-Simulation und Renderer im Median 40 ms beträgt (vgl. Abbildung 8.2a). Darüber hinaus ist an der unteren Tabelle in Abbildung 8.2a zu sehen, dass die Latenz in der Zeit zwischen dem Beenden der Verarbeitung einer neuen Transformation eines Objektes und dem Beginn des nächsten Renderings hervorgerufen wird.

Abbildung 8.3 stellt hier den Fluss der Information zwischen Physiksimulation und Renderer anhand der in Tabelle 5.2 auf Seite 66 definierten Zeitabschnitte dar. Der erste Abschnitt ist die Zeitspanne zwischen Beginn des Simulationsschritts der Physiksimulation bis eine Nachricht an den Renderer versendet wird. Die zweite Zeitspanne ist die Zeit zwischen Versand dieser Nachricht und dessen Verarbeitung. Der dritte Abschnitt ist die Zeit zwischen Verarbeitung der Nachricht und Beginn des Simulationsschritts des Renderers. Der letzte Abschnitt ist die Zeitspanne zwischen Beginn und Ende des Simulationsschritts.

Die entdeckte Latenz entsteht, wie in der Tabelle in Abbildung 8.2a zu sehen, vor allem im Abschnitt drei des Kommunikationsweges. Im Median vergingen 39 ms zwischen der Verarbeitung des übertragenen neuen Weltzustandes und dem Beginn des nächsten Rendering-Durchlaufs. In dieser Zeitspanne führt der Renderer keine Berechnungen aus,



(a) Latenzbericht bei der Verwendung von **Schema 3**, wobei eine hohe Latenz zwischen Physiksimulation und Renderer gemessen wird.



(b) Latenzbericht bei der Verwendung von **Schema 1**, wobei eine niedrigere Latenz gemessen wurde.

Abbildung 8.2: Latenzberichte zur Analyse der Barrelstack Benchmarks unter Verwendung von **Schema 1** und **Schema 3**. Um mehr Daten zu der gefundenen Latenz zu sammeln, wurde abweichend vom in Abschnitt 8.2 beschriebenen Versuchsaufbau die Anwendung länger laufen gelassen.

es handelt sich um eine reine Wartezeit, bis der Scheduler den nächsten Rendering-Durchlauf anstößt.

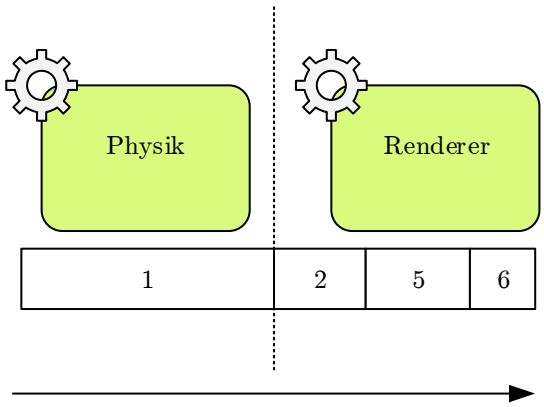
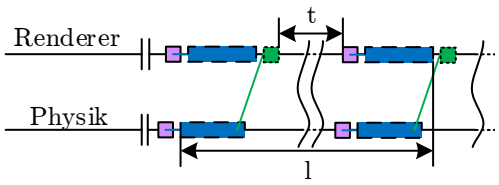
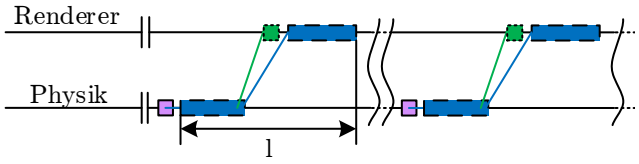


Abbildung 8.3: Zeitabschnitte des Informationsflusses zwischen der Physiksimulation und dem Renderer. Hierbei werden die in Tabelle 5.2 von Seite 66 Zeitabschnitte verwendet.



(a) Informationsfluss mit hoher Latenz l. Die neuen Informationen von der Physiksimulation treffen nach den gleichzeitig ausgeführtem Rendering ein. Hierauf folgt jedoch eine Wartezeit, bis der Scheduler den nächsten Rendering-Durchlauf startet. Die hierdurch erzeugte zusätzliche Latenz ist mit t dargestellt.



(b) Informationsfluss mit niedriger Latenz. Die neuen Informationen werden an den Renderer übertragen und anschließend stößt dieser den neuen Rendering-Durchlauf an. Die Latenz l ist in diesem Fall kürzer, da die zusätzliche Latenz wegfällt.

Abbildung 8.4: Zeitliche Darstellung des Informationsflusses und der Nachrichtenverarbeitung zwischen Physiksimulation und Renderer. Lila Balken mit durchgängigem Rahmen stellen eine durch den Scheduler ausgeführte Aktion dar. Grüne Balken mit gepunkteter Umrandung stellen die Verarbeitung von Update-Nachrichten dar. Blaue Balken mit gestrichelter Linie stellen einen durchgeführten Simulationsschritt dar.

In Abbildung 8.4 wird der entdeckte Sachverhalt noch einmal deutlicher als Timeline dargestellt. Abbildung 8.4a visualisiert hier den Grund für die gemessene Latenz aus Abbildung 8.2a. Die Physiksimulation und der Renderer starten gleichzeitig mit ihrem Simulationsschritt. Während dieses Simulationsschritts schickt die Physiksimulation neue Daten an den Renderer, welche dieser erst nach Fertigstellung des eigenen Simulationsschritts verarbeitet. Nun folgt eine längere Wartezeit, bis der Scheduler den nächsten Rendering-Durchlauf anstößt. Genau dies führt zu der hohen Latenz.

Das dargestellte Verhalten entspricht dem **Schema 4** aus Kapitel 3.1.4. Allerdings war zum Zeitpunkt der Messung das **Schema 3** eingestellt. Eine Analyse zeigte, dass der Scheduler der verwendeten Aktor-Implementierung keine kleineren Zeitscheiben als 10 ms zulässt und darüber hinaus Aktoren möglichst parallel ausführt.

Für das in Abbildung 8.4b dargestellte Verhalten wurde das **Schema 1** verwendet. Da der Renderer hier von der Physiksimulation getriggert wird, hat dieser vorher in jedem Fall die durch die Physiksimulation versandten Nachrichten verarbeitet. Darüber hinaus führt der Scheduler den Rendering-Durchlauf sofort aus und taktet das Rendering nicht in seine nächste 10 mm Zeitscheibe. Im Ergebnis führt dies, wie in Abbildung 8.2b dargestellt, zu einer erheblich geringeren Latenz.

Diskussion

Die wichtigste Erkenntnis, die sich aus diesem Ergebnis erschließt, ist der Umstand, dass in einem asynchronen Interaktiven Echtzeitsystem Latenzen entstehen können, die nicht innerhalb der Middleware oder der Anwendung entstehen, sondern ein Effekt des darunterliegenden Schedulings sind. Der entwickelte Profiler und die entwickelte Darstellung der Latenz und des Pfades der Informationsverarbeitung halfen bei der schnellen Aufdeckung der Ursache der Latenz.

Das entwickelte Synchronisationskonzept und die Implementierung in Form einer DSL erlaubte die schnelle Umkonfiguration des Systems. Hierdurch konnte leicht überprüft werden, ob die Latenz auch in anderen Schemata auftritt.

Für asynchrone Interaktive Echtzeitsysteme im Allgemeinen ergibt sich hieraus, dass diese eine starke Abhängigkeit zum Verhalten des darunterliegenden Schedulers für Threads und Tasks haben.

Stauffert et al. (2016b) führten eine Untersuchung verschiedener Techniken für die Interprozesskommunikation durch (Stauffert et al., 2016a), wobei auch das von Simulator X verwendete Akka Framework untersucht wurde. Im Ergebnis stellten Stauffauert et al. gelegentliche Spitzen in der Latenz fest, wobei vermutet wird, dass der Garbage Collector der Java Virtual Maschine hierfür verantwortlich ist. Die Untersuchung durch Stauffert et al. weist jedoch an mindestens einer Stelle einen methodischen Fehler auf. Beim Benchmarking von Locking-Mechanismen in Java führt der Garbage Collector regelmäßig Arbeiten durch, obwohl dies im geschilderten Szenario nicht passieren dürfte. Eine Analyse des Quelltextes des Benchmarks zeigte, dass die Datenstruktur, in der die Ergebnisse gespeichert wurden, nicht vorher alloziert wurde. Hierdurch muss in regelmäßigen Abständen zuerst neuer Speicher alloziert und anschließend alle Daten in den neuen Speicherbereich kopiert werden. Der alte Speicherbereich wird anschließend vom Garbage Collector Bereinigt, was die Messung verfälscht.

In Bezug auf die Latenzspitzen und ihrer Regelmäßigkeit wirken die Ergebnisse dieser Arbeit und die von Stauffert et al. zunächst widersprüchlich. Wie in Abbildung 8.4a schwankt auch in dieser Arbeit die Latenz, jedoch ergeben sich keine Spitzen, wie diese durch Stauffert et al. gemessen wurden. Wie in Abbildung 8.4b zu sehen ist, wurden zwar hierbei Latenzspitzen gemessen, jedoch ist hierbei die Skalierung der Achse zu beachten. Der dort abgebildete Bereich bildet Latenzen bis 7,5 ms Sekunden ab. Es traten also keine solchen extremen Latenzspitzen auf.

Dies bestätigt die bereits in Bezug auf die Parallelität auf Seite 22 formulierte Kritik an künstlichen Benchmarks. Solche „Micro“-Benchmarks testen einen spezifischen Sachverhalt mit hoher Frequenz. Moderne Systeme verfügen jedoch auf mehreren Ebenen über Techniken zur Selbstoptimierung. Einer dieser Optimierungsmechanismen wurde durch Stauffert et al. auch adressiert, in dem Cache-Misses provoziert wurden. Jedoch verhalten sich die getesteten Mechanismen durch weitere Optimierungen anders, als sie es im Kontext einer Gesamtanwendung tun. In Bezug auf die Parallelität waren Best et al. (2009) die Einzigen, die eine wesentliche Skalierung erreicht haben. Und diese Ergebnisse wurden auf Grundlage einer vollwertigen Anwendung erreicht.

