Mastarbeit

Informationsflusssicherheit in Systemen mit zwei Prozessoren

Jens Sauer

Technische Universität Darmstadt
Fachbereich Informatik
Fachgebiet Modeling and Analysis of Information Systems

1. Prüfer: Prof. Dr.-Ing. Heiko Mantel
2. Prüfer: Prof. Dr. Martin Otto
Betreuer: M.Sc. Matthias Perner

Abgabetermin: 23. Januar 2012

II

Erklärung:

Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen verwendet, sowie Zitate kenntlich gemacht habe.

Darmstadt, den 23. Januar 2012

Jens Sauer

Danksagung

Ich danke Herrn Prof. Dr. Heiko Mantel und Herrn Prof. Dr. Martin Otto für die Möglichkeit, über dieses Thema schreiben zu können. Weiterhin danke ich Matthias Perner und Henning Sudbrock für die vielen Anregungen und Diskussionen sowie die in jeder Hinsicht sehr gute Betreuung.

Außerdem möchte ich mich bei meinen Eltern bedanken, die mir dieses Studium ermöglicht haben und Dorina für ihre stetige und liebevolle Unterstützung.

Zusammenfassung

Um zu garantieren, dass private Daten eines Nutzers von Programmen vertraulich behandelt werden, haben sich Informationsflussanalysen als Technik bewährt. In den bisherigen Arbeiten zu diesem Thema stützen sich die Ausführungsmodelle auf Uni-Prozessor-Systeme. Derzeitig sind jedoch Multi-Core-Systeme Standard bei Desktopcomputern und finden ebenfalls Einzug in mobile Endgeräte. Im Gegensatz zu Uni-Prozessor-Systemen verwenden Multi-Prozessor-Systeme schwache Speichermodelle, um die Ausführungsgeschwindigkeit zu erhöhen. Dadurch sind aber weitere Ausführungen eines Programms möglich, die wiederum zu neuen Sicherheitslücken führen können. Diese Arbeit untersucht Informationsflussanalyse in Systemen mit mehr als einem Prozessor. Hierzu wird eine geeignete Semantik für eine Beispielsprache präsentiert, die auf einem Dual-Prozessor-System aufbaut Weiterhin wird eine neue Sicherheitseigenschaft vorgestellt.

Abstract

Information flow analysis is a common technique to guarentee that private data of a user is kept confidential by a program. Existing security properties are based on semantics of single processor systems. But multicore processors are the current standard in desktop devices and are becoming increasingly popular in mobile devices. Such systems differ from single processor systems in the memory model to optimise the performance. Therefore new possible executions of programs are possible that may introduce new security leaks. This thesis studies information flow security in the context of multi processor systems. It introduces a semantic of a toy language based on a dual processor system. Further it defines a new security property.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Moderne Programme wie Webbrowser transferieren häufig vertrauliche Daten eines Nutzers über Nachrichtenkanäle wie das Internet. Hierzu ist es wichtig, dass der Transfer der Daten nicht von Dritten abgehört werden kann. Mit der Sicherung von Nachrichtenkanälen in Form von Verschlüsselungen beschäftigt sich die Kryptographie. Weiterhin ist wesentlich, dass man dem Programm die sensiblen Daten anvertrauen kann. Z.B. könnten durch einen Programmierfehler vertrauliche Daten öffentlich gemacht werden, indem sie beispielsweise in einer öffentlichen Datei gespeichert werden. Andererseits könnte das Programm aber auch absichtlich die Daten an Dritte weiterleiten, um so vertrauliche Daten auszuspionieren.

Informationsflussanalysen sind eine Technik, um derartige Sicherheitslücken zu finden. Durch die hohe Komplexität von Programmen sind diese Analysen ebenfalls vielschichtig, insbesondere durch den Einsatz von mehreren Threads in einem Programm. Eine automatische Analyse ist daher wünschenswert, sodass bereits bei der Programmierung Sicherheitslücken vermieden werden können.

Jeder Analyse liegt eine Sicherheitseigenschaft zugrunde, die vom Programm erfüllt werden soll. Als Basis dient häufig *Noninterference*. Diese Eigenschaft verlangt, dass das öffentlich beobachtbare Verhalten eines Programms unabhängig von den vertraulichen Eingaben ist. Eine Formalisierung dessen ist *strong security* [SS00]. Hierbei wird die Technik der Bisimulation genutzt, um die Sicherheit gegen einen Angreifer zu modellieren. Der Ansatz fußt auf einer einfachen Multi-Threaded-While-Sprache und einer Semantik dieser, die eine Ausführung auf einem Uni-Prozessor-System angibt.

Heutige Prozessor-Systeme hingegen verwenden häufig mehrere Prozessoren, um die Ausführungsgeschwindigkeit zu erhöhen, sofern sich Aufgaben parallelisieren lassen. Diese Multi-Prozessor-Systeme verwenden wie Uni-Prozessor-Systemen einen gemeinsamen Hauptspeicher. Um die Ausführung

weiter zu beschleunigen, nutzen die Systeme Hardwareoptimierungen wie Caches, die einen Teil des Hauptspeichers beinhalten und durch ihre Nähe zum Prozessor die Kommunikationswege verkürzen. Hierdurch entstehen Nebenwirkungen, die es ermöglichen, dass Prozessoren Speicheroperationen von anderen Prozessoren in unterschiedlicher Reihenfolge sehen. Somit sind weitere Ausführungen im Gegensatz zu einem Uni-Prozessor-System möglich. Die möglichen weiteren Ausführungen werden mit Hilfe von *schwachen Speichermodellen* erklärt.

Bisherige Sicherheitseigenschaften in der Informationsflussanalyse wie *strong security* betrachten nur die Ausführungen eines Programms auf einem Uni-Prozessor-System. Durch die Verwendung von Multi-Prozessor-Systemen entstehen jedoch neue Ausführungsmöglichkeiten, die wiederum zu neuen Sicherheitslücken führen können.

In dieser Arbeit wird eine Semantik für eine Multi-Threaded-While-Sprache angegeben, die Ausführungen auf einem Multi-Prozessor-System mit schwachem Speichermodell modelliert. Die Semantik wird in Hinblick auf die Informationsflussanalyse entwickelt, sodass sich bereits existierende Eigenschaften übertragen lassen. Weiterhin wird eine erste Sicherheitseigenschaft angegeben, die sich an *strong security* orientiert.

In Kapitel 2 wird eine Einführung in die Informationsflusssicherheit gegeben, sowie *strong security* vorgestellt. Hierzu wird eine Semantik für die von *strong security* verwendete Multi-Threaded-While-Sprache angegeben.

In Kapitel 3 werden die Speicheroptimierung, die speziell in Multi-Prozessor-Systemen auftauchen, genauer vorgestellt.

Die Semantik einer Multi-Threaded-While-Sprache basierend auf einem Multi-Prozessor-System wird in Kapitel 4 vorgestellt. Hierbei wird ein vereinfachtes System mit zwei Prozessoren und einem reduzierten schwachen Speichermodell betrachtet.

In Kapitel 5 wird eine Sicherheitseigenschaft, die sich an *strong security* orientiert, vorgestellt. Hierbei werden speziell die Unterschiede zu einem Uni-Prozessor-System hervorgehoben.

Schließlich werden in Kapitel 6 die Ergebnisse zusammengefasst und mit bestehenden Arbeiten vergleicht. Zuletzt wird ein Ausblick auf zukünftige Arbeiten gegeben.

Kapitel 2

Vorbereitende Einführung

2.1 Terminologie und Notation

Im Folgenden ist $\mathbb{N}$ die Menge der natürlichen Zahlen einschließlich der 0. Weiterhin ist $\mathbb{Z}$ die Menge der ganzen Zahlen und $True$ und $False$ ist das größte respektive kleinste Element der kanonischen Booleschen Algebra.

Definition 2.1 (Strikte Ordnung)**.** Eine strikte Ordnung $<$ ist eine zweistellige Relation auf einer Menge M, die die folgenden Eigenschaften hat:

1. Irreflexivität, d.h. $\forall m \in M \quad (\neg(m < m))$,
2. Antisymmetry, d.h. $\forall a, b \in M \quad (a < b \wedge b < a \longrightarrow a = b)$,
3. Transitivität, d.h. $\forall a, b, c \in M \quad (a < b \wedge b < c \longrightarrow a < c)$

Die Menge aller strikten Ordnung auf einer Menge M wird mit $SO(M)$ bezeichnet.

Der transitive Abschluss einer zweistelligen Relation R wird mit R^+ bezeichnet.

Definition 2.2 (Maximales und minimales Element)**.** Sei M eine Menge und $<$ eine strikte Ordnung auf dieser. Ein Element m ist maximal bezüglich $<$ genau dann, wenn

$$\neg\exists k \in M(m < k)$$

Ein Element m ist minimal bezüglich $<$ genau dann, wenn

$$\neg\exists k \in M(k < m)$$

Definition 2.3 (Partielle Funktion)**.** Eine partielle Funktion f von X nach Y ist eine rechtseindeutige Relation und wird mit $f : X \rightharpoonup Y$ bezeichnet.

Die Menge aller Funktionen von X nach Y ist $X \longrightarrow Y$. Die Menge aller partiellen Funktionen von X nach Y wird mit $X \rightharpoonup Y$.

Mit $f[n \mapsto m]$ bezeichnen wir die Funktion

$$f(x) = \begin{cases} m & \text{falls } x = n \\ f(x) & \text{sonst} \end{cases}$$

Weiterhin wird

$$\{0 \mapsto x, 1 \mapsto y\}$$

für die Funktion

$$f : \{0, 1\} \longrightarrow \{x, y\}$$

mit $f(0) = x$ und $f(1) = y$ verwendet.

2.2 Informationsflusssicherheit

Die Vertraulichkeit (*confidentiality*) von Informationen ist häufig ein wichtiges Ziel von Programmen, die vertrauliche Daten wie z.B. Passwörter, verwenden. Diese müssen daher gegen Angriffe gewappnet sein, die versuchen diese anvertrauten Daten zu ermitteln.

Einen Schutz gegen derartige Angriffe bietet z.B. *language-based security*. Hierbei werden Schwachstellen bereits auf Programmiersprachen-Ebene behandelt.

Eine oft verwendete Sicherheitseigenschaft, die Vertraulichkeit garantieren soll, ist *Noninterference*. Dabei soll das öffentlich beobachtbare Verhalten eines Programms, nicht von geheimen oder vertraulichen Eingaben abhängen. Das bedeutet, dass keinerlei Information aus vertraulichen Quellen in öffentlich beobachtbaren Senken fließt, also der *Informationfluss* gesichert ist.

Auf der Ebene der Programmiersprache werden nun Beispiele angegeben, die Informationsflusssicherheit verletzen. Hierbei liegen die Informationen in Speicherorten, auf die durch Variablen zugegriffen werden kann.

Im Folgenden ist l eine nicht-vertrauliche und h eine vertrauliche Variable.

Beispiel 2.4. Das erste Programm weist der Variable l den Wert von h zu. Hierbei geschieht ein direkter Informationsfluss von einer vertraulichen Quelle in eine öffentliche Senke.

$$\mathrm{l} := \mathrm{h}$$

Beispiel 2.5. Das nachstehende Programm macht den Wert der Zuweisung an l vom Wert von h abhängig. Daher ist es möglich basierend auf dem Wert von l zu entscheiden, welchen Wert h hat.

if h=1 **then** l:=1 **else** l:=0 **fi**

Die Verwendung von mehreren Threads macht die Sicherheitsanalyse eines Programms komplexer, da die Ausführung des Programms nicht mehr deterministisch sein muss und von Komponenten des Systems abhängen kann, die das Programm nicht kontrollieren kann. Hierzu zählen z.B. Schedulers.

Beispiel 2.6. Das folgende Programm besteht aus zwei Threads. Hierbei sind die Variablen h_1 und h_2 vertraulich und die restlichen Variablen öffentlich. Der erste Thread vertauscht die beiden Werte von h_1 und h_2 und nutzt dazu die Variable temp. Anschließend setzt er den Wert von temp auf einen Standardwert. Ist es einem Angreifer nur möglich die Eingaben und die finalen Ausgaben eines Programms zu sehen, so ist temp stets 0 und der Wert von h_1 wird nicht offenbart. Wird jedoch der zweite Thread zwischen temp := h_1 und temp := 0 aufgerufen, so erfährt der Angreifer den Wert von h_1 durch l. Daher können sich *data races* in Programmen auf die Sicherheit auswirken.

temp := h_1	l := temp
h_1 := h_2	
h_2 := temp	
temp := 0	

Data races allein sind jedoch nicht der Grund dafür, dass ein Angreifer vertrauliche Informationen durch die Ausführung eines Multi-Threaded-Programms ermitteln kann. Der verwendete Scheduler spielt ebenfalls eine wichtige Rolle. Da man annehmen kann, dass der Angreifer das Verhalten des Schedulers durch Testen approximieren kann, muss dieser bei der Sicherheitsanalyse berücksichtigt werden.

Beispiel 2.7. Das nachstehende Programm ist sicher, sofern zur Ausführung ein Scheduler verwendet wird, der nach jedem Schritt zufällig einen Thread mit gleicher Wahrscheinlichkeit auswählt, der als nächstes ausgeführt wird. Ein Angreifer kann daher nicht auf den Wert von h schließen. Wird das Programm hingegen mit einem *Round-Robin-Scheduler* ausgeführt, kann der Angreifer den Wert von h ermitteln. Ein *Round-Robin-Scheduler* wechselt nach jedem Schritt zum nächsten Thread in Threadpool. Im Fall, dass der letzte Thread ausgeführt wurde, beginnt der Scheduler wieder mit dem ersten Thread. Dadurch ist die Ausführung unter einem solchen Scheduler deterministisch. Für das nachstehende Programm bedeutet das, dass im Fall von h

>0 am Ende der Ausführung l den Wert 1 hat, wohingegen im anderen Fall der Wert von l 0 ist. Somit kann ein Angreifer etwas über den Wert von h erfahren.

if h >0 **then** **fork**(l:=0 l:=1) **else** **fork**(l:=1 l:=0) **fi**

Die Charakterisierung von Vertraulichkeit durch *Noninterference* lässt sich auch so auffassen, dass zwei Ausführungen des selben Programms mit ununterscheidbaren Eingaben für den Angreifer sich nicht im durch den Angreifer beobachtbaren Verhalten unterscheiden.

Eine häufig genutztes Konzept zur Formalisierung dieser Ununterscheidbarkeit ist die Bisimulation. Das bedeutet, dass die Sicherheitseigenschaft, die sicheren Informationsfluss auf Programmiersprachen-Ebene garantieren soll, auf Basis einer Bisimulation für Programme definiert wird.

Ein solcher Ansatz, auf den sich zahlreiche Sicherheitseigenschaften stützen, ist *strong security* [SS00]. Da sich in dieser Arbeit auf diese Sicherheitseigenschaft bezogen wird, wird im Folgenden eine Einführung gegeben. Dazu wird eine Multi-Threaded-While-Sprache samt zugehöriger Semantik eingeführt, die die Ausführung auf einem Uni-Prozessor-System beschreibt. Anschließend wird ein Angreifer festgelegt sowie die Sicherheitseigenschaft definiert.

2.2.1 Multi-Threaded-While-Sprache

Sei Val die Menge aller Werte, die eine Variable annehmen kann. Es wird angenehmen, dass die Menge mindesten die Werte $True$ und $False$ enthält sowie die Menge der ganzen Zahlen $\mathbb{Z}$.

Die Menge der Variablen wird mit Varbezeichnet. Eine Variable verweist auf einen Speicherort.

Die Operatoren, die in Ausdrücken verwendet werden können, sind in der Menge Op zusammengefasst. Diese enthält mindestens $\{and, not, or\}$ und $\{+, -, *, /, =\}$ mit ihren Standardinterpretationen.

Definition 2.8 (Ausdrücke)**.** Die Menge der Ausdrücke $Expr$ ist induktiv definiert:

- für $m \in Val$ ist $m \in Expr$
- für $x \in Var$ ist $x \in Expr$
- wenn $expr_1, \ldots, expr_n \in Expr$, $op \in Op$ und die Arität von op n ist, so ist

$$op(expr_1, \ldots, expr_n) \in Expr$$

Als Sprache dient eine einfache imperative Programmiersprache, die neben Schleifen und Bedingungen eine **skip**-Anweisung enthält. Weiterhin erlaubt sie sequentielle Komposition und das Erstellen neuer Threads durch **fork**.

Definition 2.9 (Programme)**.** Die Menge der Programme Com_{MTL} ist induktiv definiert:

- für $var \in Var$ und $expr \in Expr$ ist $var := expr \in Com_{MTL}$
- $\mathbf{skip} \in Com_{MTL}$
- für $expr \in Expr$,$C_1, C_2 \in Com_{MTL}$ und $D_1 \dots D_n \in Com_{MTL}$ ist

$$C_1; C_2 \in Com_{MTL-L}$$
$$\mathbf{if}\ expr\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi} \in Com_{MTL}$$
$$\mathbf{while}\ expr\ \mathbf{do}\ C_1\ \mathbf{od} \in Com_{MTL}$$
$$\mathbf{fork}(C_1 D_1 \dots D_n) \in Com_{MTL}$$

Das leere Programm wird mit $\langle\rangle$ bezeichnet.

Die Menge aller Vektoren von Threads ist $\overrightarrow{Com_{MTL}} = \bigcup_{n\in\mathbb{N}} Com^n_{MTL}$. Für einen Vektor von Threads wird $\overrightarrow{C}$ oder $\langle C_1 \dots C_n\rangle$ angegeben. Weiterhin bezeichnet $C||\overrightarrow{D}$ die parallele Komposition, die in einem Vektor $\langle C D_1 \dots D_n\rangle$ resultiert.

Für die obige Sprache wird nun eine operationelle Semantik definiert, die beschreibt, wie sich eine schrittweise Änderung des Zustands eines Uniprozessorsystems ergibt.

Ein Teil des Zustands des Systems ist der Speicherzustand.

Definition 2.10. Ein *Speicherzustand* ist eine Funktion $s : Var \longrightarrow Val$, die Variablen auf Werte abbildet.

Basierend auf einem Speicherzustand s lässt sich ein Ausdruck $expr \in Expr$ evaluieren, indem für jede im Ausdruck vorkommende Variable x der Wert $s(x)$ genutzt wird und entsprechend der Struktur des Ausdrucks ausgewertet wird. Evaluiert ein Ausdruck $expr$ in einem Speicherzustand s zu n, so wird $\langle expr, s\rangle \downarrow n$ geschrieben.

Durch die Zuweisung einer Variable x erfolgt eine Änderung eines Speichers s, indem x einen neuen Wert n erhält. Diese Veränderung von s wird mit $s \otimes \{x := n\}$ angegeben.

Die Übergangsregeln der Semantik sind auf zwei Ebenen definiert: Thread- und Systemebene.

$$\frac{\langle C_1, s\rangle \longrightarrow_{\overrightarrow{D}} \langle\langle\rangle, s'\rangle}{\langle C_1; C_2, s\rangle \longrightarrow_{\overrightarrow{D}} \langle C_2, s'\rangle} \qquad \frac{}{\langle \textbf{skip}, s\rangle \longrightarrow_{\langle\rangle} \langle\langle\rangle, s\rangle}$$

$$\frac{\langle C_1, s\rangle \longrightarrow \langle C_1' \overrightarrow{D}, s'\rangle}{\langle C_1; C_2, s\rangle \longrightarrow_{\langle\rangle} \langle (C_1'; C_2)\overrightarrow{D}, s'\rangle}$$

$$\frac{\langle B, s\rangle \downarrow True}{\langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}, s\rangle \longrightarrow_{\langle\rangle} \langle C_1, s\rangle}$$

$$\frac{\langle B, s\rangle \downarrow False}{\langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}, s\rangle \longrightarrow_{\langle\rangle} \langle C_2, s\rangle}$$

$$\frac{\langle B, s\rangle \downarrow True}{\langle \textbf{while } B \textbf{ do } C \textbf{ od}, s\rangle \longrightarrow_{\langle\rangle} \langle C; \textbf{while } B \textbf{ do } C \textbf{ od}, s\rangle}$$

$$\frac{\langle B, s\rangle \downarrow False}{\langle \textbf{while } B \textbf{ do } C \textbf{ od}, s\rangle \longrightarrow_{\langle\rangle} \langle\langle\rangle, s)\rangle}$$

$$\frac{\langle expr, s\rangle \downarrow n}{\langle var := exp, s\rangle \longrightarrow_{\langle\rangle} \langle\langle\rangle, s \otimes \{var := n\}\rangle}$$

Abbildung 2.1: Operationelle Semantik für Threadzustände

Definition 2.11. Ein *Threadzustand* ist Tupel $\langle C, s\rangle$, wobei $C \in Com_{MTL}$ und s ein Speicherzustand ist.

Ein Threadzustand enthält das noch auszuführende Programm sowie den derzeitigen Speicherzustand. Die operationelle Semantik für Threadzustände ist in Abbildung 2.1 zu sehen. Hierbei geht ein Threadzustand in einen anderen über, indem C durch Ausführen der nächsten Anweisung sich zu C' wandelt und der Speicher von s zu s' transformiert wird. Zudem ist am Transitionspfeil gekennzeichnet, welche weiteren Threads durch die Ausführung erzeugt werden.

Bei der Ausführung auf Systemebene entscheidet ein Scheduler, welcher Thread als nächstes ausgeführt wird. Im Folgenden wird eine possibilistischen Scheduler genutzt, um das Modell zu vereinfachen. Dies ist möglich, da *strong security* Scheduler-unabhängig ist, d.h. die Sicherheit hängt nicht vom verwendeten Schedular ab. Daher lässt es sich unabhängig von Schedu-

$$\frac{\langle C_i, s\rangle \longrightarrow_{\overrightarrow{D}} \langle C_i', s'\rangle}{\langle\langle C_1 \ldots C_i \ldots C_n\rangle, s\rangle \twoheadrightarrow \langle\langle C_1 \ldots C_i' \overrightarrow{D} C_{i+1} \ldots C_n\rangle, s'\rangle}$$

Abbildung 2.2: Operationelle Semantik für Systemzustände

lern definieren. Damit bedarf es keiner weiteren Definition eines Schedulers und auf Systemebene besteht ein Zustand aus einem Threadpool und einem Speicherzustand.

Definition 2.12. Ein *Systemzustand* ist ein Tupel $\langle \overrightarrow{C}, s\rangle$, wobei $\overrightarrow{C} \in \overrightarrow{Com_{MTL}}$ ein Threadpool ist und s ein Speicherzustand ist.

Die operationelle Semantik für Systemzustände ist in Abbildung 2.2 zu sehen. Hierbei geht ein Systemzustand in einen anderen über, wobei der i-te Thread ausgeführt wird. Dieser kann gegebenenfalls weitere Threads erstellen, die dann an entsprechender Stelle im Threadpool angeordnet werden. Weiterhin wird ein Thread, nachdem er terminierte, aus dem Threadpool entfernt.

2.2.2 Angreifermodell

Die Sicherheitseigenschaft soll *Noninterference* formalisieren. Dazu ist es notwendig zu klären, was öffentlich beobachtbares Verhalten und öffentliche Eingaben sind. Beides zusammen beschreibt die Fähigkeiten des Angreifers. Im ersten Schritt wird öffentlich beobachtbares Verhalten informell konkretisiert. Anschließend erfolgt eine Approximation der Fähigkeiten im Modell.

Es wird angenommen, dass dem Angreifer der Quellcode des Programms zur Verfügung steht. Weiter ist er in der Lage die öffentlichen Eingaben sowie die finalen öffentlichen Ausgaben einer Ausführung zu sehen. Weiterhin ist er in der Lage das Verhalten des Schedulers zumindest zu approximieren.

Im Modell werden die öffentlichen Eingaben und Ausgaben dadurch approximiert, dass der Angreifer einen Teil des Speichers sehen kann. Die Begründung hierfür ist, dass man annehmen kann, dass sich die finalen öffentlichen Ausgaben eines Programms aus den Werten verschiedener Variablen zusammensetzen. Hierzu wird der Speicher in einen vertraulichen (high confidential) und öffentlichen (low confidential) Teil gegliedert. Jeder Variable wird eine der beiden Sicherheitsdomänen high oder low mittels folgender Funktion zugewiesen.

$$lvl : Var \longrightarrow (\{low, high\}, \leq)$$

Die Menge $(\{low, high\}, \leq)$ ist partiell geordnet mit $low \leq low$, $low \leq high$ und $high \leq high$ und gibt an, in welche Richtung Informationsfluss stattfinden darf. Eine solche Menge mit zugehöriger Funktion lvl wird *security policy* genannt.

Wie bereits erwähnt, wird *Noninterference* mittels einer Bisimulationsbasierten Eigenschaft formalisiert. Hierzu ist es notwendig zu definieren, welche Speicherzustände für einen Angreifer ununterscheidbar sind.

Für den Angreifer sind zwei Speicherzustände ununterscheidbar, wenn die Wert der low-Variablen gleich sind.

Definition 2.13. Zwei Speicher s und s' sind low-gleich (geschrieben $=_L$), wenn gilt

$$\forall x \in Var \quad (lvl(x) = low \implies s(x) = s'(x))$$

2.2.3 Strong Security

Es folgt nun die Definition der Sicherheitseigenschaft *strong security*. Die Eigenschaft beruht auf einer starken Bisimulation. Die Wahl einer starken Bisimulation verstärkt die Fähigkeiten des Angreifers im Modell, da er nun in der Lage ist den Speicher auch während der Ausführung zu sehen. Der Gewinn hierbei ist jedoch, dass sich die Eigenschaft so formulieren lässt, dass sie Scheduler-unabhängig ist. Ein weiterer Nutzen ist, dass die Eigenschaft kompositional ist. Daher lässt sich ein sicheres Programm aus sicheren Teilprogrammen zusammensetzen. Hierdurch ist eine modulare Sicherheitsanalyse in Form eines Typsystems möglich.

Die starke Bisimulation fußt auf der Idee, dass man aus zwei ununterscheidbaren Systemzuständen erneut zwei ununterscheidbare Systemzustände erreicht. Zwei Systemzustände sind hierbei ununterscheidbar, wenn die Speicher für den Angreifer ununterscheidbar sind und die Threadpools ununterscheidbar sind, d.h. die selben Ausführungsmöglichkeiten bieten. Dies wird durch die Forderung erreicht, dass nur Threadpools gleicher Größe in Relation stehen können und die punktweise Ausführung beider Threadpools zu ununterscheidbaren Systemzuständen führt. Hierdurch wird die Eigenschaft Schedular-unabhängig.

Damit folgt die starke Bisimulation der Idee von *Noninterference*, da für zwei Speicher $s_1 =_L s_2$ aber $s_1 \neq s_2$ und ein Programm $\overrightarrow{C}$ gefordert wird, dass das beobachtbare Verhalten gleich ist.

Definition 2.14 (Starke Low-Bisimulation)**.** Eine Relation $R \subset \overrightarrow{Com_{MTL}} \times \overrightarrow{Com_{MTL}}$ ist eine *starke Low-Bisimulation auf Programmen*, wenn

1. $\forall \overrightarrow{C}, \overrightarrow{D} \in \overrightarrow{Com_{MTL}}(\overrightarrow{C}\ R\ \overrightarrow{D} \implies |\overrightarrow{C}| = |\overrightarrow{D}|)$

2.

$$\begin{aligned}
&\forall \overrightarrow{C}, \overrightarrow{D}, \overrightarrow{E} \in \overrightarrow{Com_{MTL}}\ \forall \overrightarrow{C'} \in Com_{MTL}\ \forall s_1, s_2, s_1' \\
&\quad (\overrightarrow{C}\ R\ \overrightarrow{D} \wedge s_1 =_L s_2 \wedge \langle \overrightarrow{C}(i), s_1 \rangle \longrightarrow_{\overrightarrow{E}} \langle \overrightarrow{C'}, s_1' \rangle \\
&\implies \exists \overrightarrow{F} \in \overrightarrow{Com_{MTL}}\ \exists D' \in Com_{MTL}\ \exists s_2' \\
&\quad (\langle \overrightarrow{D}(i), s_2 \rangle \longrightarrow_{\overrightarrow{F}} \langle D', s_2' \rangle \wedge s_1' =_L s_2' \wedge C' \| \overrightarrow{E}\ R\ D' \| \overrightarrow{F}))
\end{aligned}$$

3. R ist symmetrisch

Die Vereinigung aller starken Low-Bisimulation auf Programmen wird mit $\cong_L$ bezeichnet.

Nach [SS00] ist ein Programm $\overrightarrow{C}$ *stark sicher*, wenn gilt

$$\overrightarrow{C} \cong_L \overrightarrow{C}$$

Damit sind Programme unsicher, wenn sie zu keinem anderen Programm einschließlich sich selbst in Relation stehen. Die Bisimulation ist also nicht reflexiv. Ein Beispiel hierfür ist das Programm l := h, wobei $lvl(l) = low$ und $lvl(h) = high$. Ist s_1 und s_2 so gewählt, dass sie sich nur in h unterscheiden, so sind die Speicher für den Angreifer ununterscheidbar. Jedoch führen die einzigen möglichen Ausführungen zu Speichern s_1' und s_2', die nicht mehr ununterscheidbar sind, da l unterschiedliche Werte hat.

Ein Typsystem dient in der Regel dazu, Variablen, Funktionen usw. einer Programmiersprache zu typisieren, um so Laufzeitfehler zu vermeiden. Beispielsweise soll bei einer Zuweisung x := y sichergestellt werden, dass x und y den gleichen Typ haben.

Ein Typsystem besteht aus mehreren Typregeln, die die folgende Form haben:

$$\frac{\vdash e_1 : \tau_1 \ldots \vdash e_n : \tau_n}{\vdash e : \tau}$$

Hierbei bedeutet $\vdash e_1 : \tau_1$, dass sich e_1 typisieren lässt und den Typ τ_1 hat. Die Angabe eines Typs ist nicht notwendig, falls es nur darum geht, ob etwas typisiert werden kann. Sind die Bedingungen oberhalb erfüllt, lässt sich e typisieren. Die Typregeln sind so gestaltet, dass sie die syntaktischen Bestandteile der Programmiersprache erfassen und so eine automatische Typisierung erlauben. Dies wird z.B. genutzt, um Laufzeitfehler bereits zur Compile-Zeit festzustellen.

Ein Sicherheitstypsystem erfüllt den Zweck, dass es ein Programm dahingehend typisiert wird, ob es einer Sicherheitseigenschaft genügt.

Ein Typsystem für Ausdrücke der eingeführten Sprache ist in Abbildung 2.3 zu sehen. Hierbei werden Ausdrücke anhand ihrer Vertraulichkeit *low* oder *high* typisiert. Werte sind nicht vertraulich und werden daher als *low* typisiert. Variablen hingegen werden entsprechend ihrem Sicherheitslevel typisiert. Komplexere Ausdrücke, die aus Ausdrücken zusammengesetzt sind, werden als *low* typisiert, wenn alle Teilausdrücke ebenfalls als *low* typisiert werden. Andernfalls ist der Typ des Ausdrucks *high*.

In Abbildung 2.4 ist ein Sicherheitstypsystem zu sehen, das, wenn es *stark sicher* ist, ein Programm typisiert. Das bedeutet, dass den syntaktischen Bestandteilen eines Programms kein Typ zugewiesen wird. Das Typsystem nutzt die Kompositionalitätsresultate für *strong security* aus, sodass es einfache Regeln für die sequentielle Komposition von Programmen enthält, sowie für Schleifen, Bedingungen und die **fork**-Anweisung. Für Schleifen ist es wichtig, dass der boolesche Ausdruck als *low* typisiert wird. Dadurch wird erreicht, dass vertrauliche Daten nicht die Schleifendurchlaufzahl beeinflussen und so ein Angreifer durch Zählen der Schritte Informationen über diese erhalten kann.

Zuweisungen in *low*-Variablen werden nur typisiert, sofern der Ausdruck ebenfalls *low* ist und somit keine vertraulichen Daten enthält.

Im Fall von Bedingungen existieren zwei Regeln. Falls der Ausdruck der Bedingung B als *low* typisiert ist und die Programme in den Verzweigungen (*if branch* und *else branch*) typisiert werden können, so ist auch die ganze Bedingung typisierbar. Das bedeutet, dass vertrauliche Daten auch hier nicht Entscheidungen beeinflussen, deren Auswirkungen der Angreifer möglicherweise unterscheiden kann. Falls die Bedingung B hingegen als *high* typisiert ist, so wird zusätzlich gefordert, dass das Laufzeitverhalten der beiden Verzweigungen für den Angreifer ununterscheidbar ist ($C_1 \cong_L C_2$) und er somit keine Information über vertrauliche Daten erfahren kann. Die Bedingung $C_1 \cong_L C_2$ in der letzten Typregel ist jedoch eine semantische Nebenbedingung, d.h. sie lässt sich nicht mittels des Typsystems prüfen. Um dies zu Vermeiden existieren Wege die Bedingung zu approximieren, sodass sie erneut eine syntaktische und damit prüfbare Bedingung darstellt [MS04].

Für das obige Typsystem lässt sich mittels Induktion über die Typregeln zeigen, dass ein typisierbares Programm auch *strongly secure* ist.

Satz 2.15. Wenn sich $\vdash C$ herleiten lässt, so gilt $C \cong_L C$.

2.2.4 Weitere Sicherheitseigenschaften

Die Sicherheitseigenschaft *strong security* ist, wie beschrieben, sehr restriktiv, da sie uneingeschränkte Kompositionalität erlaubt und Scheduler-unabhängig

$$\frac{}{\vdash val : low} \qquad \frac{lvl(x) = D}{\vdash x : D}$$

$$\frac{\vdash expr_1 : D_1 \ldots \vdash expr_n : D_n \qquad \forall i \quad D_i \leq D}{\vdash op(expr_1, \ldots, expr_n) : D}$$

Abbildung 2.3: Typsystem für Ausdrücke

$$\frac{}{\vdash \textbf{skip}} \qquad \frac{\vdash C_1 \qquad \vdash C_2}{\vdash C_1; C_2}$$

$$\frac{\vdash B : low \qquad \vdash C}{\vdash \textbf{while } B \textbf{ do } C \textbf{ od}} \qquad \frac{\vdash expr : low \qquad \vdash x : low}{\vdash x := expr}$$

$$\frac{\vdash x : high}{\vdash x := expr} \qquad \frac{\vdash C \qquad \vdash D_1 \ldots \vdash D_n}{\vdash \textbf{fork}(C\overrightarrow{D})}$$

$$\frac{\vdash B : low \qquad \vdash C_1 \qquad \vdash C_2}{\vdash \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}} \qquad \frac{\vdash B : high \qquad \vdash C_1 \qquad \vdash C_2 \qquad C_1 \approx_L^{MPS} C_2}{\vdash \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}}$$

Abbildung 2.4: Typsystem

ist. Zudem gestattet die Eigenschaft nur, dass Informationen aus öffentlichen Quellen in vertrauliche Senken fließen.

Für den praktischen Einsatz ist *strong security* daher oft zu restriktiv. Die uneingeschränkte Kompositionalität wird in vielen Fällen nicht gefordert, da man an bestimmten Stellen eines Programms annehmen kann, dass andere Threads nicht gleichzeitig ausgeführt werden. Der linke Thread in Beispiel 2.6 ist nicht stark sicher, da der öffentlichen Variable temp der Wert einer vertraulichen Variable zugewiesen wird. Dadurch soll sichergestellt werden, dass ein Thread wie der rechte nicht in der Lage ist, den Wert von temp zwischenzeitlich auszulesen. In der Praxis kann und will der Programmierer jedoch sicherstellen, dass der linke Thread ausgeführt wird, ohne dass temp zwischenzeitlich ausgelesen wird. Kann man eine solche Garantie geben, lässt sich eine schwächere aber präzisere Sicherheitseigenschaft angeben [MSS11].

Die Klasse der Scheduler, die in der Praxis vorkommen, sind eine echte Untermenge der Schedulers, für die *strong security* Informationsflusssicherheit garantiert. Damit lässt sich auch hier durch eine genauere Klassifizierung der Schedular eine schwächere Eigenschaft erhalten, die ebenfalls präziser ist [MS10].