8.3.2 Vergleich von Messungen durch den Profiler mit Vorhersagen durch den Model Checker

Mit Hilfe des Barrelstack Benchmarks wurden die vorhergesagten Werte für die Metriken **M1**–**M3** mit Messungen verglichen. Hierzu wurden die Synchronisations**schemata 1**–**5** mit Hilfe des grafischen Editors des Model Checkers erstellt. Diese Modelle benötigen für eine Prüfung Annahmen über den Zeitbedarf der Abarbeitung verschiedener Nachrichten, z. B. wie lang ein Simulationsschritt des Renderers oder der Physiksimulation braucht. Diese Daten wurden vorab mit dem Profiler ermittelt und sind in Anhang 1.4 auf Seite VII aufgeführt.

Tabelle 8.2 fasst die Ergebnisse zusammen und vergleicht die Vorhersage mit der Messung. Die Ergebnisse sind noch einmal grafisch in Abbildung 8.5 aufbereitet. Für Tabelle 8.2 gilt, dass die prozentuale Differenz d zwischen Vorhersage v und Messung m nach folgender Formel berechnet wurde:

$$d = \frac{v - m}{m} * 100 \tag{8.1}$$

Somit ist ein positiver prozentualer Wert eine Überschätzung in der Vorhersage, während ein negativer Wert eine Unterschätzung darstellt.

Schema 2 funktionierte zwar in der Testanwendung für die Testlaufzeit der Anwendung, der Model Checker entdeckte jedoch, dass der Renderer in diesem Schema saturieren kann. Deshalb wurden die Vorhersagen nicht für das **Schema 2** berechnet, da dieses Schema nicht benutzbar ist.

Für die Metrik **M1** (Nebenläufigkeit) trat die größte Unterschätzung bei **Schema 1** auf, wo die Vorhersage 1,12 war, jedoch 1,28 gemessen wurde, die Nebenläufigkeit also um -12,5 % unterschätzt wurde. Die größte Überschätzung ergab sich bei **Schema 3**, wo 1,52 vorhergesagt, jedoch lediglich 1,29 gemessen wurde. Es erfolgte somit eine Überschätzung der Nebenläufigkeit von 17,82 %. Für **Schema 4** und **Schema 5** war die Abweichung weniger als 10 %. Die Ergebnisse sind noch einmal in Abbildung 8.5a dargestellt.

Noch bessere Ergebnisse wurden bei der Vorhersage der Metrik **M2** (Latenz) erzielt. Hier lag die größte Differenz bei **Schema 3** vor, mit einer Überschätzung von 7,98 %. Als Latenz wurden 35,2 ms vorhergesagt, jedoch lediglich 32,6 ms gemessen. Die größte Unterschätzung trat bei **Schema 1** auf, wo 30,3 ms als Latenz vorhergesagt war, aber 32,7 ms gemessen wurden, was eine Unterschätzung von -7,24 % ergibt. Die Ergebnisse sind noch einmal in Abbildung 8.5b dargestellt.

Für die Konsistenz stimmten Vorhersage und Messung für **Schema 1** und **Schema 4** überein. In beiden Schemata ist die Konsistenz zwischen Renderer und Physik gewahrt. In **Schema 3** und **Schema 5** ist dies nicht der Fall. Die prozentualen Werte unterscheiden sich stark zwischen Messung und Vorhersage. Hier ist es fraglich, ob die Berechnung der prozentualen Differenz zwischen Vorhersage und Messung überhaupt Sinn ergiebt, denn die Messung zeigt nur einen kleinen Ausschnitt aus den möglichen Verhaltensweisen, während das Model Checking möglichst eine Untersuchung aller möglichen Ausführungspfade darstellt. Die erhaltenen Ergebnisse sind in Abbildung 8.5c dargestellt.

Tabelle 8.2: Vorhergesagtes und gemessenes Ergebnis für die Nebenläufigkeit, Latenz und Konsistenz im Barrelstack Benchmark für die vorgestellten Synchronisationsschemata. Die Differenz wird in Bezug zum gemessenen Wert berechnet. Ein positiver Wert bedeutet, dass das Model Checking einen Wert überschätzt hat, während einer negativer Wert bedeutet, dass dieser unterschätzt wurde.

	M1: Nebenläufigkeit			M2: Latenz			M3: Konsistenz		
	Vorhersage	Messung	Differenz	Vorhersage	Messung	Differenz	Vorhersage	Messung	Differenz
Schema 1	1,12	1,28	-12,5%	30,3ms	32,7ms	-7,34%	100%	100%	0%
Schema 2	–	1,87	–	–	30,4ms	–	–	–	–
Schema 3	1,52	1,29	17,83%	35,2ms	32,6ms	7,98%	77,79%	99,34%	-21,69%
Schema 4	1,24	1,37	-9,49%	35,9ms	33,6ms	6,85%	100%	100%	0%
Schema 5	1,63	1,52	7,24%	7,9ms	7,6ms	3,9%	5,37%	99,62%	-94,61%

In (Rehfeld et al., 2016) wurden mit Hilfe einer früheren Version der Model Checking-Anbindung Vorhersagen zu den Metriken **M1** und **M2** auf Grundlage der gleichen Profiling Daten erstellt. In Tabelle 8.3 und Abbildung 8.6 wird die Differenz der Vorhersagen der alten Version und der in dieser Arbeit verwendeten Version verglichen. Die Verbesserung i zwischen altem Wert a und neuem Wert n wird nach folgender Formel berechnet:

$$i = \frac{|a| - |n|}{|a|} * 100 \tag{8.2}$$

Lediglich die Vorhersage der Metrik **M1** (Nebenläufigkeit) für das **Schema 3** hat sich um -91,72 % verschlechtert. Dies wirkt zunächst wie eine große Verschlechterung. Die Differenz zwischen Vorhersage und Messung beträgt jedoch 17,83 %. In der alten Version betrug die größte absolute Differenz -28,9 % für die Metrik **M1** bei **Schema 5**. Trotz Verschlechterung hat diese Vorhersage immer noch eine niedrigere absolute Differenz als die schlechteste Vorhersage mit der alten Version.

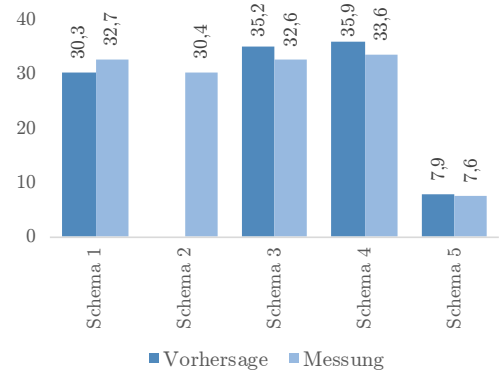
Alle anderen Vorhersagen haben sich zwischen 33,51 %–74,95 % verbessert. Insbesondere für die Latenz wurde eine deutliche Verbesserung erzielt, da die größte Differenz 7,98 % betrug.

Diskussion

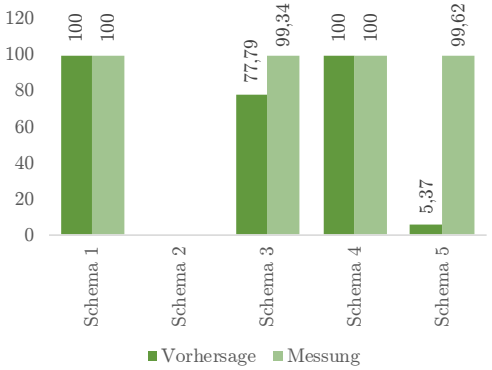
Die Vorhersage der Latenz kann mit einer absoluten Abweichung von 3,9 %–7,98 % als sehr präzise bezeichnet werden. Auch bei der Vorhersage der Parallelität waren die Ergebnisse mit einer absoluten Abweichung im Bereich 7,24 %–17,83 % brauchbar. Präzise Ergebnisse wurden ebenfalls bei der vorhersage der Konsistenz erreicht, wo alle Schemata, die zu einem inkonsistenten Weltzustand führen können, erkannt wurden.

Im Vergleich zu den ersten Ergebnisse aus (Rehfeld et al., 2016) wurde bei der Vorhersage der Latenz eine Verbesserung von bis zu 69,43 % erzielt. Bei der Vorhersage

(a) Vergleich zwischen vorhergesagter und gemessener Nebenläufigkeit (M1) in der Barrelstack-Anwendung.



(b) Vergleich zwischen vorhergesagter und gemessener Latenz (M2) in der Barrelstack-Anwendung.



(c) Vergleich zwischen vorhergesagter und gemessener Konsistenz (M3) in der Barrelstack-Anwendung.

Abbildung 8.5: Grafische Darstellung der Ergebnisse aus Tabelle 8.2.

der Latenz haben sich keine Ergebnisse verschlechtert. In Bezug auf die Vorhersage der Paralellität haben sich alle bis auf eine Vorhersage im Bereich von 33,51 % –74,95 % verbessert. Nur in einem Fall fand eine Verschlechterung um -91,72 %. Die Abweichung ist dennoch geringer als die maximale Abweichung die in (Rehfeld et al., 2016) erreicht wurde.

Alle Verbesserungen wurden dadurch erzielt, dass das Modell um das Laden von Assets erweitert wurde und Änderungen am Model-Checker durchgeführt wurden. In Anbetracht der Tatsache, dass die Vorhersage des Verhaltens eines Interaktiven Echtzeitsystems vorher noch nicht durchgeführt wurde, ist anzunehmen, dass durch weitere Änderungen auch die Vorhersage der Parallelität mit einer maximalen absoluten Abweichung von weniger als 10 % erreicht werden kann.

Auch die Vorhersage der Konsistenz war sehr erfolgreich und erfolgte auch das erste

Tabelle 8.3: Vergleich der Vorhersagen aus (Rehfeld et al., 2016) und dem verbesserten Modell aus dieser Arbeit.