Eine weitere Einschränkung von *strong security* ist, dass nur Informationen aus öffentlichen Quellen in vertrauliche Senken fließen dürfen. Bei einer Passwortabfrage hingegen fließt Information in entgegengesetzter Richtung, da die Abfrage bestehend aus dem eingegebenen Passwort und dem Resultat beobachtbar und das gespeicherte Passwort vertraulich ist. Somit gewinnt man mit jeder Anfrage mehr Informationen über das gespeicherte Passwort. Diese „undichte “Stelle ist jedoch erwünscht. Daher existieren auch hierzu Arbeiten ([LM09b], [LM09a]), die eine *Deklassifikation*, wie die Passwortabfrage, an bestimmten Stellen der Ausführung erlauben.

Kapitel 3

Speichermodelle

In gängigen Mehrprozessorsystemen verwenden alle Prozessoren einen gemeinsamen Hauptspeicher. Dies ist eine natürliche Erweiterung des Einprozessorsystems und vereinfacht die Programmierung, da man sich bspw. nicht um Datenpartitionierung kümmern muss.

Im Einprozessorsystem ist das Verhalten des Speichers bzw. das Zugriffsverhalten klar definiert. Führt der Prozessor eine Speicheroperation aus, so ist der Effekt unmittelbar sichtbar, d.h. der Prozessor wartet bis eine Speicheroperationen vollendet wurde und führt erst dann die nächste Operation aus.

Mehrprozessorsysteme hingegen erlauben im folgenden Beispiel, dass beide Prozessoren 0 ausgeben können (unter der Annahme, dass die Initialwerte für x und y 0 sind).

Beispiel 3.1 (Store Buffering)**.**

	Proc 0	Proc 1
1	x := 1	y := 1
2	print y	print x
Mögliche Ausgabe: 0 , 0		

Dieser Effekt tritt durch Hardwareoptimierungen wie Write Buffers auf, von denen jeder Prozessor einen hat und die Schreiboperationen sammeln und so zurückhalten können. Dadurch ist die beschriebene Ausgabe im obigen Beispiel möglich, da beide Prozessoren ihre Schreiboperationen zurückhalten und so im nächsten Schritt die Initialwerte lesen.

Ein *Speicherkonsistenzmodell* oder kurz *Speichermodell* ist eine formale Spezifikation des Speicherverhaltens, die dem Programmierer helfen soll, das Verhalten von Programmen zu verstehen. Hierzu abstrahiert das Modell von den tatsächlichen Speicherkomponenten, wie Caches, Buffers, Pipeline usw.,

und legt fest, welche Werte bei einem Lesezugriff gelesen werden können. Dazu beschreibt es, wie Speicheroperationen ausgeführt werden bzw. umsortiert werden dürfen.

Ein adäquates Speichermodell, das die Ausführung des obigen Beispiels erlaubt, ermöglicht das Auslesen der Werte 0 oder 1 im Zeile 2 für x und y.

3.1 Sequentiell-konsistente Speichermodelle

Die Werte, die bei einem Lesezugriff auf einem Uni-Prozessor-System ausgelesen werden können, sind die des letzten Schreibzugriffs und sind durch die Programmordnung festgelegt. Das entspricht in der Regel der Intuition eines Programmierers. Dieses Speichermodell nennt sich sequentielle Konsistenz oder sequentiell-konsistentes Speichermodell und wurde von Lamport für Mehrprozessorsysteme formal definiert.

Definition 3.2 (Sequentielle Konsistenz [Lam79])**.** [Ein Multiprozessorsystem ist sequentiell konsistent, wenn] das Resultat jeder Ausführung das Gleiche ist, als wenn die Operationen aller Prozessoren in einer sequentiellen Ordnung ausgeführt wurden und die Operationen eines einzelnen Prozessors in der Reihenfolge auftraten, wie es im Programm vorgegeben ist.

In diesem Speichermodell ist die Ausführung des Beispiels 3.1 mit der Ausgabe 0 für beide Prozessoren nicht möglich, da es hierzu nötig wäre, dass die Schreib- und Leseoperationen jeweils entgegen der Programmordnung vertauschen.

Die Verwendung eines sequentiell-konsistenten Speichermodells für Mehrprozessorsysteme würde der gängigen Intuition eines Programmierers entgegenkommen. Es birgt jedoch den Nachteil, dass die Ansichten der Prozessoren auf den Speicher, ohne dass man auf die Programmstruktur achtet, häufig unnötig oft synchronisiert werden müssen (z.B. Caches) und dies die Ausführung verlangsamt.

Hierzu wird das Beispiel 3.3 betrachtet. Das Programm summiert die Zahlen von 1 bis 10.000 nebenläufig auf, d.h. es erstellt einen zusätzlichen Thread, der die Zahlen 5.001 bis 10.000 aufsummiert, während der Hauptthread die Summe der Zahlen 1 bis 5.000 berechnet. Schließlich gibt das Programm die Summe der beiden Zwischenergebnisse aus. Es wird angenommen, dass während der Ausführung des Programms der zweite Thread *sumUpThread* auf einem anderen Prozessor ausgeführt wird. Während beide Threads die Zwischensummen berechnen, bedarf es keiner Synchronisation der Ansichten auf den Speicher, da dies für die einzelne Berechnung nicht relevant ist.

Das heißt, dass sum1 bzw. sum2 jeweils nicht vom anderen Thread verwendet werden. Daher ist von Vorteil, wenn es die Hardware erlaubt, dass die Schreiboperationen in sum1 und sum2, während der Ausführung der Schleifen nicht dem jeweils anderen Prozessor mitgeteilt werden, um so die Laufzeit zu verbessern.

Beispiel 3.3 (Aufsummieren)**.**

```
static int sum1, sum2;

void sumUp()
{
  for (int i = 5001; i <= 10000; i++)
    sum2 += i;
}

int main()
{
  thread sumUpThread(sumUp());

  for (int j = 1; j <= 5000; j++)
    sum1 += j;

  sumUpThread.join();

  printf("%i", sum1 + sum2);
}
```

3.2 Schwache Speichermodelle

Alle gängigen Mehrprozessorsysteme verwenden heutzutage schwache Speichermodelle, die das sequentiell-konsistente Speichermodell aufweichen und es erlauben, dass Speicheroperationen scheinbar entgegen der Programmordnung ausgeführt werden können.

Im Folgenden werden drei Hardwareoptimierungen beschrieben, die eine Aufweichung des sequentiell-konsistenten Speichermodells bewirken.

3.2.1 Hardwareoptimierungen

Geordnete Write Buffer

Die erste Hardwareoptimierung, die wir betrachten, sind geordnete Write Buffer (siehe Abbildung 3.1). Ein Write Buffer ist zwischen einem Prozessor und dem Hauptspeicher geschaltet und sammelt die getätigten Schreiboperationen des Prozessors. Führt ein Prozessor eine Schreiboperationen aus (z.B. x:= 1), so landet der Schreibbefehl im Buffer und wird zu gegebener Zeit (durch die Hardware oder durch eine Anweisung des Programms) auf dem Hauptspeicher ausgeführt. Hierbei ist der Write Buffer eine Queue, sodass Schreibzugriffe in der gleichen Reihenfolge auf den Hauptspeicher ausgeführt werden, wie es in der Programmordnung angegeben ist und demnach wie sie der Prozessor ausführte. Weiterhin wird nur der aktuellste Schreibzugriff zu einem Speicherort im Buffer belassen und die restlichen verworfen.

Führt ein Prozessor einen Lesezugriff aus, so wird erst ermittelt, ob noch ein Schreibzugriff auf den gleichen Speicherort im Write Buffer liegt. Ist dies der Fall, wird der Wert des aktuellsten Schreibzugriffs im Write Buffer gelesen. Andernfalls liest der Prozessor den Wert aus dem Hauptspeicher.

Ein Beispiel für die Auswirkung von Write Buffers wurde bereits in Beispiel 3.1 vorgestellt. Hierbei führen beide Prozessoren parallel $x := 1$ und $y := 1$ aus und beide Schreibzugriffe landen anschließend im Buffer. Das Ausführen von print y durch Proc 0 lässt Proc 0 den Wert von y aus dem Hauptspeicher lesen (im Initialzustand 0), da kein Schreibzugriff in y durch Proc 0 im Buffer liegt. Dadurch entsteht der Eindruck, dass die Lesezugriffe vor den Schreibzugriffen erfolgten.

Nicht-geordnete Write Buffer

Eine Erweiterung der vorherigen Architektur ist der Verzicht auf eine Ordnung der Schreibzugriffe innerhalb eines Write Buffers. Das bedeutet, dass Schreibzugriffe nicht mehr in der gleichen Reihenfolge, wie sie der Prozessor ausführte, auf dem Hauptspeicher ausgeführt werden müssen. Dadurch ergibt sich, dass Prozessoren die Schreiboperationen anderer Prozessoren in unterschiedlicher Reihenfolge sehen können, wie im folgenden Beispiel. Hierbei kann die Maschine zuerst die beiden Zuweisungen von Prozessor 0 ausführen. Diese landen beide im Write Buffer. Im nächsten Schritt entfernt das System die Zuweisung y:=1 aus dem Write Buffer und führt print x und print y aus. Da der initiale Wert von x 0 ist, gibt die erste print-Anweisung 0 aus. Durch das Entfernen von y:=1 aus dem Write Buffer von Prozessor 0 ist der Wert für y nun 1 und print y gibt 1 aus. Prozessor 1 hat somit die Schreiboperation y:=1 vor der Schreiboperation x:=1 wahrgenommen.

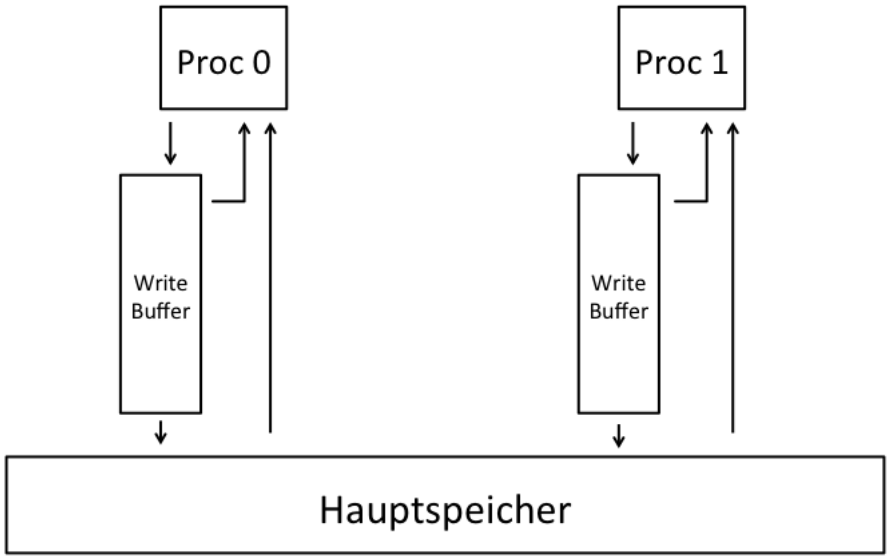


Abbildung 3.1: Write Buffer

Beispiel 3.4 (Message Passing)**.**

	Proc 0	Proc 1
1	x := 1	print x
2	y := 1	print y

Mögliche Ausgabe von Proc 1: 0, 1

Spekulative Leseoperationen

Eine weitere Hardwareoptimierung ist die spekulative Ausführung von Lesezugriffen, dass z.B. durch Prozessor-Pipelines entsteht. Hierbei ist es möglich, dass sich Leseoperationen zu unterschiedlichen Speicherorten nicht blockieren und dadurch entgegen der Programmordnung ausgeführt werden können.

Im folgenden Beispiel ist es erlaubt, dass Proc 1 spekulativ den Wert von x ausliest, bevor dieser in Zeile 1 gesetzt wird und vor Vollendung der while-Schleife.

Beispiel 3.5 (Non-blocking Reads)**.**

	Proc 0	Proc 1
1	x := 1	while (y = 0);
2	y := 1	... := x

3.2.2 Relaxierungen

Die eben vorgestellten Hardwareoptimierungen sind Beispiele für Speicher, die die sequentielle Konsistenz verletzten können. Um die einzelnen Komponenten des Speichers wie Write Buffer, Pipelines, Caches usw. nicht explizit modellieren zu müssen, nutzt man, wie oben bereits erwähnt, das Konzept der Speichermodelle. Das bedeutet, dass man die Auswirkungen von Write Buffers usw. durch eine Abschwächung oder Relaxierung des sequentiellkonsistenten Speichermodells erklärt. Hierbei schwächt man die Bedingung, dass „[...] die Operationen eines einzelnen Prozessors in der Reihenfolge auftreten, wie es im Programm vorgegeben ist“, ab.

Zusätzlich erlauben schwache Speichermodelle, dass Prozessoren ihre eigenen Schreiboperationen sehen können (*Read Own Write Early*), bevor alle anderen Prozessoren sie gesehen haben. Dies erlaubt zum Beispiel die erste beschriebene Architektur mit geordneten Write Buffers. *Read Others' Write Early* hingegen erlaubt, dass Prozessoren Schreiboperationen eines Prozessors bereits sehen können, bevor alle anderen sie gesehen haben. Dies illustriert das folgende Beispiel 3.6. Hierbei ist es möglich, dass Proc 1 die Operation $x \coloneqq 1$ sieht und dann $y \coloneqq 1$ ausführt, wohingegen Proc 2 $x \coloneqq 1$ nicht sieht und so bei print x 0 ausgibt.

Beispiel 3.6. Initialwert für alle Variablen ist 0.

	Proc 0	Proc 1	Proc 2
1	x := 1	if (x = 1)	if (y = 1)
2		y :=1	print x

Mögliche Ausgabe für Proc 2: 0

Man unterscheidet zwischen vier Umsortierungen von Speicheroperationen, die ein schwaches Speichermodell erlauben kann.

Write To Read oder $W \rightarrow R$

Erlaubt ein Speichermodell die *Write To Read*-Relaxierung, so ist es möglich, dass eine Schreiboperation mit einer nachfolgenden Leseoperation vertauscht werden kann, sofern beide Operationen unterschiedliche Speicherorte betreffen. Diese Relaxierung gestattet die Ausführung von Beispiel 3.1. Eine verschränkte Ausführung ist in Abbildung 3.2 zu sehen.

print x
print y
x := 1
y := 1

Abbildung 3.2: Verschränkte Ausführung von Beispiel 3.1

Write To Write oder $W \rightarrow W$

Ein Speichermodell mit *Write To Write*-Relaxierung erlaubt das Vertauschen von zwei aufeinander folgenden Schreiboperationen, die in unterschiedliche Speicherorte schreiben. Dies ermöglicht die Ausführung, die in Beispiel 3.4 illustiert ist. Eine verschränkte Ausführung des Beispiels ist in Abbildung 3.3 gegeben.

y := 1
print y
print x
x := 1

Abbildung 3.3: Verschränkte Ausführung von Beispiel 3.4

Read To Write/Read oder $R \rightarrow W/R$

Die letzten beiden möglichen Relaxierungen treten meist in Kombination auf und erlauben, dass aufeinander folgende Lese- und Speicheroperationen vertauscht werden können, sofern sie unterschiedliche Speicherorte betreffen. Dadurch ist die Ausführung von Beispiel 3.5 möglich, in der der Initialwert von x gelesen wird. Eine verschränkte Ausführung ist ein Abbildung 3.4 zu sehen.

... := x
x := 1
y := 1
while (y = 0);

Abbildung 3.4: Verschränkte Ausführung von Beispiel 3.5

Übersicht der vorhandenen Speichermodelle

Nahezu jede Multiprozessor-Architektur verwendet ein unterschiedliches Speichermodell, dass die obigen Relaxierungen in der einen oder anderen Form kombiniert. Eine Übersicht ist in Abbildung 3.5 gegeben.

Relaxierung	$W \to R$	$W \to W$	$R \to R/W$	Read Others' Write Early	Read Own Write Early
SC					✓
IBM 370	✓				
x86 (TSO)	✓				✓
PSO	✓	✓			✓
Alpha	✓	✓	✓		✓
PowerPC	✓	✓	✓	✓	✓

Abbildung 3.5: Ein ✓ gibt an, ob des Speichermodell die Relaxierung verwendet. SC steht für Sequential Consistency, TSO für Total Store Order und PSO für Partial Store Order. Die letzten beiden sind schwache Speichermodelle, die von SPARC Prozessoren verwendet wurden.

3.2.3 Garantien

Um die oben vorgestellten Relaxierungen dennoch kontrollieren zu können, stellt jede Architektur Prozessoranweisungen bereit. Die Anzahl und Auswirkungen dieser variiert je nach Speichermodell und Architektur. Hierdurch ist es möglich, dass jegliche Programme sequentiell-konsistent ausgeführt werden. Jedoch ist der Preis hierfür, dass sich die Laufzeit, aufgrund der Synchronisation der Speicheransichten, verlängert.

Zusätzlich zu den Synchronisationsanweisungen geben die meisten Speichermodelle, die beiden folgenden Garantien: Zum Einen sehen alle Prozessoren die Schreibzugriffe in einen Speicherort in der gleichen Reihenfolge. Zum Anderen geben die Speichermodelle die sogenannte *Datarace-Frei*-Garantie für Sprachen höherer Ordnung wie Java, C++ usw. welche Locks verwenden. Ein *Datarace* entsteht, wenn zwei Threads zeitgleich auf einen Speicherort zugreifen möchten, wobei mindestens ein Zugriff ein Schreibzugriff ist. Dadurch entsteht ein Wettlauf, wer zuerst seinen Zugriff ausführen darf. In Sprachen höherer Ordnung werden Locks verwendet, die durch ihren wechselseitigen Ausschluss garantieren, dass ein solcher Wettlauf durch den Programmierer kontrolliert oder gar vermieden werden kann. Ein *Datarace*-freies Programm enthält keine *Dataraces*, d.h. im Konflikt-stehende Zugriffe sind

durch die Verwendung von Locks geordnet. Für solche Programme garantieren die Speichermodelle von Java und C++, dass jede Ausführung eines solchen Programms eine sequentiell-konsistente Ausführung existiert mit gleichem Ergebnis. Das bedeutet, dass es zwar erlaubt ist, dass Hardwareoptimierungen verwendet werden, jedoch keine Ausführungen möglich sind, die andere Ergebnisse liefern als sequentiell-konsistente Ausführungen. Z.B. ist das Programm in Beispiel 3.3 Datarace-frei, da der Zugriff auf sum2 nach dem Aufruf sumUpThread.join() folgt und damit immer nach Vollendung des zweiten Threads. Dadurch greift der Hauptthread nie auf sum2 zu, während der zweite Thread dessen Wert verändert.