	M1: Nebenläufigkeit			M2: Latenz		
	Alt	Neu	Verbesserung	Alt	Neu	Verbesserung
Schema 1	-18,8 %	-12,5 %	33,51 %	-14,1 %	-7,34 %	47,94 %
Schema 2	–	—	–	–	—	–
Schema 3	9,3 %	17,83 %	-91,72 %	-26,1 %	7,98 %	69,43 %
Schema 4	19,7 %	-9,49 %	51,83 %	-26,8 %	6,85 %	74,44 %
Schema 5	-28,9 %	7,24 %	74,95 %	3,9 %	3,9 %	0 %

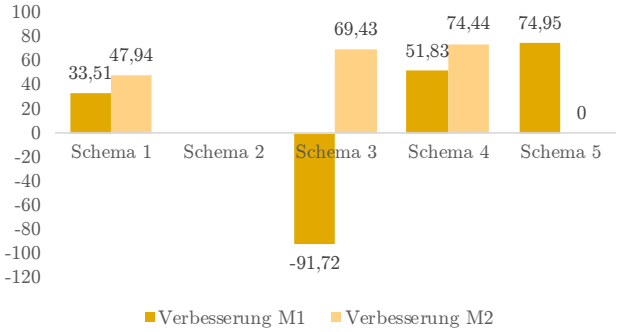


Abbildung 8.6: Verbesserung der Vorhersage und Messung im Vergleich zu (Rehfeld et al., 2016) für die Barrelstack-Anwendung.

Mal. Es zeigt sich jedoch, dass hier eine numerische Aussage wie bei den anderen Metriken wenig Sinn ergiebt. Für den Middleware- oder Anwendungsentwickler ist hier auch eher die Tatsache wichtig, ob ein Schema die Konsistenz zusichert und weniger wie häufig es dann zu inkonsistenten Zuständen kommt.

8.3.3 Temporale Granularität, State Space Explosion und „Best-Practice“

Während der Erprobung wurde festgestellt, dass eine feine temporale Granularität zu einem sehr großen State Space führt. Im Laufe der Arbeit stellte sich heraus, dass der in Abbildung 8.7 dargestellte Workflow ein effizientes Nutzen des Model Checkers erlaubt. Hierbei werden zunächst die oben genannten zeitlichen Verarbeitungszeiten geschätzt oder mit Hilfe des Profilers gemessen. Anschließend werden diese Zeiten in das entsprechende Schema eingefügt. Im ersten Schritt erfolgt ein Model Checking mit der temporalen Granularität von 10^{-3} (Millisekunden). Anschließend wird die temporale Granularität

auf 10^{-4} erhöht und verglichen, ob sich die Ergebnisse für die Metriken **M1**–**M3** verändert haben. Ist dies der Fall wird die temporale Granularität so weit erhöht, bis zwischen zwei Erhöhungen keine Veränderung mehr in der Ergebnissen stattfindet.

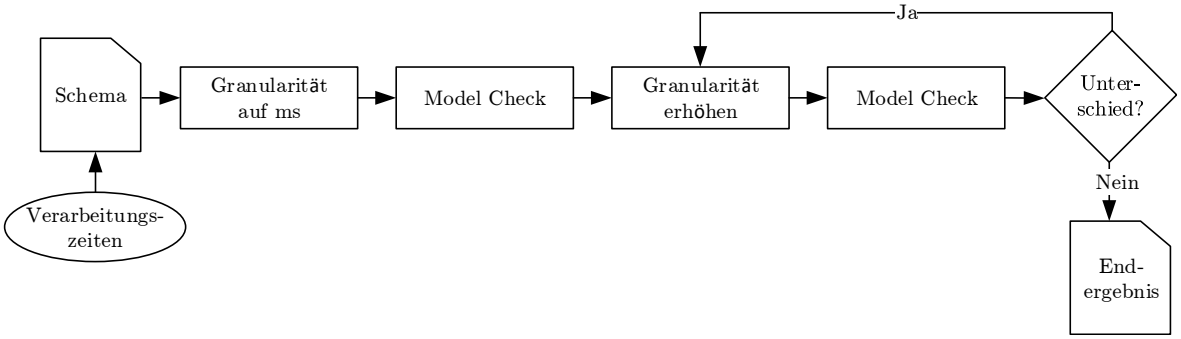
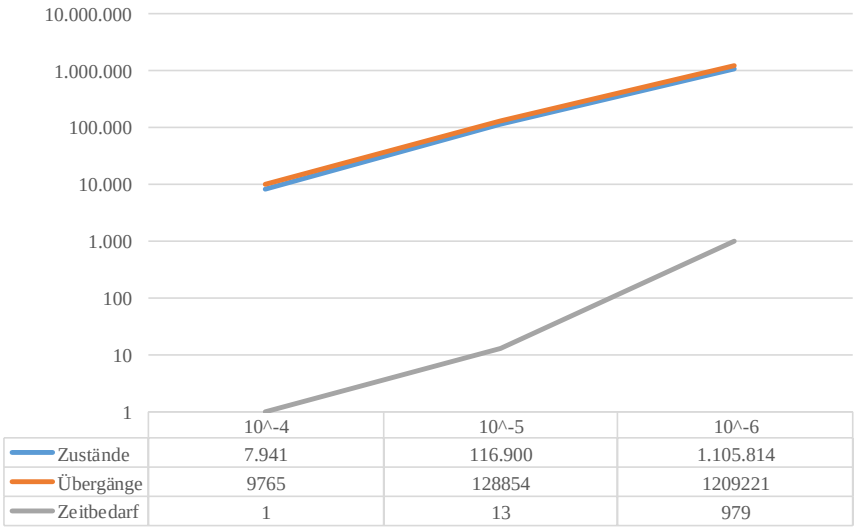


Abbildung 8.7: Arbeitsfluss zur Verwendung der Toolchain aus Abbildung 7.15.

In Abbildung 8.8 wird gezeigt, wie bei einer Erhöhung der temporalen Granularität, die Anzahl der Zustände und Übergänge sowie der zeitliche Bedarf ansteigt. Während bei einer temporalen Granularität von 10^{-4} die Erzeugung des State Spaces in weniger als einer Sekunde abgeschlossen ist, dauert es bei 10^{-5} 13 Sekunden und bei 10^{-6} über 16 Minuten. Auch die Anzahl der Zustände und Übergänge steigt exponentiell an.



	10^-4	10^-5	10^-6
Zustände	7.941	116.900	1.105.814
Übergänge	9765	128854	1209221
Zeitbedarf	1	13	979

Abbildung 8.8: Zunahme der Anzahl der Zustände, Übergange und des Zeitbedarfs (in Sekunden) bei Erhöhung der temporalen Granularität.

Die Ergebnisse des vorherigen Vergleichs zwischen Profiling und Model Checking wurden mit einer temporalen Granularität von 10^{-5} erstellt.

9 Zusammenfassung und Ausblick

In diesem Kapitel werden die Antworten auf die in der Einleitung formulierten Forschungsfragen noch einmal zusammengefasst. Anschließend werden mögliche technische Verfeinerungen an der erstellten prototypischen Implementierung genannt. Im Anschluss werden Vorschläge für auf diese Arbeit aufbauende Forschung formuliert. Das Kapitel schließt ab mit einem Fazit des erreichten Beitrags.

9.1 Forschungsfrage 1

> *Welche Modelle existieren, auf deren Grundlage ein nebenläufiges Interaktives Echtzeitsystem implementiert werden kann, wobei das Modell die typischen Probleme der parallelen Programmierung vermeidet?*

Zur Beantwortung der ersten Forschungsfrage wurden in Kapitel 2.1.4 acht verschiedene Modelle für die Nebenläufigkeit in Interaktiven Echtzeitsystemen unter fünf vorher definierten Kriterien verglichen. Die Resultate wurden in Tabelle 2.1 auf Seite 26 zusammengefasst.

Hierbei stellte sich das *Asynchrone Modell* als beste Grundlage für diese Arbeit heraus. Wesentliche Argumente für dieses Modell waren, dass es keine Festlegung über die Granularität der zu parallelisierenden Einzelteile trifft und von daher der Implementierungsaufwand geringer ist als bei der Anwendung anderer Modelle, wie z. B. einem Task-Tree Modell. Es erlaubt jedoch eine feinere Granularität, was es von Modellen unterscheidet, die den Simulation Loop lediglich auf der Ebene der Sub-Systeme parallelisiert. Darüber hinaus stellt die Asynchronität einen weiteren Freiheitsgrad neben der Granularität dar.

Durch Asynchronität wird jedoch die Konsistenz zu einem bedeutenden Problem, da asynchrone Teile des Systems über eine konsistente Sicht auf die simulierte Szene verfügen müssen. Das Asynchrone Modell enthält hierfür keine eigene Lösung. Die Wahrung der Konsistenz ist deshalb Aufgabe einer darüberliegenden Schicht.

Um eine Lösung für dieses Problem zu finden, wurden darüber im Kapitel 2.2 vier verschiedene Synchronisationstechnologien anhand von sieben Kriterien verglichen. Die Resultate wurden in Tabelle 2.2 auf Seite 33 zusammengefasst.

Hierbei wurde das bei Kontrollsystemen verbreitete Modell der *Prozessvariablen* ausgewählt. Dies wurde primär ausgewählt, weil andere Modelle unbeherrschbar sind (klassisches Locking), keine zuverlässige Implementierung existiert (STM) oder gar keine öffentlich zugängliche Implementierung verfügbar ist (SvS).

Aufgrund des ausgewählten Modells und der ausgewählten Synchronisationstechnologie wurde Simulator X als Basis für die prototypische Implementierung ausgewählt, da

Simulator X sowohl auf dem asynchronen Modell basiert, als dass es auch Prozessvariablen einsetzt.

9.2 Forschungsfrage 2

> *Welche sinnvollen Nebenläufigskeits- und Synchronisationsschemata existieren für ein asynchrones Interaktives Echtzeitsystem und wie können diese verwendet werden, um eine Optimierung hin zu einem bestimmten Verhalten zu gewährleisten?*

Für ein asynchrones Interaktives Echtzeitsystem wurden in Kapitel 3 fünf verschiedene Nebenläufigkeits- und Synchronisationsschemata identifiziert.

- Im ersten Schema wurde der klassische Simulation Loop nachgebildet und die Ausführung der Sub-Systeme wurde sequentiell abgearbeitet. Die einzigen wirklich nebenläufigen Elemente in diesem Schema sind das Laden von Assets zu Beginn der Laufzeit und Geräte, wie z. B. ein Tracker, die mit einer eigenen Frequenz laufen und nicht von der Anwendung beeinflusst werden können.
- Im zweiten Schema liefen alle Bestandteile des Systems mit ihrer maximalen Frequenz. Durch die Technik des Model Checkings wurde aufgedeckt, dass dieses Schema zur Saturierung einzelner Prozesse führen kann.
- Im dritten Schema liefen alle Prozesse mit ihrer eigenen Frequenz. Sowohl durch das Profiling und das Model Checking wurde festgestellt, dass die Konsistenz zwischen den Prozessen nicht gewahrt ist.
- Im vierten Modell wurden alle Prozesse gleichzeitig gestartet. Dies führte zu einer Wahrung der Konsistenz.
- Im fünften und letzten Modell wurde das Rendering an neue Daten vom Tracker gebunden. Dies reduzierte die Latenz deutlich, führte jedoch erneut zu Problemen bei der Konsistenz.

Die erarbeiteten Schemata sind *keine abschließende Aufstellung*. Das asynchrone Modell lässt auch weitere Schemata zu, die noch erarbeitet werden könnten. Dies hätte jedoch den Rahmen dieser Arbeit gesprengt.

Auf Grundlage dieser fünf Schemata wurden fünf Grundelemente erarbeitet und in Kapitel 3.2 vorgestellt. Diese fünf Grundelemente sind:

GE1 Unbewegter Beweger

GE2 Sub-System

GE3 Frequenzbegrenzer

GE4 Frequenz-Trigger

GE5 Barrier

Eine Implementierung dieser fünf Grundelemente wurde in Kapitel 7.2.1 für Simulator X vorgestellt. Durch diese Implementierung konnte Simulator X so konfiguriert werden, dass es sich entsprechend der vorher identifizierten Schemata verhält.

9.3 Forschungsfrage 3

> *Wie können Profiling-Daten im Kontext eines asynchronen Interaktiven Echtzeitsystems für einen Entwickler aufbereitet werden, wobei man den spezifischen Eigenschaften eines Interaktiven Echtzeitsystems gerecht wird und insbesondere die Untersuchung der Nebenläufigkeit, Latenz und Konsistenz ermöglicht wird?*

Für die Beantwortung der dritten Forschungsfrage wurden zunächst in Kapitel 4 drei Metriken definiert, die die zu untersuchende Latenz, Parallelität und Konsistenz darstellen. Auf dieser Grundlage wurden im Kapitel 5 Algorithmen spezifiziert, die aus den fürs Profiling typischen Daten die vorher definierten Metriken berechnen. Diese Algorithmen wurden in einer Implementierung umgesetzt, die im Kapitel 7.2.2 beschrieben wird.

Während sich die Profiler von anderen Interaktiven Echtzeitsystemen, wie in Kapitel 2.3.2 im Kontext von Game-Engines dargestellt, an den Profilern für die allgemeine Software-Entwicklung orientieren, wurden hier neue Darstellungsformen erarbeitet.

Dies trifft insbesondere auf die Darstellung der Latenz zu. Bei der Latenz handelt es sich um keine übliche Metrik von Profilern. In dem in dieser Arbeit vorgestellten Profiler lässt sich der Pfad einer Information konfigurieren. Aus dieser Konfiguration wird aus gemessenen Daten die Latenz dieser Information bestimmt. Bei der Darstellung der Information erwies sich insbesondere die Darstellung der einzelnen Abschnitte der Information in Form einer Tabelle als hilfreich. Diese Darstellung wurde, wie in Kapitel 8.3.2 beschrieben, für die Identifikation der Ursache und Beseitigung einer hohen Latenz verwendet.

Darüber hinaus diente der Profiler zur weiteren Untersuchung und zu Erkenntnissen über Implementierungsdetails von Simulator X und der darunterliegenden Aktor-Implementierung. Diese wurden wiederum verwendet, um das Modell im Model Checking weiter zu verfeinern und die Präzision weiter zu erhöhen.

Auch die Einführung der Messung der Konsistenz durch einen Profiler stellt einen neuen Beitrag dar. Hierfür wird im Profiler ausgewählt, zwischen welchen Sub-Systemen die Konsistenz ermittelt werden soll. Im Anschluss wird die Konsistenz in einer prozentualen Angabe dargestellt.

9.4 Forschungsfrage 4

> *Kann Model Checking verwendet werden, um das Modell eines Interaktiven Echtzeitsystems auf die zu erwartende Nebenläufigkeit, Konsistenz und Latenz zur prüfen und liegen die vorhergesagten Werte nahe genug an den ge-*

messenen Werten des Profilings, damit die erzielten Resultate in der Praxis brauchbar sind?

Für die Beantwortung der vierten Forschungsfrage wurde zunächst in Kapitel 6 die vorher definierten Metriken aus einem State Space berechnet. Darüber hinaus wurden in Kapitel 7.2.3 neun formale Sprachen und ihre Model Checker anhand von sechs Kriterien miteinander verglichen. Die Ergebnisse wurden in Tabelle 7.1 auf Seite 109 zusammengefasst. Aus diesem Vergleich ging die Sprache *Rebeca* und ihr Model Checker *RMC* als beste Kandidaten hervor.

Die Implementierung wurde in Kapitel 7.2.3 beschrieben. Zwei Änderungen waren für diese Arbeit an RMC notwendig. Für die Spezifikation eines Interaktiven Echtzeitsystems wurde ein grafischer Editor erstellt, mit dem Spezifikationen eines asynchronen Interaktiven Echtzeitsystems erstellt werden können. Aus dieser grafischen Darstellung wird eine Rebeca-Spezifikation erstellt. Aus dieser Spezifikation erzeugt RMC den State Space. Aus diesem State Space werden wiederum die vorher formal spezifizierten Metriken berechnet.

In Kapitel 8.3.2 wurden die Daten aus dem Profiler und die Vorhersagen aus dem Model Checker im Kontext einer Anwendung verglichen. Hierbei war insbesondere die Vorhersage der Latenz sehr erfolgreich. Hierbei lag die maximalen Differenz bei 7,98 % bzw. 2,6 ms. Die niedrigste Differenz zwischen Vorhersage und Messung der Latenz lag bei 3,9 % bzw. 0,3 ms absolut. Auch bei der Vorhersage der Nebenläufigkeit waren die Ergebnisse sehr zufriedenstellend. Hierbei lag die größte Differenz bei 17,8 % und die geringste bei -9,5 %.

Bei der Untersuchung der Konsistenz wurden alle Schemata, die zu einem inkonsistenten Weltzustand bei einem Prozess führen können, identifiziert.

In (Rehfeld et al., 2016) wurden die Konsistenz und Latenz bereits mit einer vorherigen Version der Model-Checking-Anbindung überprüft. In allen bis auf einem Fall gelang eine erhebliche Verbesserung der Präzision der Vorhersage. Die Differenz wurde hierbei zwischen 33,51 % und 74,95 % verbessert. Nur in einem Fall verschlechterte sich die Präzision der Vorhersage. Trotz dieser Verschlechterung war die Differenz in diesem Fall immer noch geringer als bei der schlechtesten Vorhersage im alten System.

Die Ergebnisse zeigen, dass die Technik des Model Checkings zur Vorhersage der Leistung eines Interaktiven Echtzeitsystems eine äußerst vielversprechende Technik ist. Die Präzision der Vorhersagen konnte durch relativ einfache Mittel beträchtlich gesteigert werden.

9.5 Mögliche technische Verfeinerungen

Im folgenden werden einige Ideen für mögliche technische Verbesserungen der bestehenden prototypischen Implementierung formuliert.

9.5.1 Anbindung an die Ontologie von Simulator X

Simulator X ist nicht nur ein asynchrones Interaktives Echtzeitsystem, es ist auch ein *intelligentes* Interaktives Echtzeitsystem. Die semantische Wissensrepräsentation durch

Ontologien ist elementarer Bestandteil von Simulator X. Hierbei wird die semantische Wissensrepräsentation nicht nur für die Modellierung von Anwendungen verwendet, auch über die Simulator X Middleware selbst können Aussagen getroffen werden.

Jedoch weder die Implementierung des Synchronisationskonzeptes, noch das Profiling oder das Model Checking nutzen derzeit diese Informationen. Sub-Systeme könnten jedoch auch selbst eine Aussage über ihre Synchronisationspunkte treffen. Darüber hinaus können die Profiling-Daten durch semantische Informationen angereichert werden, um einen noch stärkeren Bezug zwischen der Darstellung von Metriken und der erstellten Anwendung herzustellen.

9.5.2 Konfiguration über einen zentralen Ort

Derzeit können die im Model Checker erstellten Modelle eines Schemas nicht als Konfiguration einer Simulator-X-Anwendung verwendet werden. Hierdurch muss ein Schema im Model Checker modelliert werden und anschließend noch einmal als Konfiguration für Simulator X mit der im Kapitel 7.2.1 beschriebenen DSL erstellt werden. Die im vorherigen Abschnitt vorgeschlagene Verwendung der semantischen Wissensrepräsentation aus Simulator X würde es u. U. erlauben, dass die einzelnen Bestandteile in Simulator X entsprechende Informationen über sich geben können, die zum einen die Konfiguration über eine externe Datei erlauben und zum anderen zur automatisierten Generierung eines Modells fürs Model Checking beitragen.

9.5.3 Zusammenführung der bestehenden Tools

Der im Rahmen dieser Arbeit entwickelte Profiler sowie die erstellte Toolchain fürs Model Checking stellen derzeit zwei eigene Anwendungen dar. Neben diesen beiden Werkzeugen existiert für Similator X noch ein Editor (Fischbach, 2011), mit dem Szenen erstellt und verändert werden können. Aus technischer Sicht erscheint es sinnvoll, diese drei Werkzeuge in eines zusammenzuführen. Dies würde es zum einem erlauben, die im Editor erstellten Anwendungen mittels Profiling detailliert zu analysieren. Zum anderen kann, vorausgesetzt die im vorherigen Abschnitt genannte Verfeinerung wurde implementiert, im Hintergrund ein Model Checking erfolgen. Hierdurch kann, ähnlich zur statischen Code-Analyse in der allgemeinen Software-Entwicklung, überprüft werden, ob problematische Ausführungspfade in der Anwendung existieren.