Bemerkung 1. Für Sprachen wie Java, C# oder C++ sind unabhängig von den vorhandenen Architekturen Speichermodelle Teil der Semantik der Sprachen. Hierzu gehört auch die Semantik von Locks. Die bisher beschriebenen Relaxierungen kommen auch in diesen Speichermodellen vor.

3.2.4 Locks

In Sprachen wie Java oder C++ ist es in der Regel nicht möglich, Prozessoranweisungen direkt auszuführen. Dadurch steht nicht die Möglichkeit bereit, dass man den Speicher mittels der vom Prozessor bereitgestellten Anweisungen kontrollieren kann. Jedoch garantieren die schwachen Speichermodelle, wie oben beschrieben, dass die Verwendungen von korrekt gesetzten Locks dazu führt, dass sich ein Programmierer keine Sorgen machen muss, dass sein Programm nicht sequentiell-konsistent ausgeführt wird.

Hierzu haben Locks neben ihrer Funktion zum wechselseitigen Ausschluss weitere Effekte. Die beiden Lock-Operationen Acquire und Release, die einen Lock l aneignen bzw. freigeben, dienen als Speicherbarrieren im Programm, sodass ein schwaches Speichermodell an diesen Stellen keine Umsortierungen vornehmen kann. Das heißt im Detail, dass keine Speicheroperationen vor eine Acquire-Operation und keine Speicheroperationen nach eine Release-Operation umsortiert werden können. Dadurch wird garantiert, dass Operationen innerhalb eines Acquire-Release-Paares auch innerhalb dieses ausgeführt werden. Im folgenden Beispiel ist nicht möglich, dass $x := 1$ mit $l.acquire$ oder $l.release$ vertauschen kann, wohingegen $y := 1$ mit $l.release$ vertauschen darf.

Beispiel 3.7. Initialwert für alle Variablen ist 0.

	Proc 0	Proc 1
1	l.acquire	while (y = 0);
2	x:= 1	l.acquire
3	l.release	z:= x
4	y := 1	l.release

In manchen Programmiersprachen wie C# sind Acquire- und Release-Operationen *Zwei-Wege-Speicherbarrieren*, d.h. Operationen können in beiden Richtungen nicht Acquire- und Release-Operationen passieren. Für obiges Beispiel heißt das, dass ein schwaches Speichermodell keine Operationen vertauschen kann.

Locks werden zum wechselseitigen Ausschluss verwendet und dienen meist dazu, dass man von verschiedenen Threads aus auf gemeinsame Speicherorte zugreifen möchte. Daher ist es wichtig, dass beim Aneignen eines Locks l, alle Änderungen, die bei einer vorherigen Verwendung von l geschehen sind, für den Prozessor sichtbar sind, der $l.acquire$ ausführt. Das bedeutet, dass in diesem Fall eine Teil-Synchronisation der Speicheransichten durchgeführt wird. In Beispiel 3.7 sieht Proc 1 den Schreibzugriff $x := 1$ nach dem Ausführen von $l.acquire$, da dieser vor $l.release$ in Zeile 3 geschah.

Ein weiteres wichtiges Verhalten von Locks ist, dass es nicht gestattet ist, dass Lock-Operationen miteinander vertauscht werden. Das heißt, dass die Programmordnung angibt, in welcher Reihenfolge sie ausgeführt werden. Hierzu betrachten wir Beispiel 3.8. Durch das Verwenden von Locks ist es Proc 1 nicht möglich die beiden Anweisungen $x := 1$ und $y := 1$ in unterschiedlicher Reihenfolge zu sehen.

Beispiel 3.8. Initialwert für alle Variablen ist 0.

	Proc 0	Proc 1
1	l.acquire	print x
2	x:= 1	print y
3	l.release	
4	z.acquire	
5	y := 1	
6	z.release	

3.3 Informationsflusssicherheit

Die Verwendung schwacher Speichermodelle bzw. Hardwareoptimierungen in Multiprozessorsystemen führt dazu, dass Programme, die in Einprozessorsystemen eine gewisse Sicherheit erfüllen, diese nicht mehr erfüllen.

	Thread 0	Thread 1
1	$flag_0 := True$	$flag_1 := True$
2	$turn := 1$	$turn := 0$
3	**while** $flag_1 = True \; and \; turn = 1$ **do**	**while** $flag_0 = True \; and \; turn = 0$ **do**
4	// Warten	// Warten
5	**od**	**od**
6	// Kritische Sektion	// Kritische Sektion
7	$flag_0 := False$	$flag_1 := False$

Abbildung 3.6: Peterson-Algorithmus: Die Initialwerte sind: $flag_0 = False$, $flag_1 = False$ und $turn = 0$.
Zur Annotation des Programms werden hier Kommentare verwendet, die mit // beginnen.

Für jede der drei genannten Relaxierungen wird ein Beispiel angegeben. Zuerst wird mit einem Beispiel für ein Speichermodell begonnen, das ausschließlich $W \rightarrow R$-Relaxierungen erlaubt.

In Abbildung 3.6 ist eine Implementierung des Peterson-Algorithmus angeführt, der eine Lösung des Problems des wechselseitigen Ausschluss ist. Die Implementierung funktioniert in einem Ein-Prozessor-System korrekt, d.h. nur ein Thread kann stets in eine der beiden kritische Sektionen gelangen.

Werden die beiden Threads jedoch in einem Multi-Prozessor-System ausgeführt und gleichmäßig auf die Prozessoren verteilt, so ist es möglich, dass beide Threads zeitgleich ihre kritische Sektion erreichen. Dies ist möglich, da beide Prozessoren die Schreiboperation der ersten beiden Zeilen 1 und 2 in ihrem Write Buffer belassen. Dadurch evaluieren die Bedingungen der While-Schleifen in Zeile 3 jeweils zu $False$, wodurch beide die kritischen Sektionen im nächsten Ausführungsschritt erreichen.

Enthalten die kritischen Sektionen sicherheitsrelevanten Quellcode, der nicht gleichzeitig ausgeführt werden darf, so entsteht hierdurch eine Sicherheitslücke.

Das obige Beispiel nutzte ein Speichermodell, das $W \rightarrow R$- Relaxierungen erlaubt. Betrachtet man ein Modell, das zusätzlich $W \rightarrow W$-Relaxierungen zulässt, so wird das Programm in Abbildung 3.7 unsicher. Prozessor 1 ist es erlaubt, dass er die Schreiboperationen in Zeile 4 und 5 von Prozessor 0 in umgekehrter Reihenfolge zur Programmordnung sieht. Dadurch kann er die Schleife in Zeile 1 verlassen, obschon *temp* noch den Wert von h_1 hat. Somit kann ein Angreifer abhängig von der Anzahl der Schleifendurchgänge in Zeile 2 etwas über h_1 erfahren.

Zuletzt wird eine Maschine mit schwachem Speichermodell betrachtet, das

	Proc 0	Proc 1
1	$temp := h_1$	**while** $flag = 0$ **do od**
2	$h_1 := h_2$	**while** $temp < 10$ **do**
3	$h_2 := temp$	$temp := temp + 1$
4	$temp := 0$	**od**
5	$flag := 1$	

Abbildung 3.7: Die Initialwerte sind für $temp = 0$ und $flag = 0$. Die Variablen h_1 und h_2 enthalten nicht öffentliche Daten, wohingegen die restlichen Variablen alle öffentlich sind.

	Proc 0	Proc 1
1	$z := temp$	**while** $flag = 0$ **do od**
2	$flag := 1$	$temp := h_1$

Abbildung 3.8: Die Initialwerte sind $temp = 0$, $flag = 0$. Die Variable h_1 enthält nicht öffentliche Daten, wohingegen $temp$ sowie die restlichen Variablen öffentlich sind.

$R \to W$-Relaxierungen erlaubt und das Programm in Abbildung 3.8 ausführt. Die Relaxierung ermöglicht, dass Prozessor 1 spekulativ den Wert von h_1 in $temp$ schreibt, bevor die Schleife in Zeile 1 erfolgreich durchlaufen wurden. Somit kann Prozessor 0 in Zeile 1 den Wert von h_1 aus $temp$ lesen und z zuweisen. Dadurch entsteht an dieser Stelle ein verbotener Informationsfluss, der in einem Uni-Prozessor-System nicht möglich ist.

Allen hier aufgezeigten Beispielen ist gemein, dass eine Sicherheitslücke durch unzureichende Synchronisation entsteht. Das bedeutet, dass durch die Verwendung von Locks oder Synchronisationkommandos der Architektur die Programme modifiziert werden können und so die beschriebenen Ausführungen nicht möglich sind.

Kapitel 4

Ein Multiprozessorsystem

In diesem Kapitel wird eine Semantik für Programme vorgestellt, die auf einem Multi-Prozessor-System ausgeführt werden. Hierbei wurde eine neue Semantik entwickelt, da die bestehenden Arbeiten ([JPR10], [DRD10], [SSO+10], [BOS+11]) die nachstehenden Anforderungen nicht erfüllen. Für einen besseren Vergleich mit bestehenden Arbeiten in der Informationsflusssicherheit ([LM09a], [MS10], [LM09b], [MSS11]) soll als Programmiersprache eine angepasste Multi-Threaded-While-Sprache dienen. Diese ist eine Sprache höherer Ordnung und keine Maschinensprache. Zudem soll eine ähnliche Notation der Thread- und Systemzustände verwendet werden, um Ähnlichkeiten in der Definition von Sicherheitseigenschaften aufzeigen zu können. Die Semantik soll eine operationelle Semantik sein, die die einzelnen Schritte des Systems modelliert (*small-step semantic*). Dadurch ist es möglich auch Sicherheitseigenschaften basierend auf (starken) Bisimulationen zu definieren. Zuletzt soll die Semantik die Ausführung eines Programms auf einer Multi-Prozessor-Maschine modellieren. Das bedeutet die Verwendung eines schwachen Speichermodells und die Möglichkeit einer echt parallelen Ausführung von Anweisungen.

Das Kapitel beginnt mit der Definition der Programmiersprache und der informellen Beschreibung der Laufzeitumgebung einschließlich des verwendeten Speichermodells. Anschließend erfolgt die Formalisierung der Semantik. Das Kapitel schließt mit einer Übersicht der Eigenschaften der Semantik, die u.a. durch Beispiele erörtert werden. Zudem werden Alternativen und Erweiterungen vorgestellt.

4.1 Multi-Threaded-While-Sprache

Als Basis der folgenden Sprache MTL-L dient die in 2.2.1 vorgestellte Programmiersprache MTL. Die Sprache MTL-L hat im Gegensatz zu MTL kein Sprachkonstrukt zur Erstellung neuer Threads (d.h. es fehlt **fork**). Dafür enthält MTL-L Locks. Diese sind in Multi-Prozessor-Systemen notwendig, um korrekt synchronisierte Programme schreiben zu können (siehe hierzu die Beispiele 3.7). Daher ist neben der Menge $Var\ Lock$ die Menge der sogenannte *Lockvariablen*, die ausschließlich zum Setzen von Locks verwendet werden.

Definition 4.1 (Programme in MTL-L)**.** Die Menge der Programme in MTL-L $Com_{MTL\text{-}L}$ ist induktiv definiert:

- für $var \in Var$ und $expr \in Expr$ ist $var := expr \in Com_{MTL\text{-}L}$
- für $l \in Lock$ ist $l.acquire \in Com_{MTL\text{-}L}$ und $l.release \in Com_{MTL\text{-}L}$
- $\mathbf{skip} \in Com_{MTL\text{-}L}$
- für $expr \in Expr$ und $C_1, C_2 \in Com_{MTL\text{-}L}$ ist

$$C_1; C_2 \in Com_{MTL\text{-}L}$$
$$\mathbf{if}\ expr\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi} \in Com_{MTL\text{-}L}$$
$$\mathbf{while}\ expr\ \mathbf{do}\ C_1\ \mathbf{od} \in Com_{MTL\text{-}L}$$

Das leere Programm wird ebenfalls mit $\langle\rangle$ bezeichnet. Für ein Programm $C \in Com_{MTL\text{-}L}$ wird mit C_0 die erste Anweisung (Zuweisung, Acquire, Release, Skip, if, while) im Programm bezeichnet.

4.2 Beschreibung der Laufzeitumgebung

Zur Vereinfachung des Modells beschränkt sich die Semantik auf ein Dual-Prozessor-System. Zudem verwendet das System eine reduzierte Anzahl an Hardwareoptimierungen, sodass das zugehörige schwache Speichermodell nicht alle in vorherigem Kapitel vorgestellten Relaxierungen erlaubt. Das System hat einen gemeinsam genutzten Hauptspeicher und für jeden Prozessor einen ungeordneten Write Buffer. Den Prozessoren ist es erlaubt, dass sie die Werte eigener Schreiboperationen lesen können, bevor diese vom anderen Prozessor gesehen wird. Da die Sprache keine Threaderstellung erlaubt, besteht ein Programm, das auf diesem System läuft, aus einem oder mehreren Threads. Am Anfang der Ausführung wählt das System eine Verteilung der Threads

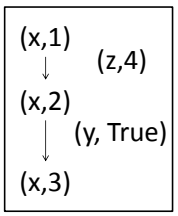


Abbildung 4.1:

auf die beiden Prozessoren. Zwischen der Ausführung der Threads durch die Prozessoren bestimmt das System, welche Schreiboperationen eines anderen Prozessors gesehen werden. Dies geschieht durch das teilweise Leeren der Write Buffer. Zudem ist es dem System erlaubt im Fall, dass ein Write Buffer mehrere Schreiboperationen in eine Variable enthält, nicht alle Schreiboperationen in den Hauptspeicher zu schreiben, sondern ältere Schreiboperationen zu verwerfen. Dadurch nimmt ein Prozessor nicht notwendigerweise jede Schreiboperation des anderen Prozessors wahr.

Ein Beispiel hierzu ist in Abbildung 4.1 gegeben. In der Abbildung sieht man den Write Buffer von Prozessor 0. Dieser enthält drei Schreiboperationen in x. Die Pfeile geben die Ordnung an, in der sie ausgeführt wurden. Dem System ist erlaubt erst $(x, 1)$ dann $(x, 2)$ und anschließend $(x, 3)$ in den Hauptspeicher zu schreiben. Weiterhin ist es aber auch in der Lage nur $(x, 3)$ in den Hauptspeicher zu schreiben und die anderen beiden Schreiboperationen zu verwerfen. In diesem Fall würde Prozessor 1 nur die Schreiboperation $(x, 3)$ wahrnehmen.

Aufgrund der beschriebenen Hardware erlaubt das zugehörige Speichermodell die Relaxierungen *Write To Read* und *Write To Write*. Weiterhin hat das Speichermodell die Eigenschaft *Read Own Writes Early* und erlaubt nicht, dass Prozessoren Schreibzugriffe in einen Speicherort in unterschiedlicher Reihenfolge sehen. Die Lockoperationen unterliegen den Einschränkungen, dass sie im Speichermodell nicht vertauschen. Zudem kann eine Acquire-Anweisung nicht mit einer nachfolgenden Anweisung des gleichen Prozessors vertauschen. Gleiches gilt für eine Release-Anweisung und eine zuvor getätigte Anweisung.

Bemerkung 2. Der hier vorgestellte Speicher hat ein schwaches Speichermodell, das schwächer ist als das des x86-Systems, da es zusätzlich *Write-To-Write*-Relaxierungen erlaubt. Weiterhin ist es stärker als die Speichermodelle von PowerPC- und ARM-Systemen und stärker als die Speichermo-

delle von Java und C++11, weil alle vier zusätzlich $ReadToWrite/Read$-Relaxierungen erlauben.

4.3 Semantik von MTL-L

Der entscheidende Unterschied der folgenden Semantik gegenüber Semantiken von Uni-Prozessor-Systemen ist der Speicher. Im einfachen Fall ist der Speicher eine Funktion, die Variablen $v \in Var$ einen Wert in Val zuweist.

Dieser Ansatz ist jedoch zur Modellierung der verschiedenen Ansichten auf den Speicher durch die Prozessoren unzureichend. Daher wird hier ein Ansatz gewählt, der [JPR10] folgt. Hierbei ist der Speicher eine Menge von getätigten Schreiboperationen einschließlich der getätigten Lockoperationen.

Jeder Prozessor hat eine eigene Ansicht auf den Speicher, die in Form einer Ordnung auf die Menge der Schreiboperationen formalisiert sind. Die getätigten Lockoperationen werden berücksichtigt, um die Konsequenzen für die Sichtbarkeit von Schreiboperationen durch die Verwendung von Locks zu modellieren (siehe hierzu Beispiel 3.7 und 3.8).

Sei $Proc = \{0, 1\}$ die Menge der Prozessoren. Im Folgenden ist p ein Prozessor und $1 - p$ der entsprechend andere Prozessor.

4.3.1 Speicher

Der Speicher hat eine Historie aller Schreiboperationen, die während der Ausführung durchgeführt wurden. Eine Schreiboperation modelliert das Schreiben eines Werts in einen Speicherort durch einen Prozessor. Anfänglich enthält die Historie Schreiboperationen in jede Variable, die im Programm verwendet wird, um den Initialwert festzulegen. Als Symbol dient hierzu $\perp$ anstatt der Nennung eines Prozessor.

Definition 4.2 (Schreiboperation)**.** Eine Schreiboperation ist ein Element der Menge aller Schreiboperationen $(\{\perp\} \cup Proc) \times (Var \cup Lock) \times Val$, die mit WO bezeichnet wird.

Die erste Komponente des Triples gibt an, welcher Prozessor den Schreibzugriff durchführte. Die anderen beiden Komponenten geben die Variable und den Wert an.