9.5.4 Automatisierte zeitliche Verfeinerung

Während der Verwendung des Model Checkings stellte sich heraus, dass eine iterative zeitliche Verfeinerung des Modells notwendig ist. Diese Prozesse könnten automatisiert werden, um insbesondere innerhalb eines zusammengeführten Tools, welches im vorherigen Abschnitt beschrieben wurde, möglichst automatisiert im Hintergrund zu arbeiten.

9.6 Forschungsausblick

Neben technischen Verfeinerungen ergeben sich aus den Ergebnissen dieser Arbeit neue Fragestellungen für eine weitere Forschung. Einige werden in diesem Abschnitt dargestellt.

9.6.1 Erarbeitung weiterer Schemata

Wie bereits in Abschnitt 9.2 ausgeführt, stellen die vorgestellten Schemata keine vollständige Auflistung aller möglichen Nebenläufigkeits- und Synchronisationsschemata eines asynchronen Interaktiven Echtzeitsystems dar. Die Erforschung weiterer Schemata ist eine sich hieraus ergebene fortgeführte Forschungsfrage. Hierbei könnten z. B. Schemata mit feinerer Granularität erarbeitet werden.

9.6.2 Abbildung weiterer stochastischer Effekte beim Model Checking

Durch die Abbildung des Scheduling-Intervalls der verwendeten Aktor-Implementierung beim Model Checking wurde die Vorhersage der Latenz erheblich verbessert. Der Einbau weiterer stochastischer Effekte könnte hier die Differenz zwischen Vorhersage und Messung weiter reduzieren. Weitere Beispiele hierfür wären das Scheduling des verwendeten Betriebssystem oder das Verhalten der verwendeten Hardware. Dessen Verhalten ist zwar grundlegend deterministisch, entzieht sich jedoch dem Einfluss eines Interaktiven Echtzeitsystems und kann somit auf dieser Abstraktionsebene als stochastisch aufgefasst werden.

9.6.3 Garantie von Frequenzen und Latenzen

Im Rahmen dieser Arbeit wurde im Kapitel 3 ein Synchronisationskonzept für asynchrone Interaktive Echtzeitsysteme vorgestellt. Die Implementierung des Konzepts in Form einer DSL wurde in Kapitel 8 beschrieben und diente dazu, Simulator X im Rahmen der Auswertung zu konfigurieren.

Die Auswertung zeigte, dass diese Frequenzen eher als „Empfehlung“ an das Interaktive Echtzeitsystem zu verstehen sind, als als verbindliche Konfiguration. Es ist nicht sichergestellt, dass Sub-Systeme die konfigurierten Frequenzen einhalten, z. B. wirklich alle 16 ms ein neues Bild gerendert wurde. Insbesondere ist nicht sichergestellt, dass Eingaben innerhalb einer bestimmten Zeit zu einer Reaktion des Systems führen, also eine bestimmte Latenz nicht überschritten wird.

Es existiert derzeit kein Interaktives Echtzeitsystem mit entsprechendem Synchronisationskonzept, welches bestimmte Frequenzen und Latenzen vollständig garantieren kann. Solch eine Entwicklung könnte jedoch auf Grundlage der Ergebnisse dieser Arbeit erreicht werden.

9.6.4 Vorhersage weiterer Metriken

In (Rehfeld et al., 2014) wurde ein Satz von neun Metriken vorgestellt, die auch vom Profiler unterstützt werden. Der Model Checker unterstützt derzeit nur drei diese Metriken. Auch die anderen sechs Metriken sollten sich mit Model Checking vorhersagen lassen. Die stellt mehr als eine technische Verfeinerung dar, da das Modell gegebenenfalls erweitert werden muss, was erneut zu Anpassungen am Model Checker führen könnte.

9.7 Fazit

Verglichen mit den detaillierten Algorithmen, mit denen Systeme in anderen Fachbereichen auf ihr Verhalten und ihre Leistungsfähigkeit geprüft werden, besteht hier im Bereich der Entwicklung Interaktiver Echtzeitsysteme großer Nachholbedarf. Diese Arbeit ist ein erster Schritt in Richtung spezifischerer Tools zur Leistungsmessung und Gesamtanalyse von Interaktiven Echtzeitsystemen. Hierfür wurde Model Checking als neues Verfahren im Kontext von Interaktiven Echtzeitsystemen erfolgreich eingeführt. Die Qualität der Vorhersagen durch Model Checking lagen bereits beim ersten Versuch in einem brauchbaren Bereich. Durch weitere Anpassungen des Modells wurde eine erhebliche Steigerung der Präzision erzielt. Hier ist zu erwarten, dass mit vertretbarem Forschungsaufwand die Differenz zwischen Schätzung und Messung so weit reduziert werden kann, dass die Aussagekraft des geschätzten Verhaltens höher ist als die einer Messung mittels eines Profilers. Die vorgestellten Nebenläufigkeits- und Synchronisationsschemata, die daraus abgeleiteten Grundelemente und die Implementierung in der Form einer DSL haben sich im Kontext eines asynchronen Interaktiven Echtzeitsystems bewährt.

Literaturverzeichnis

Abdelkhalek, A. und Bilas, A. (2004). Parallelization and performance of interactive multiplayer game servers. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 72–.

Abdelkhalek, A., Bilas, A., und Moshovos, A. (2003). Behavior and performance of interactive multi-player game servers. *Cluster Computing*, 6:355–366.

Adl-Tabatabai, A.-R., Kozyrakis, C., und Saha, B. (2006). Unlocking concurrency. *Queue*, 4:24–33.

AMD (2015). *AMD GPU Performance API – User Guide*. AMD.

Andrews, J. (2009). Designing the framework of a parallel game engine. WWW.

Armstrong, J. und Ab, E. T. (1997). The development of Erlang. In *in Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 196–203. ACM Press.

Bajić-Bizumić, B., Petitpierre, C., Chi Huynh, H., und Wegmann, A. (2013). A model-driven environment for service design, simulation and prototyping. In *Exploring Services Science*, volume 143 of *Lecture Notes in Business Information Processing*, pages 200–214. Springer Berlin Heidelberg.

Beeler, D. und Gosalia, A. (2016). Asynchronous timewarp on Oculus Rift `https://developer.oculus.com/blog/asynchronous-timewarp-on-oculus-rift/`. Online.

Behrmann, G., David, A., und Larsen, K. G. (2006). *A Tutorial on* Uppaal *4.0*. Department of Computer Science, Aalborg University, Denmark.

Ben-Ari, M. (2008). *Principles of the Spin Model Checker*. Springer London, 1 edition.

Ben-Ari, M. M. (2010). A primer on model checking. *ACM Inroads*, 1(1):40–47.

Best, M. J., Fedorova, A., Dickie, R., Tagliasacchi, A., Couture-Beil, A., Mustard, C., Mottishaw, S., Brown, A., Huang, Z. F., Xu, X., Ghazali, N., und Brownsword, A. (2009). Searching for concurrent design patterns in video games. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 912–923, Berlin, Heidelberg. Springer-Verlag.

Best, M. J., Mottishaw, S., Mustard, C., Roth, M., Fedorova, A., und Brownsword, A. (2011). Synchronization via scheduling: techniques for efficiently managing shared state. *SIGPLAN Not.*, 46(6):640–652.

Bierbaum, A. D. (2000). *VR Juggler: A Virtual Platform for Virtual Reality Application Developement*. PhD thesis, Iowa State University.

Blow, J. (2004). Game development: Harder than you think. *Queue*, 1:28–37.

Borkar, S. und Chien, A. A. (2011). The future of microprocessors. *Commun. ACM*, 54:67–77.

Breshears, C. (2009). *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly Media.

Bues, M., Gleue, T., und Blach, R. (2008). Lightning: Dataflow in motion. In Latoschik, M. E., Reiners, D., Blach, R., Figueroa, P., und Dachselt, R., editors, *Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), proceedings of the IEEE Virtual Reality 2008 workshop*, pages 7–11. Shaker Verlag.

Burmeister, R. (2013). Reactor: A notation for the specifications of actor systems and its semantics. In für Informatik, G., editor, *Software Engineering 2013 - Fachtagung des GI-Fachbereichs Softwaretechnik*, pages 127–142. Köllen Druck+Verlag GmbH.

Burmeister, R. und Helke, S. (2012). The observer pattern applied to actor systems: A TLA/TLC-based implementation analysis. In *Proc. International Conference on Theoretical Aspects of Software Engineering (TASE 2012)*. IEEE Computer Society.

Cascaval, C., Blundell, C., Michael, M., Cain, H. W., Wu, P., Chiras, S., und Chatterjee, S. (2008). Software transactional memory: Why is it only a research toy? *Queue*, 6:46–58.

Chabukswar, R., Lake, A., und Lee, M. R. (2005). Multi-threaded rendering and physics simulation.

Clarke, E. M. und Emerson, E. A. (1982). Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK. Springer-Verlag.

Clarke, E. M., Emerson, E. A., und Sifakis, J. (2009). Model checking: Algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84.

Creeger, M. (2005). Multicore cpus for the masses. *Queue*, 3:64–ff.

Dayal, A., Woolley, C., Watson, B., und Luebke, D. (2005). Adaptive frameless rendering. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA. ACM.

Deligiannidis, L. (2000). *Dlove: a specification paradigm for designing distributed vr applications for single or multiple users*. PhD thesis, Tufts University, Medford, MA, USA. AAI9955979.

Dept, I. S. (1998). IEEE Standard for a Software Quality Metrics Methodology. Technical report.

Detlefs, D., Flood, C. H., Garthwaite, A., Martin, P., Shavit, N., und Steele, Jr., G. L. (2000). Even better dcas-based concurrent deques. In *Proceedings of the 14th International Conference on Distributed Computing*, DISC '00, pages 59–73, London, UK, UK. Springer-Verlag.