Die kanonischen Projektionen sind

$$\begin{aligned} proc &: WO \longrightarrow (\{\perp\} \cup Proc), & (p,x,n) &\mapsto p \\ var &: WO \longrightarrow Var, & (p,x,n) &\mapsto x \\ val &: WO \longrightarrow Val, & (p,x,n) &\mapsto n \end{aligned}$$

Tätigt ein Prozessor p eine l.acquire-Anweisung, so wird dem Speicher eine Schreiboperation $(p, l, 1)$ hinzugefügt. Im Fall einer l.release-Anweisung ist es $(p, l, 0)$.

Da ein Prozessor in einen Speicherort mehrmals den gleichen Wert während einer Ausführung schreiben kann, ist eine Menge von Schreiboperationen unbrauchbar. Daher wird eine Indexmenge zur Indizierung der Schreiboperationen genutzt, sodass gleiche Schreiboperationen unterschieden werden können.

Definition 4.3 (Schreibhistorie)**.** Eine Schreibhistorie ist eine partielle Funktion von einer Indexmenge $\mathcal{J}$ nach WO, d.h.

$$wh : \mathcal{J} \rightharpoonup WO$$

Das Prädikat $AcquireOp(j)$ ist wahr, wenn $j \in dom(wh)$ auf eine Schreiboperation $(p, l, 1)$ mit $l \in Lock$ abbildet. Für eine Release-Operation $(p, l, 0)$ wird $ReleaseOp(j)$ verwendet.

Als Indexmenge wird im Folgenden $\mathcal{J} = \mathbb{N}$ genutzt. Dadurch ist es einfach, einen noch nicht verwendeten Index zu bestimmen, indem man den Nachfolger des größten verwendeten Index wählt. So ist es auch möglich anhand der Indizes die Reihenfolge der Schreiboperationen während der Ausführung zu bestimmen.

Der Speicher besteht aus mehreren Komponenten. Ein Teil ist eine Schreibhistorie wh, wie sie oben definiert ist. Jeder Prozessor hat eine Sicht auf diese Historie in Form einer strikten Ordnung auf dem Definitionsbereich der Schreibhistorie. Eine strikte Ordnung wurde gewählt, da ein Prozessor nicht alle getätigten Schreiboperation des anderen Prozessor wahrnimmt und daher alle Schreiboperation nicht in einer totalen Ordnung sieht. Zudem ist die Sicht nicht reflexiv (ein Prozessor sieht eine Schreiboperation nicht vor ihr selbst), aber antisymmetrisch und transitiv. Um den Ansatz erweiterbar zu halten, wird eine Funktion V verwendet, die Prozessoren auf deren Sichten abbildet.

Die Ansichten der Prozessoren auf die Schreibhistorie geben an, welche Schreiboperationen ein Prozessor bereits gesehen hat und in welcher Reihenfolge. Bisher nicht gesehene Schreiboperationen stehen in der strikten

Ordnung in keiner Relation zu anderen Schreiboperationen. Ausgenommen hiervon sind die initialen Schreiboperationen, die jeder Prozessor anfänglich gesehen hat. Der aktuelle Wert einer Variable v ist durch die zuletzt gesehen Schreiboperation in v gegeben, d.h. die maximale gesehene Schreiboperation in v. Die Ordnung der bereits gesehenen Schreiboperationen respektiert das zuvor beschriebene schwache Speichermodell.

Die Nutzung von Locks wirkt sich auf die Sichtbarkeit der Schreiboperationen aus. Um diese Bedingungen zu erfassen, wird eine weitere strikte Ordnung L verwendet. Diese *Lockordnung* ordnet die Schreiboperationen entsprechend der Semantik der Locks an. Z.B. ist eine Schreiboperation von p kleiner als alle vorangegangen Acquire-Operation von p, da das Speichermodell nicht erlaubt, dass Anweisung innerhalb eines Acquire-Release-Paares aus diesem heraus verschoben werden. Die Prozessoren können Schreiboperationen des anderen Prozessors nicht entgegen der Lockordnung sehen.

Die Zustände der Locks (angeeignet und frei) werden in einer Funktion ls festgehalten, die einen Lock $l \in Lock$ auf einen Prozessor (sofern der Lock angeeignet ist) oder $\perp$ (sofern der Lock frei ist) abbildet.

Um sowohl einen einfachen Zugriff auf die zuletzt getätigte Schreiboperation zu erlauben als auch eine einfache Berechnung eines noch nicht verwendeten Index zu bieten, ist die letzte Komponente des Speicher der Index der zuletzt getätigten Schreiboperation im System. Dieser wird mit i bezeichnet.

Es wird nun ein Beispiel für einen Speicher in Abbildung 4.2 vorgestellt, der aus der Ausführung des folgenden Programms resultiert.

	Proc 0	Proc 1
1	x := 1	l.acquire
2	y := 1	t := 0
3	l.acquire	l.release
4	x := 1	
5	l.release	

Das Programm wurde hierbei, wie folgt, ausgeführt. Zuerst führte Prozessor 0 die ersten beiden Zeilen parallel zu den ersten drei Zeilen von Prozessor 1 aus. Anschließend wartete Prozessor 0 bis der Lock l wieder frei war und führte danach das verbleibende Programm aus. Nachdem Prozessor 1 t := 0 ausführte, lies das System Prozessor 1 die bereits getätigten Schreiboperationen in x und y in umgekehrter Reihenfolge sehen. Bevor Prozessor 0 den Lock l aneignete, synchronisierte das System die Ansichten der Prozessoren dahingehend, dass Prozessor 0 nun die Schreiboperation t:=0 sieht.

Die Ansichten der Prozessoren sowie die Lockordnung sind in Abbildung 4.2 zu sehen. Hierbei werden anstatt der Indizes die Schreiboperation direkt gezeigt. Zudem sind die strikten Ordnungen durch direkte Graphen dar-

gestellt, wobei zur besseren Übersicht Kanten entfernt wurden, sofern ein längerer Pfad existiert.

Definition 4.4 (Speicher)**.** Der *Speicher* des Systems ist ein 5-Tupel (wh, V, L, ls, i), wobei wh eine Schreibhistorie ist,

$$V : Proc \longrightarrow SO(dom(wh))$$

die Ansichten der Prozessoren auf die Historie darstellt, $L \in SO(dom(wh))$ die durch die Verwendung von Locks vorgegebene Ordnung der Schreiboperationen ist,

$$ls : Lock \rightharpoonup (\{\bot\} \cup Proc)$$

der Lockzustand und i der Index der zuletzt getätigten Schreiboperation ist.

Die Menge aller *Speicher* werden mit *MEM* bezeichnet. Die Elemente dieser Menge werden in der Regel mit $mem_1, mem_2, \ldots$ oder $s_1, s_2, \ldots$ bezeichnet.

Im Initialzustand des Systems enthält der Speicher für jede im Programm vorkommende Variable *var* und Lockvariable l eine Schreiboperation, die den Initialwert festlegt. Die Ansichten beider Prozessoren sind im Initialzustand die selben und ordnen die initialen Schreibzugriffe nicht, d.h. $V(0) = V(1) = \varnothing$. Formal ist dies, wie folgt, definiert:

Definition 4.5 (Initialer Speicherzustand)**.** Sei $\overrightarrow{C}$ ein Programm und $X \subset Var$ die Menge der in einem Programm vorkommenden Variablen, sowie $X_L \subset Lock$ die Menge der Lockvariablen. Für jedes $x \in X$ sei $iv(x) \in Var$ der Initialwert der Variable im Programm. Weiterhin sei $n = |X| + |X_L|$ und $\pi : \{0, \ldots, n-1\} \longrightarrow (X \cup X_L)$ eine bijektive Funktion. Der initiale Speicherzustand zu $\overrightarrow{C}$ ist:

$$wh : \{0, \ldots, n-1\} \subset \mathbb{N} \rightharpoonup WO$$
$$i \mapsto \begin{cases} (\bot, \pi(i), iv(\pi(i))) & \text{falls } \pi(i) \in X \\ (\bot, \pi(i), 0) & \text{sonst} \end{cases}$$

$$V(0) = V(1) = L = \varnothing$$

$$ls : X_L \subset Lock \rightharpoonup (\{\bot\} \cup Proc), \quad l \mapsto \bot$$

$$i = n - 1$$

Der Index der eindeutigen initialen Schreiboperation in eine Variable x wird mit $init_x$ bezeichnet.

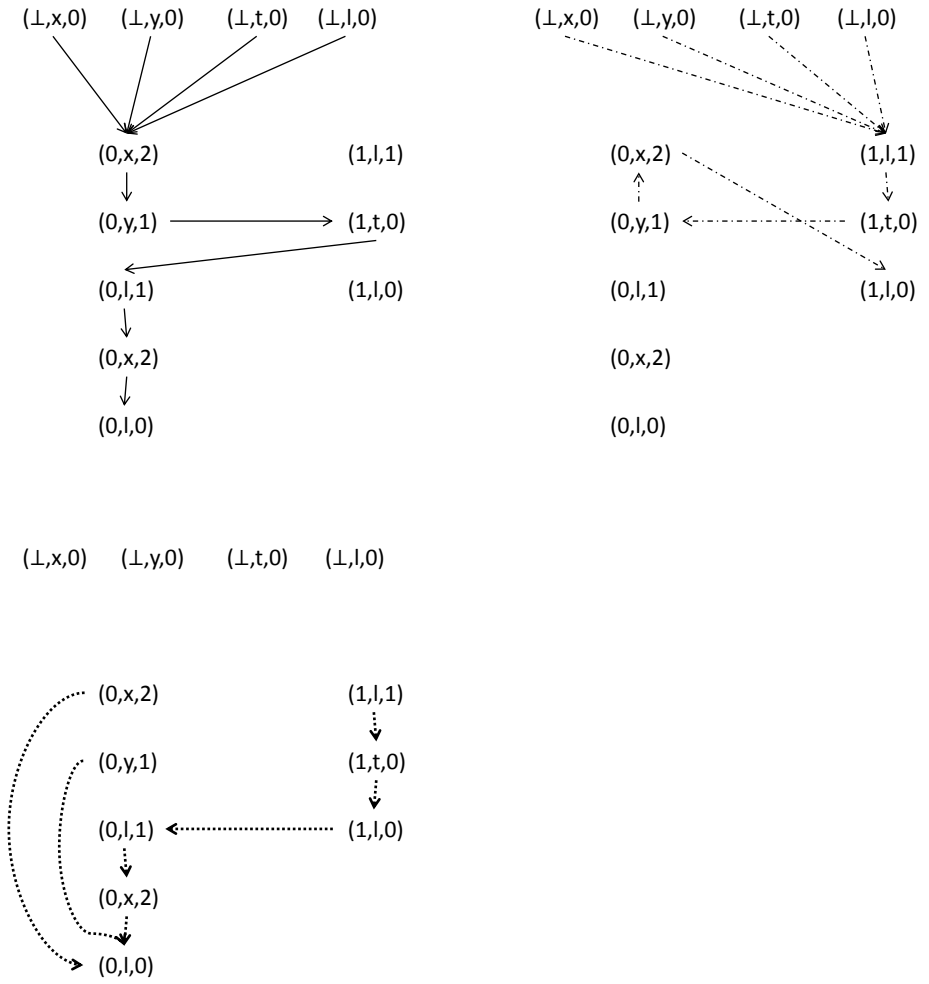


Abbildung 4.2: Oben links ist die Ansicht von Prozessor 0 zu sehen. Wie beschrieben, synchronisiert die Ansicht von Prozessor 0 und 1 vor der Ausführung von l.acquire durch Prozessor 0. Oben rechts ist die Ansicht von Prozessor 1 gegeben. Nach Ausführen des Threads auf Prozessor 1 wurde dieser nicht weiter über Schreiboperationen von Prozessor 0 informiert, sodass Prozessor 1 die letzte Schreiboperation x:=2 nicht sieht. Unten ist die Lockordnung gegeben. Entsprechend der Semantik der Locks ergeben sich die dort gezeigten Relationen. Z.B ist $(0, l, 0)$ ein Einweg-Barriere, sodass alle vorherigen Operationen nicht nach $(0, l, 0)$ gesehen werden können.

Sei $mem = (wh, V, L, ls, i) \in MEM$. Die Menge $Var(mem) \subset Var$ bezeichnet die Menge der im Speicher vorkommenden Variablen, d.h. es existiert $x \in Var(mem)$ genau dann, wenn

$$\exists j \in dom(wh)(var(wh(j)) = x)$$

Ein Prozessor p hat eine Schreiboperation $j \in dom(wh)$ eines Speichers (wh, V, L, ls, i) gesehen (bezeichnet mit $seen(p, j)$), wenn j eine initiale Schreiboperation ist oder j in Relation zu ein anderen Schreiboperation in $V(p)$ steht, d.h.

$$proc(wh(j)) = \perp \vee \exists k \quad (k\ V(p)\ j \vee j\ V(p)\ k)$$

Dies bedeutet nicht, dass p die Schreiboperation j auch wahrgenommen hat. Im zweiten Fall des beschriebenen Beispiels zu Abbildung 4.1 hat Prozessor 1 nur $(x, 3)$ wahrgenommen. Die anderen beiden Operationen hat er lediglich gesehen.

Der aktuelle Wert einer Variable x aus Sicht von p ist gegeben durch den Wert der zuletzt gesehenen Schreiboperation j in x, d.h.

$$var(wh(j)) = x \wedge seen(p, j) \wedge \neg \exists k \quad (var(wh(k)) = x \wedge seen(p, k) \wedge j\ V(p)\ k)$$

Falls j den aktuellen Wert für x aus Sicht von p angibt, wird dies mit $current(p, j, x)$ bezeichnet.

Die Ausführung von Kommandos eines Programms haben lokale als auch globale Auswirkungen. Lokale Auswirkungen betreffen die Ausführung auf Threadebene, wohingegen globale Auswirkungen die Ausführung auf Systemebene betreffen. Daher ist die Semantik auf beiden Ebenen (Thread und System) definiert.

Während der Ausführung eines Programms entscheidet das System zwischen der Ausführung der einzelnen Kommandos, inwieweit die Write Buffer der Prozessoren geleert werden. Das bedeutet, dass zum Zeitpunkt der Ausführung eines Kommandos auf Threadebene feststeht, was ein Prozessor und demnach der Thread für Werte für jede Variable aktuell sieht.

4.3.2 Threadzustand und lokale Auswirkungen

Ein Thread hat nur eine lokale Sicht auf den Speicher, d.h. er hat die selbe Sicht wie Prozessor p, auf dem er ausgeführt wird. Der Thread sieht daher auch nur einen Teil des Speicher (wh, V, L, ls, i), nämlich den Teil, der für die Ausführung des Threads relevant ist. Das ist neben der Historie der Schreiboperationen wh, die Ansicht des Prozessors auf diese Menge $V(p)$, den Lockzustand ls und den Index i der letzten getätigten Schreiboperation.

Definition 4.6 (Lokaler Speicher)**.** Ein lokaler Speicher ist ein 4-Tupel $(wh, V(p), ls, i)$, wobei wh, ls und i wie in 4.4 definiert sind und $V(p) \in SO(wh)$ eine Ansicht auf den Speicher darstellt.

Die Menge der lokalen Speicher wird mit $LOCALMEM$ bezeichnet. Elemente von $LOCALMEM$ werden u.a. mit lm_1, lm_2 etc. bezeichnet.

Im Folgenden werden die lokalen, also durch einen Thread hervorgerufenen, Auswirkungen auf den lokalen Speicher beschrieben.

Das Ausführen einer Schreiboperation bewirkt lokal, dass p in x den Wert n schreibt und diese Schreiboperation (p, x, n) die aktuellste gesehene Schreiboperation in x aus Sicht von p ist. Zudem wird die Schreiboperation entsprechend der Programmordnung von p eingeordnet, d.h. (p, x, n) ist größer als alle anderen von p getätigten Schreiboperationen. Die entsprechende Funktion ist gegeben durch

$$
\begin{aligned}
&\mathit{localWrite} : \mathit{Proc} \times \mathit{Var} \times \mathit{Val} \times \mathit{LOCALMEM} \longrightarrow \mathit{LOCALMEM} \\
&(p, x, n, (wh, V(p), ls, i)) \mapsto \\
&\quad (wh \cup \{(i+1, (p, x, n))\}, \\
&\qquad V(p) \quad \cup \{(j, i+1) \mid \mathit{seen}(V(p), j)\}, \\
&\quad ls, \\
&\quad i+1)
\end{aligned}
$$

Ein Lock l ist frei ($\mathit{lockFree}(l, ls)$) genau dann, wenn kein anderer Prozessor sich den Lock angeeignet hat, d.h. $ls(l) = \perp$.

Ein Lock l ist von einem Prozessor p angeeignet ($\mathit{lockAcquired}(p, l, ls)$) genau dann, wenn derselbe Prozessor sich den Lock zuvor angeeignet hat, d.h. $ls(l) = p$.

Das Ausführen einer $l.acquire$ Anweisung durch einen Thread auf einem Prozessor p ist möglich, wenn l frei ist, d.h. $lockFree(l, ls)$, und bewirkt lokal, dass p in l den Wert 1 schreibt.

Ein Prozessor p hingegen kann eine $l.release$ Anweisung ausführen, wenn l von p angeeignet wurde oder l frei ist, d.h.

$$\mathit{lockAcquired}(p, l, ls) \vee \mathit{lockFree}(l, ls)$$

Der letzte Fall hat keinen Effekt, wohingegen der erste Fall bewirkt, dass p in l den Wert 0 schreibt.

Da in beiden Fällen (Acquire und Release) nur ein Schreibzugriff in l erfolgt, lassen sich die lokale Auswirkungen mit Hilfe von $localWrite$ ausdrücken.

$$localAcquire : Proc \times Lock \times LOCALMEM \longrightarrow LOCALMEM$$
$$(p, l, (wh, V(p), ls, i)) \mapsto localWrite(p, l, 1, (wh, V(p), ls, i))$$

$$localRelease : Proc \times Lock \times LOCALMEM \longrightarrow LOCALMEM$$
$$(p, l, (wh, V(p), ls, i)) \mapsto localWrite(p, l, 0, (wh, V(p), ls, i))$$

Das Auswerten eines Ausdrucks geschieht auf Basis der aktuell gesehenen Werte der Variablen.

Sei $X \subset Var$ die Menge der in einem Ausdruck $expr \in Expr$ vorkommenden Variablen und $(wh, V(p), ls, i)$ ein lokaler Speicher. Sei n_x der aktuelle Wert für x aus Sicht von p, d.h. für das j mit
$current(p, j, x)$ ist $val(wh(j)) = n_x$. Die Auswertung des Ausdrucks $expr$ geschieht dann mit Hilfe der üblichen Interpretation der Operatoren $op \in Op$ und der Werte n_x für die Variablen $x \in X$.

Wenn der Ausdruck auf Basis des lokalen Speichers zu n evaluiert, so wird dies mit $eval(expr, lm) = n$ abgekürzt. Für globale Speicher mem wird zusätzlich der Prozessor p angegeben, d.h. $eval_p(expr, mem) = n$.

Der Zustand eines Threads besteht aus dem noch auszuführenden Programm sowie dem lokalen Speicher bzw. der lokalen Ansicht des Threads auf den Speicher.

Definition 4.7 (Threadzustand)**.** Ein *Threadzustand* ist ein Tupel $\langle C, (wh, V(p), ls, i)\rangle$, wobei $C \in Com$ und $(wh, V(p), ls, i)$ der lokale Speicher und damit die Sicht des Threads auf den Speicher ist.

Die operationale Semantik für Threadzustände ist in Abbildung 4.3 zu sehen. Hierbei geht ein Threadzustand in einen anderen über. Zudem informiert ein Thread beim Übergang von einem Zustand zum nächsten das System über den Typ der Speicheroperation und den damit verbundenen Schreiboperationen. Dies wird als Speicherereignis bezeichnet. Es existieren vier Typen von Speicherereignissen für die Anweisungen Acquire, Release und eine Zuweisung, sowie für den Fall, dass keine Änderung des Speichers auf globaler Ebene notwendig ist.

Definition 4.8 (Speicherereignis)**.** Ein *Speicherereignis* ist ein Element der Menge $\{acq, rel, w, none\}$.

$$\text{Seq1}\ \frac{\langle C_1, lm\rangle \longrightarrow_e \langle\langle\rangle, lm'\rangle}{\langle C_1; C_2, lm\rangle \longrightarrow_e \langle C_2, lm'\rangle}$$

$$\text{Seq2}\ \frac{\langle C_1, lm\rangle \longrightarrow_e \langle C_1', lm'\rangle}{\langle C_1; C_2, lm\rangle \longrightarrow_e \langle C_1'; C_2, lm'\rangle}$$

$$\text{Skip}\ \frac{}{\langle \mathbf{skip}, lm\rangle \longrightarrow_{none} \langle\langle\rangle, lm\rangle}$$

$$\text{If-True}\ \frac{eval(B, lm) = True}{\langle \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, lm\rangle \longrightarrow_{none} \langle C_1, lm\rangle}$$

$$\text{If-False}\ \frac{eval(B, lm) = False}{\langle \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, lm\rangle \longrightarrow_{none} \langle C_2, lm\rangle}$$

$$\text{While-True}\ \frac{eval(B, lm) = True}{\langle \mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od}, lm\rangle \longrightarrow_{none} \langle C; \mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od}, lm\rangle}$$

$$\text{While-False}\ \frac{eval(B, lm) = False}{\langle \mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od}, lm\rangle \longrightarrow_{none} \langle\langle\rangle, lm\rangle}$$

$$\text{Assign}\ \frac{\begin{array}{c} eval(expr, lm) = n \\ lm' = localWrite(p, var, n, lm) \end{array}}{\langle var := exp, lm\rangle \longrightarrow_w \langle\langle\rangle, lm'\rangle}$$

$$\text{Acq}\ \frac{\begin{array}{c} lockFree(l, ls) \\ lm' = localAcquire(p, l, lm) \end{array}}{\langle l.acquire, (wh, V(p), ls, i)\rangle \longrightarrow_{acq} \langle\langle\rangle, lm\rangle}$$

$$\text{Rel1}\ \frac{\begin{array}{c} lockAcquired(p, l, ls) \\ lm' = localRelease(p, l, lm) \end{array}}{\langle l.release, lm\rangle \longrightarrow_{rel} \langle\langle\rangle, lm'\rangle}$$

$$\text{Rel2}\ \frac{lockFree(l, ls)}{\langle l.release, lm\rangle \longrightarrow_{none} \langle\langle\rangle, lm\rangle}$$

Abbildung 4.3: Operationale Semantik für Threads

4.3.3 Systemzustand und globale Auswirkungen

Das System hat eine globale Sicht auf den Speicher, d.h. es sieht alle Komponenten, wie sie in 4.4 definiert sind.

Wie bereits erwähnt, entscheidet das System vor der Ausführung einer Anweisung durch einen Prozessor, welche Schreiboperationen die Prozessoren 0 und 1 jeweils neu wahrnehmen. Da die Laufzeitumgebung die Eigenschaft *Read Own Writes Early* hat, kann ein Prozessor p nur neue Schreiboperation in eine Variable x von $1-p$ wahrnehmen, wenn der eigene Write Buffer keine Schreiboperation in x enthält. Bestimmt man also, dass ein Prozessor eine Schreiboperation wahrnimmt, so muss vorher der Write Buffer des Prozessor entsprechend geleert werden.

Hierzu werden Funktionen $\varepsilon^p_{n,x,t}$ [1] verwendet, die diese Veränderung des Speichers, insbesondere der Ansichten, modellieren. Die Funktion $\varepsilon^p_{n,x,t}$ verändert einen Speicher mem dahingehend, dass p die n-te ungesehene Schreiboperation in x mit Wert t von $1-p$ als aktuellste Schreiboperation in x wahrnimmt, d.h. sie ist größer (nach $V(p)$) als alle bisher gesehenen Operationen in x. Zudem verändert die Funktion die Sicht von $1-p$, sodass dieser alle Schreiboperation in x von p vor dieser Schreiboperation sieht. Die Anwendung einer Funktion $\varepsilon^p_{n,x,t}$ entfernt also alle Schreiboperationen in x aus dem Write Buffer von p und anschließend die ersten n Schreiboperationen in x aus dem Write Buffer von $1-p$.

Sei j die n-te ungesehene Schreiboperation in x mit Wert t von $1-p$. Dann ist die zugehörige ε-Funktion definiert durch

$$
\begin{aligned}
&\varepsilon^p_{n,x,t} : MEM \longrightarrow MEM \\
&(wh, V, L, ls, i) \mapsto \\
&(wh, \\
&\{p \mapsto \begin{array}{ll} V(p) & \cup\{(k,j) \mid seen(p,k)\} \\ & \cup\{(k,j) \mid var(wh(k)) = x, k\ V(1-p)\ j\} \end{array}\ , \\
&1-p \mapsto \begin{array}{l} (V(1-p) \cup \{(k,j) \mid var(wh(k)) = x, proc(wh(k)) = p\} \\ \cup\{(k,m) \mid var(wh(k)) = x, var(wh(m)) = x, k\ V(p)\ m\})^+\} \end{array}\ , \\
&ls, i)
\end{aligned}
$$

In Abbildung 4.4 ist ein Beispiel einer Anwendung einer ε-Funktion zu sehen. Der linke Speicher ist der Ausgangsspeicher auf den die Funktion $\varepsilon^0_{2,x,4}$ angewandt wird. In diesem Speicher haben die Prozessoren mehrere Schreiboperationen in x getätigt, aber noch nicht eine Schreiboperation des anderen gesehen. Das Resultat der Anwendung ist der rechte Speicher. Die

[1] griechisch $\varepsilon\pi o\pi\tau\varepsilon u\omega$, wahrnehmen

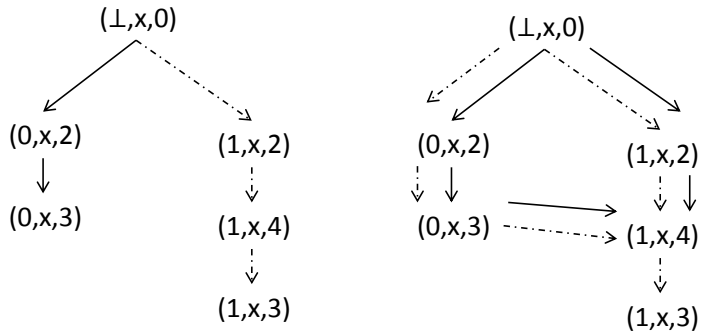


Abbildung 4.4:

nicht gestrichelten Pfeile stellen die Ordnung von Prozessor 0 dar, wohingegen die gestrichelten Pfeile die Ansicht von Prozessor 1 repräsentieren.

Das System wendet vor der Ausführung einer Anweisung eine Kette dieser ε-Funktionen an, wobei hierbei die Lockordnung L berücksichtigt wird bzw. die entstandenen Bedingungen auf den Write Buffern.

Definition 4.9 (ε-Kette)**.** Eine ε-Kette ist die Komposition mehrerer ε-Funktionen oder die Identität auf MEM.

Durch die Lockordnung L kann es vorkommen, dass das Sehen einer Schreiboperation $(1-p, x, t)$ durch p das Sehen oder Wahrnehmen weiterer Schreiboperationen nach sich zieht. Als Beispiel sei hier erneut das Programm aus Beispiel 3.8 gegeben.

	Proc 0	Proc 1
1	l.acquire	print x
2	x:= 1	print y
3	l.release	
4	z.acquire	
5	y := 1	
6	z.release	

Hierbei kann die Schreiboperation $y := 1$ durch 0 erst den Write Buffer verlassen, wenn $x := 1$ diesen ebenfalls verlässt oder bereits verließ. Das bedeutet, dass die Anwendung von $\varepsilon^1_{1,y,1}$ ebenfalls der Anwendung von $\varepsilon^1_{1,x,1}$ bedarf.

Das Ausführen einer Acquire-Anweisung erzwingt ebenfalls das Entfernen bestimmter Schreiboperationen aus dem Write Buffer, sofern die letzte Release-Anweisung auf dem gleichen Lock l vom anderen Prozessor ausgeführt wurde. Ein Beispiel hierzu ist Beispiel 3.7.

Sei $mem = (wh, V, L, ls, i) \in MEM$ und $j \in dom(wh)$. Die mit einer Schreiboperation j einhergehenden Schreiboperationen wird in der folgende Menge zusammengefasst.

$$K_{j,mem} = \{k \; L \; j \mid k \in dom(wh), var(wh(k)) \in Var\}$$

Eine ε-Kette $\varepsilon^{p_n}_{m_n,x_n,t_n} ... \varepsilon^{p_1}_{m_1,x_1,t_1}$ ist gültig, wenn die Lockordnung L respektiert wird, d.h. dass wie im obigen Beispiel $\varepsilon^1_{1,y,1}$ nur in der Kette auftauchen kann, wenn auch $\varepsilon^1_{1,x,1}$ auftaucht und $\varepsilon^1_{1,y,1}$ nach $\varepsilon^1_{1,x,1}$. Weiterhin hängt die Gültigkeit einer Kette davon ab, ob als nächstes eine Acquire-Anweisung ausgeführt wird. Ist dies der Fall, muss das System eine Synchronisation der Ansichten vornehmen, d.h. alle Operationen vor der letzten zugehörigen Release-Anweisung müssen vom Prozessor gesehen werden. Z.B. würde eine l.acquire-Anweisung von Prozessor 1 nach print y im obigen Beispiel dazu führen, dass die ε-Kette die Funktion $\varepsilon^1_{1,x,1}$ enthält.

Definition 4.10 (Gültige ε-Kette)**.** Sei $mem = (wh, V, L, ls, i)$ und

$$\varepsilon^{p_n}_{m_n,x_n,t_n} ... \varepsilon^{p_1}_{m_1,x_1,t_1}$$

eine ε-Kette.

Im Fall, dass die Kette nicht die Identität ist, seien op_i die Schreiboperationen für jede $\varepsilon^{p_i}_{m_i,x_i,t_i}$, die hierdurch als gesehen gesetzt wird (d.h. die m_i-te Schreiboperation in x_i mit Wert t_i von $1 - p_i$).

Die Kette ist *gültig* bezüglich mem und zwei Kommandos $C_1, C_2 \in Com_{MTL-L}$, wenn die Kette die Identität ist oder für $mem' = \varepsilon^{p_n}_{m_n,x_n,t_n} \ldots \varepsilon^{p_1}_{m_1,x_1,t_1}(mem)$ gilt,

- $\forall op_i \forall j \in K_{op_i,mem} \quad (seen(p, j))$
- wenn $C_1 = l_1.acquire$, die letzte $l_1.release$ Anweisung von Prozessor 1 ausgeführt wurde und j_1 der Index der zugehörigen Schreiboperation ist, dann gilt $\forall j \in K_{j_1,mem} \quad (seen(0, j))$,
- wenn $C_2 = l_2.acquire$, die letzte $l_2.release$ Anweisung von Prozessor 0 ausgeführt wurde und j_2 der Index der zugehörigen Schreiboperation ist, dann gilt $\forall j \in K_{j_2,mem} \quad (seen(1, j))$
- die Reihenfolge der ε-Funktionen entspricht der Lockordnung, d.h. für op_i und op_j mit $i < j$ gilt, dass $\neg(op_j \; L \; op_i)$

Eine gültige Kette wird mit

$$validChain(mem, C_1, C_2, \varepsilon^{p_n}_{m_n,x_n,t_n} \dots \varepsilon^{p_1}_{m_1,x_1,t_1})$$

bezeichnet.

Es werden nun die Auswirkungen der Acquire-, Release- und Zuweisungsoperationen auf globaler Ebene beschrieben. Diese Auswirkungen geschehen nach der Ausführung der Anweisung durch die Threads. Daher zeigt im Fall, dass der Schreibhistorie eine Schreiboperation hinzugefügt wurde, der Index i des Speichers auf diese Schreiboperation.

Eine Zuweisungsoperation tätigt eine Schreiboperation (p, x, n), die den globalen Effekt hat, dass ausschließlich die Lockordnung L aktualisiert wird, indem (p, x, n) nach jeder Acquire-Operation des gleichen Prozessor eingeordnet wird. Dies modelliert die Eigenschaft des Speichermodells, dass Anweisungen nicht vor Acquire-Anweisungen des selben Prozessors gesehen werden können, die nach der Acquire-Anweisung getätigt wurden. Es ist Prozessor $1 - p$ also nicht möglich (p, x, n) vor einer Operation $(p, l, 1)$ mit $l \in Lock$ zu sehen. Hierzu sei

$$\begin{aligned}
&globalWrite : Proc \times MEM \longrightarrow MEM \\
&(p, (wh, V, L, ls, i)) \mapsto \\
&(wh, V, (L \cup \{(k, i) \mid k \in dom(wh), var(wh(k)) \in Lock, AcquireOp(k)\})^+, ls, i)
\end{aligned}$$

Eine Acquire-Operation bewirkt, dass der Lock l als angeeignet markiert wird, sodass fortan kein Thread den Lock aneignen kann. Weiterhin aktualisiert die untenstehende Funktion $globalAcquire$ die Lockordnung L, indem die Acquire-Operation größer als alle Lockoperationen von p gesetzt wird. Dies modelliert, dass Synchronisationkommandos vom anderen Prozessor in der gleichen Reihenfolge gesehen werden müssen, wie sie getätigt wurden.

Sei j der Index der letzten $l.release$-Operation (im Fall, dass keine ausgeführt wurde, ist dies der Index der initialen Schreiboperation in l). Dann ist $globalAcquire$ definiert durch

$$\begin{aligned}
&globalAcquire : Proc \times Lock \times MEM \longrightarrow MEM \\
&(p, j, (wh, V, L, ls, i)) \mapsto \\
&(wh, V, \\
&(L \cup \{(j, i)\} \cup \{(k, i) \mid k \in dom(wh), var(wh(k)) \in Lock, proc(wh(k)) = p\})^+, \\
&ls, i)
\end{aligned}$$

Eine Release-Operation hat global den Effekt, dass die Lockordnung L dahingehend aktualisiert wird, dass jede Schreiboperation des gleichen Prozessors vor der Release-Operation eingeordnet wird. Dadurch wird modelliert,

dass ein anderer Prozessor Anweisungen, die vor einer Release-Anweisung getätigt wurden, nicht nach dieser sehen kann. Die entsprechende Funktion lautet

$$
\begin{aligned}
&globalRelease : Proc \times Lock \times MEM \longrightarrow MEM \\
&(p, l, (wh, V, L, ls, i)) \mapsto \\
&(wh, V, (L \cup \{(k, i) \mid k \in dom(wh), proc(wh(k)) = p\})^+, ls[l \mapsto \bot], i)
\end{aligned}
$$

Abhängig vom Speicherereignis der ausführenden Threads werden die obigen Funktionen auf den Speicher angewandt. Dies stellt die folgende Funktion dar.

$$
\begin{aligned}
&globalEffect : Proc \times \{acq, rel, w, none\} \times MEM \longrightarrow MEM \\
&(p, e, J, (wh, V, L, ls, i)) \mapsto \\
&\begin{cases}
globalAcquire(p, var(wh(i)), (wh, V, L, ls, i)) & \text{falls } e = acq \\
globalRelease(p, (wh, V, L, ls, i)) & \text{falls } e = rel \\
globalWrite(p, (wh, V, L, ls, i)) & \text{falls } e = w \\
(wh, V, L, ls, i) & \text{sonst}
\end{cases}
\end{aligned}
$$

Das System ist in der Lage zwei Threads parallel auszuführen. Hierbei gilt jedoch die Einschränkungen, dass beide Threads nicht gleichzeitig eine Acquire-Operation für den gleichen Lock l ausführen.

Bei der parallelen Ausführung werden für jeden Thread die lokalen und globalen Auswirkungen ermittelt und anschließend zusammengeführt. Dabei kann der Fall auftreten, dass beide Threads eine Schreiboperation tätigten und so eine einfache Vereinigung der Funktionen wh und wh' nicht möglich ist, da beide den gleichen Index auf unterschiedliche Schreiboperationen abbilden. Daher muss ein Index durch einen freien, also um Eins größer, ersetzt werden, falls $i = i'$.