Di Luca, M. (2010). New Method to Measure End-to-End Delay of Virtual Reality. *PRESENCE Teleoperators and Virtual Environments*, 19(6):560–584.

Doherty, S., Detlefs, D. L., Groves, L., Flood, C. H., Luchangco, V., Martin, P. A., Moir, M., Shavit, N., und Steele, Jr., G. L. (2004). Dcas is not a silver bullet for nonblocking algorithm design. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 216–224, New York, NY, USA. ACM.

Drepper, U. (2007). What every programmer should know about memory.

Drepper, U. (2008). Parallel programming with transactional memory. *Queue*, 6:38–45.

Earle, C. und Fredlund, L.-k. (2012). Verification of timed erlang programs using McErlang. In Giese, H. und Rosu, G., editors, *Formal Techniques for Distributed Systems*, volume 7273 of *Lecture Notes in Computer Science*, pages 251–267. Springer Berlin Heidelberg.

Earle, C. B. und Fredlund, L.-∀. (2009). Debugging and verification of multi-agent systems. In Moreno-Díaz, R., Pichler, F., und Quesada-Arencibia, A., editors, *EUROCAST*, volume 5717 of *Lecture Notes in Computer Science*, pages 263–270. Springer.

Earle, C. B. und Fredlund, L.-A. (2010). *McErlang: a Tutorial*.

El Rhalibi, A., England, D., und Costa, S. (2005). Game engineering for a multiprocessor architecture. In *DiGRA 2005 Conference: Changing Views–Worlds in Play*.

Faria, J. M. S. (2008). *Formal Developement of Solutions for Real-Time Operating Systems with TLA+/TLC*. PhD thesis, University of Porto.

Fischbach, M. (2011). Übertragung einer multimodalen Interaktionsumgebung auf einen Scala-basierten Simulationskern. Master's thesis, Universtät Bayreuth.

Fischbach, M., Wiebusch, D., Giebler-Schubert, A., Latoschik, M., Rehfeld, S., und Tramberend, H. (2011). Sixton's curse- simulator x demonstration. In *Virtual Reality Conference (VR), 2011 IEEE*, pages 255 –256.

Frank, L. H., Casali, J. G., und Wierwille, W. W. (1988). Effects of visual display and motion system delays on operator performance and uneasiness in a driving simulator. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 30(2):201–217.

Fredlund, L. und Sánchez Penas, J. (2007). Model checking a video-on-demand server using McErlang. In *Proceedings of the 11th International Conference on Computer Aided Systems Theory (Eurocast 2007)*, volume 4739 of *LNCS*. Springer.

Fredlund, L.-A. und Earle, C. B. (2011). *McErlang User Manual*. Babel group, LSIIS, Facultad de Informatica.

Fredlund, L.-A. und Svensson, H. (2007). McErlang: A model checker for a distributed functional programming language. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 125–136, New York, NY, USA. ACM.

Futuremark (2014). `http://www.futuremark.com/benchmarks/3dmark`.

Gabb, H. und Lake, A. (2005). Threading 3d game engine basics. WWW.

Gajinov, V., Zyulkyarov, F., Unsal, O. S., Cristal, A., Ayguade, E., Harris, T., und Valero, M. (2009). Quaketm: parallelizing a complex sequential application using transactional memory. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 126–135, New York, NY, USA. ACM.

Graham, S. L., Kessler, P. B., und Mckusick, M. K. (1982). Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126.

Grange, S., Conti, F., Rouiller, P., Helmer, P., und Baur, C. (2001). The delta haptic device. Technical report.

Graw, G., Herrmann, P., und Krumm, H. (2000). Verification of uml-based real-time system designs by means of cTLA. In *Object-Oriented Real-Time Distributed Computing, 2000. (ISORC 2000) Proceedings. Third IEEE International Symposium on*, pages 86–95.

Gregory, J. (2009). *Game engine architecture*. A K Peters, first edition.

Herlihy, M. und Moss, J. E. B. (1993). Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21:289–300.

Herrmann, P. und Krumm, H. (1997). Specification of hybrid systems in cTLA+. In *Parallel and Distributed Real-Time Systems, 1997. Proceedings of the Joint Workshop on*, pages 212–216.

Hewitt, C., Bishop, P., und Steiger, R. (1973). A universal modular ACTOR formalism for artificial intelligence. In *IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Hojiat, H. und Kaviani, Z. S. (2012). *Rebeca2 Reference Manual*.

Hollingsworth, J. K., Lumpp, J., und Miller, B. P. (1995). Techniques for performance measurement of parallel programs. *Parallel Computers: Theory and Practice*.

Holzmann, G. (2003). *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition.

Holzmann, G. J. (1997). The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295.

Holzmann, G. J. (2014). Mars code. *Commun. ACM*, 57(2):64–73.

Hu, L. und Gorton, I. (1997). *Performance evaluation for parallel systems: A survey*. Citeseer.

Ilsche, T., Schuchart, J., Schöne, R., und Hackenberg, D. (2015). Combining instrumentation and sampling for trace-based application performance analysis. In Niethammer, C., Gracia, J., Knüpfer, A., Resch, M. M., und Nagel, W. E., editors, *Tools for High Performance Computing 2014*, pages 123–136. Springer International Publishing.

Intel (2012). *Intel Threading Building Blocks - Tutorial*. Intel, 1.22 edition.

Intel (2015). *Intel® VTune™ Amplifier XE 2016 and Intel® VTune™ Amplifier 2016 for Systems Help* `https://software.intel.com/en-us/amplifier_2015_help_win`. Intel.

Jackson, D. (2002). Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290.

Jackson, D. (2012a). *Alloy API examples* `http://alloy.mit.edu/alloy/alloy-api-examples.html`.

Jackson, D. (2012b). *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.

Jackson, D. (2012c). *Tutorial for Alloy Analyzer 4.0* `http://alloy.mit.edu/alloy/tutorials/online/index.html`.

Jafari, A., Khamespanah, E., Sirjani, M., und Hermanns, H. (2014). Performance analysis of distributed and asynchronous systems using probabilistic timed actors. *ECEASST*, 70.

JBullet (2016). JBullet `http://jbullet.advel.cz/`. Web.

Johnson, A. N., Anderson, B. J., Kraimer, M. R., Norum, W. E., Hill, J. O., Lange, R., Franksen, B., und Denison, P. (2012). *EPICS Application Developers's Guide*.

Joselli, M., Zamith, M., Clua, E., Leal-Toledo, R., Montenegro, A., Valente, L., Feij, B., und Pagliosa, P. (2010). An architecture with automatic load balancing for real-time simulation and visualization systems. *JCIS - Journal of Computational Interdisciplinary Sciences*, 1:207–224.

Khamespanah, E. (2014). Rmc `https://github.com/ekhamespanah`.

Khamespanah, E., Sabahi Kaviani, Z., Khosravi, R., Sirjani, M., und Izadi, M.-J. (2012). Timed-rebeca schedulability and deadlock-freedom analysis using floating-time transition system. In *Proceedings of the 2nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions*, AGERE! 2012, pages 23–34, New York, NY, USA. ACM.

Kristinsson, H., Jafari, A., Khamespanah, E., Magnusson, B., und Sirjani, M. (2013). Analysing timed Rebeca using McErlang. In *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2013, pages 25–36, New York, NY, USA. ACM.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565.

Lamport, L. (2002). *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Lamport, L. (2006a). Checking a multithreaded algorithm with +CAL. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 151–163, Berlin, Heidelberg. Springer-Verlag.

Lamport, L. (2006b). Checking a multithreaded algorithm with +CAL. In Dolev, S., editor, *Distributed Computing*, volume 4167 of *Lecture Notes in Computer Science*, pages 151–163. Springer Berlin Heidelberg.

Lamport, L. (2009). The pluscal algorithm language. In *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing*, ICTAC '09, pages 36–60, Berlin, Heidelberg. Springer-Verlag.

Lamport, L. (2012). *A PlusCal User's Manual P-Syntax Version 1.8*. Microsoft Research.

Lamport, L. (2013). *A PlusCal User's Manual C-Syntax Version 1.8*. Microsoft Research.

Lamport, L. (2015). *The TLA+ Hyperbook* `http://research.microsoft.com/en-us/um/people/lamport/tla/hyperbook.html`. Microsoft Research.

Lamport, L. (2016). The TLA toolbox `http://research.microsoft.com/en-us/um/people/lamport/tla/toolbox.html`. Online.

Larsen, K. G., Pettersson, P., und Yi, W. (1997). Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152.

Larson, J. (2008). Erlang for concurrent programming. *Queue*, 6:18–23.

Latoschik, M. E. (2015). *eRis Turorial: Engineering Real-Time Interactive Systems*.

Latoschik, M. E. und Tramberend, H. (2011a). A scala-based actor-entity architecture for intelligent interactive simulations. In *Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), proceedings of the IEEE Virtual Reality 2011 workshop*.

Latoschik, M. E. und Tramberend, H. (2011b). Simulator X: A Scalable and Concurrent Software Platform for Intelligent Realtime Interactive Systems. In *Proceedings of the IEEE VR 2011*.

Lee, E. A. (2006). The problem with threads. *Computer*, 39:33–42.

Lee, K. (2012). Augmented reality in education and training. *TechTrends*, 56(2):13–21.

Lincke, R., Lundberg, J., und Löwe, W. (2008). Comparing software metrics tools. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 131–142, New York, NY, USA. ACM.

Lupei, D., Simion, B., Pinto, D., Misler, M., Burcea, M., Krick, W., und Amza, C. (2010). Transactional memory support for scalable and transparent parallelization of multiplayer games. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 41–54, New York, NY, USA. ACM.

Lutter, T., Pols, A., und Poguntke, M. (2016). Gaming hat sich in allen Altersgruppen etabliert `https://www.bitkom.org/Presse/Presseinformation/Gaming-hat-sich-in-allen-Altersgruppen-etabliert.html`. Online.

Margery, D., Arnaldi, B., Chauffaut, A., Donikian, S., und Duval, T. (2002). Openmask: Multi-Threaded | Modular animation and simulation Kernel | Kit : a general introduction. In Richir, S., Richard, P., und Taravel, B., editors, *VRIC 2002 Proceedings*, pages 101–110.