Das Zusammenführen der lokalen und globalen Auswirkungen besteht aus vier Teilen:

1. der Vereinigung der Schreibhistorien
2. der Vereinigung und anschließenden Bildung der transitiven Hülle der Ansichten
3. der Vereinigung und anschließenden Bildung der transitiven Hülle der Lockordnung
4. der Zusammenführung der Lockzustände

Das Zusammenführen der Lockzustände ist durch folgende Funktion gegeben. Sie nutzt aus, dass die Lockzustände ls' und ls'' sich nur jeweils an einer Stelle von ls unterscheiden. Weiterhin ist es nicht erlaubt, dass beide Prozessoren den gleichen Lock zur selben Zeit aneignen. Daher sind die Stellen an denen sich ls' und ls'' von ls unterscheiden nicht die selben.

Sei $LS = Lock \rightharpoonup (\{0,1\} \times Proc)$.

$$composeLockStates : LS \times LS \times LS \longrightarrow LS$$
$$(ls, ls', ls'')(l) \mapsto \begin{cases} ls'(l) & \text{falls } ls'(l) \neq ls(l) \\ ls''(l) & \text{falls } ls''(l) \neq ls(l) \\ ls(l) & \text{sonst} \end{cases}$$

Tätigte höchstens einer der beiden Threads eine Schreiboperation, so lassen sich die resultierenden Speicher ohne Weiteres zusammenführen.

Die Funktion *compose* führt zwei Speicher (wh', V', L', ls', i') und $(wh'', V'', L'', ls'', i'')$, die aus (wh, V, L, ls, i) resultieren, zusammen. Hierbei wird angenommen, dass $i' = i$ oder $i'' = i$.

Sei

$$\begin{aligned} ls''' &= composeLockStates(ls, ls', ls'') \\ V'''(p) &= (V'(p) \cup V''(p))^+ \\ V'''(1-p) &= (V'(1-p) \cup V''(1-p))^+ \\ L''' &= (L' \cup L'')^+ \end{aligned}$$

Die Funktion lautet dann:

$$\begin{aligned} &compose : MEM \times MEM \times MEM \longrightarrow MEM \\ &((wh, V, L, ls, i), (wh', V', L', ls', i'), (wh'', V'', L'', ls'', i'')) \mapsto \\ &(wh' \cup wh'', V''', L''', ls''', \max\{i', i''\}) \end{aligned}$$

Für den Fall, dass beide Threads eine Schreiboperation tätigten, ersetzt die Funktion *substituteIndex* einen Index j durch einen Index k in einem Speicher.

$$\begin{aligned} &substituteIndex : \mathcal{J} \times \mathcal{J} \times MEM \longrightarrow MEM \\ &(j, k, (wh, V, L, ls, i)) \mapsto substituteIndex(j, k, (wh, V, L, ls, i)) \end{aligned}$$

Nun lässt sich die eigentliche Zusammenführung von Speichern definieren. Die Funktion *composeMemories* führt zwei Speicher $mem' = (wh', V', L', ls', i')$ und mem'', die aus mem resultieren, zusammen.

$$composeMemories : MEM \times MEM \times MEM \longrightarrow MEM$$
$$(mem, mem', mem'') \mapsto$$
$$\begin{cases} compose(mem, substituteIndex(i', i'+1, mem'), mem'') & \text{falls } i' = i'' \neq i \\ compose(mem, mem', mem'') & \text{sonst} \end{cases}$$

Wenn das System ein Programm bestehend aus mehreren Threads ausführt, so wählt es am Anfang eine Verteilung, die einen Thread einem Prozessor zuordnet. Der Thread wird fortan nur von diesem Prozessor ausgeführt.

Definition 4.11 (Verteilung)**.** Eine *Verteilung* für einen Threadpool der Länge n ist eine Funktion $\pi : \{1, \ldots, n\} \longrightarrow Proc$.

Als Scheduler wird wie in Kapitel 2 ein possibilistischer Scheduler verwendet, sodass der Systemzustand aus einem Threadpool, einer Verteilung und einem Speicherzustand besteht.

Definition 4.12 (Systemzustand)**.** Ein *Systemzustand* ist ein Tripel $\langle \overrightarrow{C}, \pi, mem \rangle$, wobei $\overrightarrow{C} \in \overrightarrow{Com_{MTL-L}}$, π eine Verteilung für $\overrightarrow{C}$ und $mem \in MEM$ der Speicher ist.

Die operationelle Semantik für Systemzustände ist in Abbildung 4.5 zu sehen. Die Semantik hat zwei Regeln **Non-Par** und **Par**. Die erste Regel modelliert die Ausführung des Programms durch nur einen Prozessor, wohingegen die zweite Regel die parallele Ausführung des Programms modelliert. Die zusätzlichen Bedingungen für die parallele Ausführung sind, dass beide Threads unterschiedlichen Prozessoren zugewiesen wurden und beide Threads in diesem Schritt nicht den selben Lock aneignen. Vor der Ausführung einer Anweisung durch einen Thread führt das System eine ε-Kette aus. Im Anschluss werden die globalen Auswirkungen ermittelt. Im Fall der parallelen Ausführung werden zusätzlich die einzelnen Auswirkungen der Threads als auch der damit zusammenhängenden globalen Auswirkungen zusammengeführt.

Ein System startet mit einem Programm $\overrightarrow{C}$ und einem zugehörigen initialen Speicher (wh, V, L, ls, i). Die einzelnen Threads werden auf die beiden Prozessoren aufgeteilt. Das Programm wurde ausgeführt, wenn jeder Thread im terminierenden Zustand ist, d.h. nur noch das leere Programm auszuführen ist. Im Gegensatz zur Semantik für MTL werden terminierte Threads nicht aus dem Threadpool entnommen.

Definition 4.13 (Gültiger Speicher)**.** Ein Speicher $mem \in MEM$ wird als *gültig* ($valid(mem)$) bezeichnet, wenn ein initialer Speicher mem' und ein Programm $\overrightarrow{C}$ existiert, sodass mem der resultierende Speicher der Ausführung von $\overrightarrow{C}$ mit Speicher mem' ist.

$$\text{Non-Par}\ \frac{\begin{array}{c} validChain(mem, C_{j,0}, \epsilon, \varepsilon^{p_n}_{m_n,x_n,t_n} \dots \varepsilon^{p_1}_{m_1,x_1,t_1}) \\ (wh', V', L', ls', i') = \varepsilon^{p_n}_{m_n,x_n,t_n} \dots \varepsilon^{p_1}_{m_1,x_1,t_1}(mem) \\ \langle C_j, (wh', V(\pi(j))', ls', i')\rangle \longrightarrow_e \langle C'_j, (wh'', V(\pi(j))'', ls'', i'')\rangle \\ mem^{III} = globalEffect(\pi(j), e, (wh'', V[\pi(j) \mapsto V''(\pi(j))], L, ls, i'')) \end{array}}{\langle C_1 \dots C_j \dots C_n, \pi, mem\rangle \rightarrowtriangle \langle C_1 \dots C'_j \dots C_n, \pi, mem^{III}\rangle}$$

$$\text{Par}\ \frac{\begin{array}{c} \pi(j) \neq \pi(k) \qquad C_{j,0} \neq l.acq \vee C_{k,0} \neq l.acq \\ validChain(mem, C_{j,0}, C_{k,0}, \varepsilon^{p_n}_{m_n,x_n,t_n} \dots \varepsilon^{p_1}_{m_1,x_1,t_1}) \\ (wh^I, V^I, L^I, ls^I, i^I) = \varepsilon^{p_n}_{m_n,x_n,t_n} \dots \varepsilon^{p_1}_{m_1,x_1,t_1}(mem) \\ \langle C_j, (wh^I, V(\pi(j))^I, ls^I, i^I)\rangle \longrightarrow_e \langle C_j, (wh^{II}, V(\pi(j))^{II}, ls^{II}, i^{II})\rangle \\ mem^{III} = globalEffect(\pi(j), e, (wh^{II}, V[\pi(j) \mapsto V^{II}(\pi(j))], L, ls, i^{II})) \\ \langle C_k, (wh^I, V(\pi(k))^I, ls^I, i^I)\rangle \longrightarrow_f \langle C'_k, (wh^{IV}, V(\pi(k))^{IV}, ls^{IV}, i^{IV})\rangle \\ mem^V = globalEffect(\pi(k), f, (wh^{IV}, V[p \mapsto V^{IV}(\pi(k))], L, ls, i^{IV})) \\ mem^{VI} = composeMemories(mem, mem^{III}, mem^V) \end{array}}{\langle C_1 \dots C_j \dots C_k \dots C_n, \pi, mem\rangle \rightarrowtriangle \langle C_1 \dots C'_j \dots C'_k \dots C_n, \pi, mem^{VI}\rangle}$$

Abbildung 4.5: Operationale Semantik für Systemzustände

Für das nachstehende Lemma ist es relevant, dass für jede Variable exakt eine maximale gesehene Schreiboperation aus Sicht eines Prozessors existiert, die den Wert der Variable angibt. Für einen gültigen Speicher ist dies gegeben, da die einzelnen Auswirkungen der Semantik stets Operationen größer als zuvor gesehene anordnen.

Satz 4.14. Für die ε-Funktionen gelten folgende Eigenschaften. Sei $mem \in MEM$ und gültig.

1. Für $mem' = \varepsilon^p_{n,x,t}(mem)$ gilt:
   (a) $1 - p$ hat alle Schreiboperationen in x von p gesehen
   (b) Wenn m die Anzahl der ungesehen Schreiboperation in x ist aus Sicht von p in mem, so ist die Anzahl der ungesehen Schreiboperation in x in mem' aus Sicht von p $m - n$,

(c) die ungesehenen Schreiboperation in $y \neq x$ sind in mem' und mem die selben

2. In $\varepsilon^p_{n,x,t}(mem)$ ist der Wert für x aus Sicht von p t und alle restlichen Werte der Variablen sind die selben wie in mem.

3. $\varepsilon^{1-p}_{m,x,k}$ ist nicht anwendbar auf $\varepsilon^p_{n,x,t}(mem)$

Beweis. 1. Sei $mem \in MEM$ und gültig, $mem' = \varepsilon^p_{n,x,t}(mem)$ und j die n-te ungesehene Schreiboperation in x aus Sicht von p.

(a) Es gilt in mem'

$$\forall k \in dom(wh) \quad (var(wh(k)) = x \wedge proc(wh(k)) = p \implies k\, V(1-p)\, j)$$

Das bedeutet, dass $1-p$ alle Schreiboperationen von p in x gesehen hat.

(b) Sei m die Anzahl der ungesehenen Schreiboperationen in x aus Sicht von p in mem. Da $\varepsilon^p_{n,x,t}$ anwendbar ist auf mem, gilt $m \geq n$. Dann existieren $m-n$ Schreiboperationen die größer (nach $V(1-p)$) sind als j in mem.

Die Funktion $\varepsilon^p_{n,x,t}$ setzt gesehene Operationen von p, die Operation j und Schreiboperationen in x von $1-p$ kleiner $(V(1-p))$ als j in Beziehung. Daher stehen die $m-n$ Operationen in x, die größer $(V(1-p))$ als j sind in mem, weiterhin in keiner Relation aus Sicht von p in mem'. Somit sind sie weiter ungesehen aus Sicht von p.

(c) Sei $y \neq x$. Zuerst betrachten wir die ungesehen Schreiboperationen aus Sicht von p. Sei k ein ungesehene Schreiboperation in y aus Sicht von p. Nach Anwendung von $\varepsilon^p_{n,x,t}$ wurde k in keine Relation zu einer anderen Schreiboperation aus Sicht von p gesetzt. Damit ist k weiter ungesehen in mem'.

Wir betrachten nun die ungesehenen Schreiboperation aus Sicht von $1-p$. Sei k eine solche Schreiboperation. Nach der Definition von $\varepsilon^p_{n,x,t}$ werden nur Operationen in x von p in Relation gesetzt. Damit bleibt k unberührt und weiter ungesehen.

2. Sei $mem \in MEM$ und $mem' = \varepsilon^p_{n,x,t}(mem)$. Zudem sei $y \in Var(mem)$.

Es werden zuerst die Werte der Variablen aus Sicht von p in mem' betrachtet. Sei hierzu zuerst $y = x$. Da in mem' j größer (nach $V(p)$)

als alle Schreiboperationen von p in x ist, ist nach Definition des aktuell gesehenen Werts für eine Variable der Wert von j, also t, der aktuelle für x aus Sicht von p.

Sei nun $y \neq x$. Sei l die aktuellste Schreiboperation in y in mem. In mem' gilt $l\ V(p)\ j$. Jedoch gilt weiterhin, dass l die aktuellste Schreiboperation in y ist. Somit ist der Wert für y aus Sicht von p in mem und mem' der selbe.

Als nächstes betrachten wir die Werte der Variablen aus Sicht von $1-p$. Hierzu sei $y = x$. Sei l die aktuellste Schreiboperation in y in mem aus Sicht von $1-p$. Entweder ist $l = j$ oder $j\ V(1-p)\ l$. Daher gilt in beiden Fällen in mem', dass alle Schreiboperation in x von p kleiner als l sind und sich damit der Wert für x aus Sicht von $1-p$ nicht geändert hat.

Sei zuletzt $y \neq x$. Sei l die aktuellste Schreiboperation in y in mem aus Sicht von $1-p$. Da durch $\varepsilon^{p}_{n,x,t}$ nur neue Relation zwischen Schreiboperationen in x zu mem hinzugefügt werden, ist l weiterhin die aktuellste Schreiboperation in y in mem' aus Sicht von $1-p$.

3. Nach 1 (a) gilt, dass $1-p$ alle Schreiboperation in x von p in $mem' = \varepsilon^{p}_{n,x,t}(mem)$ gesehen hat. Somit ist kein $\varepsilon^{1-p}_{m,x,k}$ auf mem' anwendbar.

□

4.4 Eigenschaften und Beispielprogramme

Es werden nun Eigenschaften der oben definierten Semantik und ihrer Komponenten vorgestellt. Zuerst wird das Speichermodell der Semantik analysiert und herausgestellt, dass es echt mehr Ausführungen erlaubt als ein Uni-Prozessor-System. Zudem werden Beispielprogramme vorgestellt.

Eine wesentliche Eigenschaft der Semantik ist, dass sie alle sequentiell-konsistenten Ausführungen zulässt und so mindestens alle Ausführungen erlaubt, die ein Uni-Prozessor-System zulässt.

Satz 4.15. Sei $\overrightarrow{C} \in \overrightarrow{Com_{MTL-L}}$. Die Semantik von MTL-L erlaubt jede sequentiell-konsistente Ausführung von $\overrightarrow{C}$.

Beweis. Nach Definition 3.2 ist eine sequentiell-konsistente Ausführung eine Ausführung, die die Anweisungen der Threads eines Programms in verschränkter Weise ausführt, wobei alle Prozessoren alle getätigten Schreiboperationen in gleicher Reihenfolge also entsprechend der Programmordnung sehen. Daher reicht es, zu zeigen, dass die zugrundeliegende Maschine ein Uni-Prozessor-System mit possibilistischen Scheduler simulieren kann.

Die verschränkte, also nicht parallele Ausführung, eines Programms wird explizit durch die Regel **Non-Par** der Semantik erlaubt.

Um zu erreichen, dass die Prozessoren alle getätigten Schreiboperationen in gleicher Reihenfolge sehen, kann man nach jeder ausgeführten Anweisung, die einen Schreibzugriff tätigte, den anderen Prozessor diese Schreiboperationen wahrnehmen lassen.

Sei beispielsweise $\overrightarrow{D} \in \overrightarrow{Com_{MTL-L}}$ ein noch auszuführendes Programm, dass in $\overrightarrow{C}$ startete und $D_{i,0} = x := expr$. Ferner seien $s, s' \in MEM$, sodass in s keine ungesehene Schreiboperation existiert und

$$\langle D_1 \dots D_i \dots D_n, \pi, s\rangle \rightarrowtriangle \langle D_1 \dots D_i' \dots D_n, \pi, s'\rangle$$

Dann ist i' in s' der Index der getätigten Schreiboperation $(\pi(i), x, n)$, wobei $eval_{\pi(i)}(expr, s) = n$. Im nächsten Schritt wird nun die ε-Kette, die aus einer ε-Funktion besteht, gewählt. Diese ε-Funktion lässt den entsprechenden Prozessor $1 - \pi(i)$ die Operation i' wahrnehmen.

Wählt man die ε-Kette immer entsprechend wie oben, so nehmen die Prozessoren die Operationen der anderen Operationen immer unmittelbar wahr und damit in der Reihenfolge in der sie getätigt wurden. Da durch das Ausführen einer Anweisung nur eine Schreiboperationen getätigt werden kann, ist es so nicht möglich, dass ein Prozessor Schreiboperationen des anderen in einer anderen Reihenfolge sieht. □

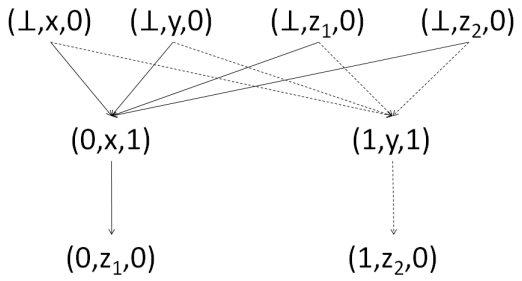
Für ein Programm, dass nur aus einem Thread besteht, ist es nicht notwendig, wie im obigen Beweis, dass der andere Prozessor die Schreiboperationen wahrnimmt. Da das System jedoch nach jeder Schreiboperation (oder später) entscheiden kann, ob der zweite Prozessor neue Operationen wahrnimmt, ist selbst für die Ausführungen eines einzelnen Threads das Verhalten der Maschine nicht deterministisch.

Im Weiteren wird gezeigt, dass das der Semantik zugrunde liegende Speichermodell ein schwaches ist. Hierzu werden die Beispiele aus Kapitel 3 herangezogen. Dadurch wird gezeigt, dass die Semantik *WriteToRead*- und *WriteToWrite*-Relaxierungen zulässt.

Als erstes Beispiel wird das Programm in Abbildung 4.6 betrachtet. Die Ausführung des Programms auf der in diesem Kapitel beschriebenen Maschine erlaubt eine nicht-sequentiell-konsistente Ausführung, wie in Kapitel 3 beschrieben.