Mark, W. R., McMillan, L., und Bishop, G. (1997). Post-rendering 3D warping. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, I3D '97, pages 7–ff., New York, NY, USA. ACM.

Mcgurck, H. und Macdonald, J. W. (1976). Hearing lips and seeing voices. *Nature*, 264(246-248).

Miller, D. C. und Thorpe, J. A. (1995). SIMNET: the advent of simulator networking. *Proceedings of the IEEE*, 83(8):1114–1123.

Millington, I. (2010). *Game Physics Engine Development*. CRC Press.

Mine, M. (1993). Characterization of end-to-end delays in head-mounted display systems. *The University of North Carolina at Chapel Hill, TR93-001*.

Mönkkönen, V. (2006). Multithreaded game engine architectures. WWW.

NetBeans (2016). NetBeans RCP `https://netbeans.org/features/platform/`. Online.

Newcombe, C. (2011). Debugging designs. *14th International Workshop on High Performance Transaction Systems, Monterey*.

Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., und Deardeuff, M. (2015). How amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73.

Normand, J.-M., Servières, M., und Moreau, G. (2012). A new typology of augmented reality applications. In *Proceedings of the 3rd Augmented Human International Conference*, AH '12, pages 18:1–18:8, New York, NY, USA. ACM.

NVIDIA (2014). *NVIDIA Performance Toolkit 4.2.0 User Guide*. NVIDIA, 2701 San Tomas Expressway Santa Clara, CA 95050.

NVIDIA (2015). *Profiler User's Guide* *http://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf*. NVIDIA.

Odersky, M. (2011). The Scala Language Specification Version 2.9.

Olukotun, K. und Hammond, L. (2005). The future of microprocessors. *Queue*, 3:26–29.

Pirzada, U. (2014). Destiny's budget makes it more expensive than any movie ever made – budget comparison to other ips. *WCCF Tech*.

Pnueli, A. (1977). The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA. IEEE Computer Society.

Queille, J.-P. und Sifakis, J. (1982). Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, UK. Springer-Verlag.

Rehfeld, S., Latoschik, M. E., und Tramberend, H. (2016). Estimating latency and concurrency of asynchronous real-time interactive systems using model checking. In *Proceedings of the 23rd IEEE Virtual Reality (IEEE VR) conference*.

Rehfeld, S., Tramberend, H., und Latoschik, M. E. (2013). An actor-based distribution model for realtime interactive systems. In *Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), 2013 6th Workshop on*, pages 9–16.

Rehfeld, S., Tramberend, H., und Latoschik, M. E. (2014). Profiling and benchmarking event- and message-passing-based asynchronous realtime interactive systems. In *Proceedings of the 20th ACM Symposium on Virtual Reality Software and Technology*, VRST '14, pages 151–159, New York, NY, USA. ACM.

Reynisson, A. H., Sirjani, M., Aceto, L., Cimini, M., Jafari, A., Ingólfsdóttir, A., und Sigurdarson, S. H. (2014). Modelling and simulation of asynchronous real-time systems using timed Rebeca. *Sci. Comput. Program.*, 89:41–68.

Rohlf, J. und Helman, J. (1994). IRIS performer: a high performance multiprocessing toolkit for real-time 3D graphics. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, pages 381–394, New York, NY, USA. ACM.

Roßbach, M. (2010). Entwicklung einer shaderbasierten grafik-engine für den einsatz in der lehre. Master's thesis, Beuth Hochschule für Technik Berlin.

Ruys, T. C. und Holzmann, G. J. (2004). Advanced spin tutorial `http://spinroot.com/spin/Doc/Spin_tutorial_2004.pdf`. Technical report.

Sarmiento, S. (2004). Real-world case studies: Threading games for high performance on Intel®Processors.

Schneider, F., Easterbrook, S., Callahan, J., und Holzmann, G. (1998). Validating requirements for fault tolerant systems using model checking. In *Requirements Engineering, 1998. Proceedings. 1998 Third International Conference on*, pages 4–13.

Schnipper, M. (2013). Seeing is believing: The state of virtual reality.

Shin, K. und Ramanathan, P. (1994). Real-time computing: a new discipline of computer science and engineering. *Proceedings of the IEEE*, 82(1):6–24.

Sirjani, M., Movaghar, A., Shali, A., und de Boer, F. S. (2004). Modeling and verification of reactive systems using Rebeca. *Fundam. Inform.*, 63(4):385–410.

Srivastava, A. und Eustace, A. (1994). Atom: A system for building customized program analysis tools. *SIGPLAN Not.*, 29(6):196–205.

Stallman, R. M. und the GCC Developer Community (2015). *Using the GNU Compiler Collection*. GNU Press.

Statista (2016). Umsatz mit Computer- und Videospielen in den USA von 2000 bis 2015 (in Milliarden US-Dollar) `http://de.statista.com/statistik/daten/studie/190134/umfrage/umsatz-mit-computer-und-videospielen-in-den-usa/`. Online.

Stauffert, J.-P., Niebling, F., und Latoschik, M. E. (2016a). Reducing application-stage latencies for real-time interactive systems. In *9th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*. IEEE Computer Society.

Stauffert, J.-P., Niebling, F., und Latoschik, M. E. (2016b). Reducing application-stage latencies of interprocess communication techniques for real-time interactive systems. In *in Proceedings of the 23rd IEEE Virtual Reality (IEEE VR) conference*.

Steed, A. (2008). A simple method for estimating the latency of interactive, real-time graphics simulations. In *Proceedings of the 2008 ACM Symposium on Virtual Reality Software and Technology*, VRST '08, pages 123–129, New York, NY, USA. ACM.

Steed, A. und Oliveira, M. F. (2009). *Networked Graphics: Building Networked Games and Virtual Environments*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*.

Sutter, H. und Larus, J. (2005). Software and the concurrency revolution. *Queue*, 3:54–62.

Sweeney, T. (2006). The next mainstream programming language: a game developer's perspective. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 269–269, New York, NY, USA. ACM.

Taylor, II, R. M., Hudson, T. C., Seeger, A., Weber, H., Juliano, J., und Helser, A. T. (2001). VRPN: A device-independent, network-transparent VR peripheral system. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, VRST '01, pages 55–61, New York, NY, USA. ACM.

Teather, R. J., Pavlovych, A., und Stuerzlinger, W. (2009). Effects of latency and spatial jitter on 2D and 3D pointing. In *Virtual Reality Conference, 2009. VR 2009. IEEE*, pages 229–230.

Tramberend, H. (2003). *Avocado : a Distributed Virtual Environment framework*. PhD thesis, Bielefeld University.

Tulip, J., Bekkema, J., und Nesbitt, K. (2006). Multi-threaded game engine design. In *Proceedings of the 3rd Australasian conference on Interactive entertainment*, IE '06, pages 9–14, Murdoch University, Australia, Australia. Murdoch University.

Unity 3D (2016). `https://unity3d.com/`.

Unreal (2016). `https://www.unrealengine.com/what-is-unreal-engine-4`.

Valente, L., Conci, A., und Feijó, B. (2005). Real time game loop models for single-player computer games. In *n Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment*, pages 89–99.

Vardi, M. Y. und Wolper, P. (1994). Reasoning about infinite computations. *Inf. Comput.*, 115(1):1–37.

VisualVM (2014). `http://visualvm.java.net/`.

Waldo, J. (2008). Scaling in games & virtual worlds. *Queue*, 6:10–16.

Wegmann, A., Le, L.-S., Rychkova, I., und Regev, G. (2007). An example of a hierarchical system model using seam and its formalization in alloy. In *EDOC Conference Workshop, 2007. EDOC '07. Eleventh International IEEE*, pages 260–268.

Wiebusch, D. (2015). *Reusability for Intelligent Realtime Interactive Systems*. PhD thesis, Julius-Maximilians-Universität Würzburg.

Willhalm, T., Dementiev, R., und Fay, P. (2012). Intel performance counter monitor - a better way to measure cpu utilization. Technical report, Intel.

YourKit (2014). `http://www.yourkit.com/features/`.

Zhang, H., Gu, M., und Song, X. (2010). Specifying time-sensitive systems with TLA+. In *Computer Software and Applications Conference (COMPSAC), 2010 IEEE 34th Annual*, pages 425–430.

Zhuravlev, S., Saez, J. C., Blagodurov, S., Fedorova, A., und Prieto, M. (2012). Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Comput. Surv.*, 45(1):4:1–4:28.

Zyulkyarov, F., Gajinov, V., Unsal, O. S., Cristal, A., Ayguadé, E., Harris, T., und Valero, M. (2009). Atomic quake: using transactional memory in an interactive multiplayer game server. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 25–34, New York, NY, USA. ACM.

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Spezifikationen

1 Profiling und Benchmarking

1.1 Modul ProfilingAlgorithmBase

MODULE *ProfilingAlgorithmBase*

EXTENDS *Sequences*, *Naturals*, *TLC*

$$
\begin{array}{l}
MessageRecord \triangleq [\\
\quad id : Nat, \\
\quad type : Nat, \\
\quad sender : Nat, \\
\quad receiver : Nat, \\
\quad sent : Nat, \\
\quad beginProcessing : Nat, \\
\quad endProcessing : Nat \\
]
\end{array}
$$

$$
\begin{array}{l}
ProcessRecord \triangleq [\\
\quad id : Nat, \\
\quad received : Seq(MessageRecord), \\
\quad sent : Seq(MessageRecord) \\
]
\end{array}
$$

$$FilterByType(seq,\ type) \triangleq \text{LET } Test(v) \triangleq v.type = type \text{ IN } SelectSeq(seq,\ Test)$$

$$FilterBySent(seq,\ time) \triangleq \text{LET } Test(v) \triangleq v.sent \geq time \text{ IN } SelectSeq(seq,\ Test)$$

$$FilterByBeginProcessing(seq,\ time) \triangleq \text{LET } Test(v) \triangleq v.beginProcessing \geq time \text{ IN } SelectSeq(seq,\ Test)$$

$$FilterByReceiver(seq,\ r) \triangleq \text{LET } Test(v) \triangleq v.receiver = r \text{ IN } SelectSeq(seq,\ Test)$$

$$FilterBySpan(seq,\ begin,\ end) \triangleq \text{LET } Test(v) \triangleq v.beginProcessing > begin \land v.beginProcessing < end \text{ IN } SelectSeq(seq,\ Test)$$

$$Min(a,\ b) \triangleq \text{IF } a < b \text{ THEN } a \text{ ELSE } b$$

$Max(a,\ b) \triangleq$ IF $a > b$ THEN a ELSE b

1.2 Modul DegreeOfParallelism

MODULE *DegreeOfParallelism*

EXTENDS *ProfilingAlgorithmBase*
CONSTANTS *PROCESSES*, *SPAN*

ASSUME $PROCESSES \in Seq(ProcessRecord)$
ASSUME $SPAN \in Nat$

$CompTimeStamp(a,\ b) \triangleq a[1] < b[1]$

```
 --algorithm DegreeOfParallelism
variables
  events = ⟨⟩,
  dop = ⟨⟩,
  d = 0,
  processes = PROCESSES,
  messages = ⟨⟩ ;
begin
  while processes ≠ ⟨⟩ do
    messages := Head(processes).received ;
    while messages ≠ ⟨⟩ do
      events := Append(events, ⟨Head(messages).beginProcessing, TRUE⟩) ;
      events := Append(events, ⟨Head(messages).endProcessing, FALSE⟩) ;
      messages := Tail(messages) ;
    end while ;
    processes := Tail(processes) ;
  end while ;
  events := SortSeq(events, CompTimeStamp) ;
  while events ≠ ⟨⟩ do
    if  Head(events)[2] then
      d := d + 1 ;
     else
      d := d − 1 ;
    end if ;
    dop := Append(dop, ⟨Head(events)[1], d⟩) ;
    events := Tail(events) ;
  end while ;
end
```

1.3 Modul Latency

MODULE *Latency*

EXTENDS *ProfilingAlgorithmBase*
CONSTANTS *PATH*

$PathNode \triangleq [$
  $state : Nat,$
  $messageType : Nat,$
  $proc : ProcessRecord$
$]$

ASSUME $PATH \in Seq(PathNode)$

$BEGIN\_OF\_SIMULATION\_STEP\_TO\_MESSAGE\_IN\_MAILBOX \triangleq 1$
$MESSAGE\_WAITS\_IN\_MAILBOX \triangleq 2$
$BEGIN\_OF\_MESSAGE\_PROCESSING\_TO\_BEGIN\_OF\_SIMULATION\_STEP \triangleq 3$
$BEGIN\_OF\_MESSAGE\_PROCESSING\_TO\_MESSAGE\_IN\_MAILBOX \triangleq 4$
$FINAL\_MESSAGE\_PROCESSED \triangleq 5$
$FINAL\_SIMULATION\_STEP \triangleq 6$

--algorithm *Latency*
variables
  $path = \langle\rangle,$
  $data = \langle\rangle,$
  $buffer1 = \langle\rangle,$
  $buffer2 = \langle\rangle,$
  $endTime = 0$;

procedure *calculateLatency*($lPath$, $start$)
  variable
    $lBuffer1 = \langle\rangle$;
    $lBuffer2 = \langle\rangle$;

begin
  if $Head(lPath).state =$
      $BEGIN\_OF\_SIMULATION\_STEP\_TO\_MESSAGE\_IN\_MAILBOX$ **then**
    $lBuffer1 := FilterByType(Head(lPath).proc.received, Head(lPath).messageType)$;
    $lBuffer1 := FilterByBeginProcessing(lBuffer1, start)$;
    $lBuffer2 := FilterBySent(Head(lPath).proc.sent, Head(lBuffer1).beginProcessing)$;
    $lBuffer2 := FilterByType(lBuffer2, lPath[2].messageType)$;
    $lBuffer2 := FilterByReceiver(lBuffer2, lPath[2].proc.id)$;
    call *calculateLatency*(
      $Tail(lPath)$,

```
      Head(lBuffer2).sent
    );
    return;
  elsif Head(lPath).state =
      MESSAGE_WAITS_IN_MAILBOX then
    lBuffer1 := FilterByType(Head(lPath).proc.received, Head(lPath).messageType);
    lBuffer1 := FilterByBeginProcessing(lBuffer1, start);
    call calculateLatency(
      Tail(lPath),
      Head(lBuffer1).beginProcessing
    );
  elsif Head(lPath).state =
     BEGIN_OF_MESSAGE_PROCESSING_TO_BEGIN_OF_SIMULATION_STEP then
    lBuffer1 := FilterByType(Head(lPath).proc.received, Head(lPath).messageType);
    lBuffer1 := FilterByBeginProcessing(lBuffer1, start);
    call calculateLatency(
      Tail(lPath),
      Head(lBuffer1).beginProcessing
    );
    return;
  elsif Head(lPath).state =
      BEGIN_OF_MESSAGE_PROCESSING_TO_MESSAGE_IN_MAILBOX then
    lBuffer1 := FilterByType(Head(lPath).proc.sent, Head(lPath).messageType);
    lBuffer1 := FilterBySent(lBuffer1, start);
    call calculateLatency(
      Tail(lPath),
      Head(lBuffer1).sent
    );
    return;
  elsif Head(lPath).state =
      FINAL_MESSAGE_PROCESSED then
    lBuffer1 := FilterByType(Head(lPath).proc.received, Head(lPath).messageType);
    lBuffer1 := FilterByBeginProcessing(lBuffer1, start);
    endTime := Head(lBuffer1).endProcessing;
    return;
  elsif Head(lPath).state =
      FINAL_SIMULATION_STEP then
    lBuffer1 := FilterByType(Head(lPath).proc.received, Head(lPath).messageType);
    lBuffer1 := FilterByBeginProcessing(lBuffer1, start);
    endTime := Head(lBuffer1).endProcessing;
    return;
  end if ;
  return;
end procedure ;
```

```
begin if Head(PATH).state =
        BEGIN_OF_SIMULATION_STEP_TO_MESSAGE_IN_MAILBOX then
  buffer1 := FilterByType(Head(PATH).proc.received, Head(PATH).messageType);
  buffer2 := FilterBySent(Head(PATH).proc.sent, Head(buffer1).beginProcessing);
  buffer2 := FilterByType(buffer2, PATH[2].messageType);
  buffer2 := FilterByReceiver(buffer2, PATH[2].proc.id)   ;
  while buffer2 ≠ ⟨⟩ do
    call calculateLatency(
      Tail(PATH),
      Head(buffer2).sent
    );
    data := Append(
      data,
      endTime − Head(buffer1).beginProcessing
    );
    buffer2 := Tail(buffer2);
  end while ;
 else
  buffer1 := FilterByType(Head(PATH).proc.received, Head(PATH).messageType);
  while buffer1 ≠ ⟨⟩ do
    call calculateLatency(
      Tail(PATH),
      Head(buffer1).beginProcessing
    );
    data := Append(
      data,
      endTime − Head(buffer1).sent
    );
    buffer1 := Tailer(buffer1);
  end while ;
 end if ;
end
```

1.4 Modul Consistency

MODULE *Consistency*

EXTENDS *ProfilingAlgorithmBase*
CONSTANTS *SOURCE*, *DESTINATION*, *TRIGGERTYPE*, *UPDATETYPE*, *N*

ASSUME $SOURCE \in ProcessRecord$
ASSUME $DESTINATION \in ProcessRecord$
ASSUME $TRIGGERTYPE \in Nat$

ASSUME $UPDATETYPE \in Nat$
ASSUME $N \in Nat$

$Min(a,\ b) \triangleq$ IF $a < b$ THEN a ELSE b
$Max(a,\ b) \triangleq$ IF $a > b$ THEN a ELSE b

```
  --algorithm Consistency
variables
  buffer1 = ⟨⟩,
  buffer2 = ⟨⟩,
  sourceProcessing = ⟨⟩,
  updateMessages = ⟨⟩,
  targetProcessing = ⟨⟩,
  min = N,
  max = 0,
  inconsistenceCounter = 0;

begin
  sourceProcessing := FilterByType(SOURCE.received, TRIGGERTYPE);
  updateMessages := FilterByReceiver(SOURCE.sent, DESTINATION.id);
  updateMessages := FilterByType(updateMessages, UPDATETYPE);
  targetProcessing := FilterByType(DESTINATION.received, TRIGGERTYPE);

  buffer1 := sourceProcessing;
  while buffer1 ≠ ⟨⟩ do
    buffer2 := updateMessages;
    min := N;
    max := 0;
    while buffer2 ≠ ⟨⟩ do
      if Head(buffer2).sent > Head(buffer1).beginProcessing
         ∧ Head(buffer2).sent < Head(buffer1).endProcessing then
        min := Min(min, Head(buffer2).beginProcessing);
        max := Max(max, Head(buffer2).endProcessing);
      end if ;
      buffer2 := Tail(buffer2);
    end while ;

    if FilterBySpan(targetProcessing, min, max) ≠ ⟨⟩ then
      inconsistenceCounter := inconsistenceCounter + 1;
    end if ;
    buffer1 := Tail(buffer1);
  end while ;
end
```

Eingabedaten Model Checking

1 Zeitdauer von Simulationsschritten

Tabelle 1: Eingabe für die Zeitdauer von Simulationsschritten pro Sub-System und Schema.

	Renderer	Physik	Logik
Schema 1	18,1 ms	2 ms	2 ms
Schema 2	3,4 ms	0,036 ms	2 ms
Schema 3	15 ms	2,2 ms	2 ms
Schema 4	15,9 ms	1,8 ms	2 ms
Schema 5	7,9 ms	2,2 ms	2 ms

2 Gemessenes Laden von Ressourcen

Tabelle 2: Ladezeiten der drei Resource Loader Instanzen aus den Messdaten zu den einzelnen Schemata.

	Loader 1	Loader 2	Loader 3
Schema 1	5670 ms	1407 ms	5766 ms
Schema 2	–	—	–
Schema 3	6171 ms	1616 ms	6220 ms
Schema 4	5635 ms	1466 ms	5740 ms
Schema 5	6817 ms	1896 ms	6852 ms