Sei (wh, V, L, ls, i) ein Initialspeicher mit

$$wh(0) = (\bot, x, 0), wh(1) = (\bot, y, 0), wh(2) = (\bot, z_1, 0), wh(3) = (\bot, z_2, 0)$$
$$V(0) = V(1) = \varnothing, L = \varnothing, ls = \varnothing, i = 3$$



	C_1	C_2
1	$x := 1$	$y := 1$
2	$z_1 := y$	$z_2 := x$

Abbildung 4.6: Links ist das Programm. Rechts oben der Speicher nach der nicht seqentiell-konsistenten Ausführung zu sehen. Die gestrichelten Pfeile sind die Ansicht von Prozessor 1, wohingegen die nicht gestrichelten Pfeile die Ansicht von Prozessor 0 darstellen.

Startend in $\langle C_1, C_2, (wh, V, L, ls, i)\rangle$ wird als einzige gültige ε-Kette die Identität gewählt. Anschließend führt das System C_1 und C_2 parallel aus. Dadurch gelangt das System in den Zustand $\langle C_1, C_2, (wh', V', L', ls', i')\rangle$ mit

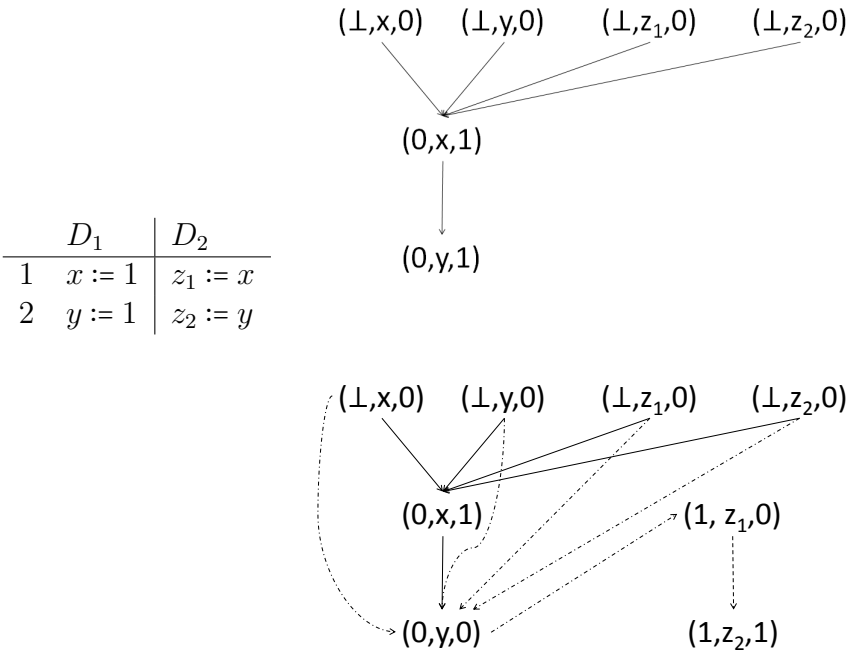
$$wh'(0) = wh(0), wh'(1) = wh(1), wh'(2) = wh(2), wh'(3) = wh(3)$$
$$wh'(4) = (0, x, 1), wh'(5) = (1, y, 1)$$
$$V(0) = \{(0,4),(1,4),(2,4),(3,4)\}, V(1) = \{(0,5),(1,5),(2,5),(3,5)\}$$
$$L = \varnothing, ls = \varnothing, i = 5$$

Für die nicht sequentiell-konsistente Ausführung, die in Kapitel 3 beschrieben wurde, wählt das System im nächsten Schritt erneut die Identität als ε-Kette, d.h. die Werte für y und x die Prozessor 0 respektive 1 lesen kann, sind die Werte der initialen Schreiboperationen, also jeweils 0. Das System führt weiter beide Threads parallel aus und landet im Endzustand, der in Abbildung 4.6 rechts oben zu sehen ist.

Als nächstes wird das Programm in Abbildung 4.7 betrachtet. Hierbei ist es möglich, dass die Ausführung auf der hier verwendeten Maschine dazu führt, dass Prozessor 1 die Schreiboperationen von Prozessor 0 in unterschiedlicher Reihenfolge sieht.

Das System startet mit dem selben Initialspeicher wie im vorherigen Beispiel, also im Zustand $\langle D_1, D_2, (wh, V, L, ls, i)\rangle$. Anschließend führt das System erst D_1 und dann $y := 1$ aus mit jeweils der Identität als ε-Kette. Der resultieren Speicher ist in Abbildung 4.7 rechts oben zu sehen.

Im nächsten Schritt wendet das System die ε-Kette $\varepsilon^1_{1,y,1}$ an, sodass Prozessor 1 die Schreiboperation $y := 1$ sieht. Anschließend führt das System die



	D_1	D_2
1	$x := 1$	$z_1 := x$
2	$y := 1$	$z_2 := y$

Abbildung 4.7: Links ist das Programm. Rechts oben der Speicher nach der Ausführung von Thread D_1 und rechts unten der Speicher nach der Ausführung des gesamten Programms.

beiden verbleibenden Anweisungen aus, wobei im nächsten Schritt wieder die Identität als ε-Kette gewählt wird. Prozessor 1 schreibt demnach 0 in z_1 und 1 in z_2 und sieht daher $y := 1$ vor $x := 1$. Der resultierende Speicher ist in Abbildung 4.7 rechts unten zu sehen.

Neben den obigen Relaxierungen garantiert das Speichermodell die gewünschten Eigenschaften, dass Speicheroperationen in einen Speicherort von den Prozessoren nicht in unterschiedlicher Reihenfolge gesehen werden können, sowie dass Synchronisationkommandos als Einweg-Speicherbarrieren dienen. Weiterhin garantiert die Verwendung von Locks, dass die Ansichten der Prozessoren an den notwendigen Stellen synchronisiert werden. Die Eigenschaften lassen sich, wie folgt, formalisieren.

Satz 4.16. Sei $(wh, V, L, ls, i) \in \mathit{MEM}$ und gültig. Dann sehen die Prozessoren Schreiboperationen in einen Speicher x nicht in unterschiedlicher Rei-

henfolge, d.h.

$$\neg\exists k, j \in dom(wh) \quad (var(wh(k)) = var(wh(j)) \wedge k\, V(0)\, j \wedge j\, V(1)\, k)$$

Beweis. Da der Speicher gültig ist, lässt sich mittels der Regeln der Semantik argumentieren, weshalb die Eigenschaft gilt. Für einen Initialspeicher gilt die Eigenschaft trivialerweise, da es für jede Variable höchstens eine Schreiboperation gibt. Der Teil der Semantik der Prozessoren fremde Schreiboperationen wahrnehmen lässt, ist der ε-Schritt. Daher reicht es aus, zu argumentieren, dass eine ε-Funktion Schreiboperationen in einen Speicherort x nicht so anordnet, sodass die Prozessoren zwei Operationen in unterschiedlicher Reihenfolge sehen. Sei k die Schreiboperation, die p durch $\varepsilon^p_{n,x,t}$ wahrnimmt. Durch Anwendung der Funktion werden alle Schreiboperation in x durch $1-p$, die vor k geschahen, auch vor k aus Sicht von p angeordnet. Somit hat p alle Operationen in x vor k von $1-p$ gesehen, wodurch p diese später (im den nächsten Schritten des Systems) nicht in umgekehrter Reihenfolge mit k sehen kann. Weiterhin werden alle Operationen in x von p vor k aus Sicht von $1-p$ angeordnet. Dadurch ist sichergestellt, dass $1-p$ in den nächsten Schritten nicht Operationen von p nach k sehen kann und somit in umgekehrter Reihenfolge. □

Die Lockordnung L erfasst, welche Restriktionen sich für die möglichen Umsortierungen im Speichermodell ergeben. Wichtig hierbei ist, dass die Prozessoren diese Ordnung respektieren, wenn sie Schreiboperationen anderer Prozessoren wahrnehmen. Dadurch ist sichergestellt, dass Synchronisationkommandos als Einweg-Barrieren fungieren.

Satz 4.17. Sei $(wh, V, L, ls, i) \in MEM$ und gültig. Dann sieht Prozessor p Schreiboperationen von $1-p$ nicht entgegen der Lockordnung, d.h

$$\forall k, j \in dom(wh) \quad (proc(wh(k)) = proc(wh(j)) = p \wedge k\, V(1-p)\, j \implies \neg j\, L\, k)$$

Beweis. Wie oben wird zuerst für einen Initialzustand und anschließend anhand der Regeln der Semantik argumentiert, dass die Eigenschaft erfüllt ist. Im Initialzustand ist die Lockordnung die leere Menge. Daher erfüllt der Speicher die Bedingung trivialerweise. Der Teil der Semantik der Prozessoren fremde Schreiboperationen wahrnehmen lässt, ist der ε-Schritt. Die gültigen ε-Ketten, die in einem ε-Schritt, verwendet werden, respektieren die Lockordnung L, sodass Prozessor p nicht Schreiboperationen von $1-p$ entgegen der Lockordnung wahrnehmen oder sehen kann. □

Satz 4.18. Sei $(wh, V, L, ls, i) \in MEM$ und gültig. Jede Schreiboperation mit Index k und $AcquireOp(k)$ in einen Lock l synchronisiert mit der letzten Release-Operation in l, d.h.

$$\forall k \in dom(wh) \quad (var(wh(k)) \in Lock \wedge AcquireOp(k) \implies$$
$$\exists j \in dom(wh)(var(wh(k)) = var(wh(j)) \wedge ReleaseOp(j) \wedge j < k \wedge j \; L \; k))$$

Beweis. Im Initialzustand existiert keine Schreiboperation k mit $AcquireOp(k)$, sodass die Eigenschaft erfüllt ist. Sobald eine Acquire-Operation durchgeführt wird, wird eine neue Schreiboperation $(p, l, 1)$ der Historie des Speichers hinzugefügt. Dies sind die lokalen Auswirkungen der Anweisung. Anschließend ordnen die globalen Auswirkungen der Semantik die Operation kleiner als die letzte Release-Operation in l in der Ordnung L an. Somit gilt die Eigenschaft. □

4.5 Alternativen und Erweiterungen

Eine Alternative zur obigen Modellierung des Speichers ist eine Formalisierung, die die einzelnen Komponenten der Hardware, die Teil des Speichers sind, explizit angibt. Dadurch ließen sich mögliche Effekte erfassen, die durch die abstrakte Darstellung des Speichers nicht auftreten. Aufgrund der zahlreichen Unterschiede der Architekturen könnte eine adäquate Modellierung, die zudem modular ist, schwierig sein.

Die der Semantik zugrunde liegende Maschine hat bisher nur zwei Prozessoren. Daher ist es wünschenswert, den Ansatz auf eine beliebige Ansatz von Prozessoren zu erweitern. Zudem ist das schwache Speichermodell noch nicht schwach genug, um auch Architekturen wie PowerPC oder das Speichermodell von C++11 zu erfassen. Als Scheduler wurde ein possibilistischer Scheduler gewählt, um die Formalisierung zu vereinfachen. Es ist aber ebenso wünschenswert, einen beliebigen Scheduler verwenden zu können.

Die oben definierte Multi-Threaded-While-Sprache bietet nicht die Möglichkeit, dass man zur Laufzeit weitere Threads erstellt. Dies diente zur Vereinfachung, jedoch ist es wünschenswert, die Sprache, um Anweisungen wie **fork** oder **spawn** zu erweitern. Hierbei ist jedoch zu beachten, dass im Fall, dass der Ersteller-Thread und der erstellte Thread auf unterschiedlichen Prozessoren laufen, eine Synchronisation der Ansichten der Prozessoren notwendig ist. Dadurch ist sichergestellt, dass der erstellte Thread auf die bisherigen Berechnungen des Ersteller-Threads zurückgreifen kann.

Eine ähnliche Problematik ergibt sich, wenn man anstatt einer statischen Verteilung von Threads auf Prozessoren eine Umverteilung während der Laufzeit zulässt. Auch hierbei muss eine Synchronisation der Ansichten erfolgen,

damit nach der Umverteilung der Thread mit seinen bisherigen Werten weiter arbeiten kann.

Eine weitere sinnvolle Erweiterung ist die explizite Modellierung von Ausgabekanälen. Diese könnten dann von Sicherheitseigenschaften verwendet werden, um Sicherheit präziser zu definieren.

Kapitel 5

Angreifer und Sicherheitseigenschaft

Nach der Formalisierung einer Semantik für MTL-L basierend auf einem Dual-Prozessor-System wird nun eine Sicherheitseigenschaft vorgestellt, die sich an *strong security* aus Kapitel 2 anlehnt. Das Kapitel beginnt mit der Vorstellung des Angreifers und dessen Modellierung. Darauf folgt die Definition der Sicherheitseigenschaft und deren Eigenschaften. Anschließend wird ein Typsystem vorgestellt. Schließlich werden Beispielprogramme auf ihre Sicherheit getestet und Alternativen und Erweiterungen aufgezeigt.

5.1 Angreifer

Essentiell für die Definition einer Sicherheitseigenschaft sind die Fähigkeiten des Angreifers. Hierzu wird zuerst eine informelle intuitive Beschreibung gegeben. Anschließend werden die Fähigkeiten des Angreifers ins Modell übertragen.

Es wird angenommen, dass dem Angreifer der Quellcode des Programms zur Verfügung steht und er bei Beobachtung der Ausführung des Programms in der Lage ist, die öffentliche Eingaben zu sehen. Beispielsweise ist es möglich Eingaben in Formulare, die das Programm anzeigt, bis auf maskierte Felder, wie Passwortfelder, zu sehen. Weiterhin ist es ihm möglich, während der Ausführung Ausgaben über öffentliche Ausgabekanäle zu sehen. Ein Ausgabekanal ist z.B. die Standardausgabe des Programms, ein Fenster auf dem Desktop oder eine Netzwerkschnittstelle. Zudem ist der Angreifer in der Lage das Scheduler-Verhalten zu approximieren.

Um die obige informelle Beschreibung des Angreifers zu formalisieren, bedarf es zu klären, was öffentliche Ausgaben und Eingaben im Modell sind.

Die Semantik für MTL-L enthält keine explizite Modellierung von Ein- und Ausgabekanälen während der Ausführung eines Programms. Daher wird die Verwendung der Kanäle approximiert durch die Nutzung eines Teils des Speichers. Hierzu werden bestimmte Variablen als öffentlich klassifiziert und es wird angenommen, dass diese zur Produktion von Ausgaben dienen. Zur Klassifizierung dient die Funktion lvl aus Kapitel 2. Diese Annahme ist berechtigt, da beispielsweise das Verändern des Inhalts eines Fensters durch das Setzen von Variablen erfolgt. Der Angreifer wird durch diese Approximation stärker, da er nun direkt einen Teil des Speichers des Programms beobachten kann. Jedoch impliziert eine Sicherheitseigenschaft basierend auf diesem Angreifer den Schutz gegen den informellen Angreifer.

Nach der informellen Beschreibung ist es dem Angreifer möglich, die Ausgaben während der Ausführung zu sehen. Da es keinen expliziten Aufruf für eine Ausgabe in der Semantik gibt, wird angenommen, dass das Programm nach jedem Schritt eine Ausgabe basierend auf den öffentlichen Variablen tätigt. Daher ist es dem Angreifer im Modell möglich, nach jedem Schritt der Ausführung den öffentlichen Speicher einzusehen. Dies stellt erneut eine Stärkung des Angreifers dar. Aufgrund dessen wird die Sicherheitseigenschaft als eine starke Bisimulation definiert. Ähnlich zu *strong security* aus Kapitel 2, bedarf es einer Klärung, welche Speicherzustände für den Angreifer ununterscheidbar sind.

Für den Angreifer sind zwei Speicherzustände ununterscheidbar, wenn zum Einen die Werte der öffentlichen Variablen übereinstimmen. Hierbei wird dem Angreifer weiter gestattet, den Wert einer Variable aus Sicht der beiden Prozessoren unterscheiden zu können. Zudem ist die Verwendung von Locks ebenfalls öffentlich beobachtbares Verhalten, sodass der Angreifer die Lockzustände sieht.

Sei

$$\begin{aligned} value :& Proc \times MEM \times Var \rightharpoonup Val \\ & (p, mem, x) \mapsto value(p, mem, x) \end{aligned}$$

die partielle Funktion, die für einen Speicher den aktuellen Wert einer Variable aus Sicht von Prozessor p zurückgibt.

Weiter sei

$$\begin{aligned} values :& MEM \times Var \rightharpoonup Val \times Val \\ & (mem, x) \mapsto (value(0, mem, x), value(1, mem, x)) \end{aligned}$$

die partielle Funktion, die einen Speicher auf die aktuellen Werte einer Variable aus Sicht von Prozessor 0 und Prozessor 1 abbildet.

Abbildung 5.1:

Die Semantik für MTL, die ein Uni-Prozessor-System verwendet, nutzt als Speicher eine Funktion von Variablen nach Werten. Dies stellte den Hauptspeicher dar. Die Semantik für MTL-L hingegen, verwendet einen komplexeren Speicher, der zusätzlich Write Buffer modelliert. Daher enthält ein solcher Speicherzustand auch Informationen über vorangegangene getätigte Speicheroperationen in öffentliche Variablen. Diese haben zusätzlich einen Einfluss auf die weitere Ausführung, sodass sie nicht vernachlässigt werden können. Im Folgenden wird herausgestellt, welche Teile des Speichers der Angreifer zusätzlich sehen muss, um die zusätzlichen Informationen über getätigte Speicheroperationen korrekt zu erfassen.

Angenommen der Angreifer sieht nur die Werte der öffentlichen Variablen und die Lockzustände. In Abbildung 5.1 sind zwei Speicher zu sehen, die für diesen Angreifer ununterscheidbar sind, da in beiden Fällen die Werte für x für beide Prozessoren die selben sind (1 und 3). Jedoch existiert im ersten Speicher eine weitere ungesehene Schreiboperation in x aus Sicht von Prozessor 0. Führt man die Anweisung **skip** ausgehend von diesen Speichern aus, so erwartet man, dass die resultieren Speicher wieder ununterscheidbar sind, da **skip** keine Auswirkung auf den Speicher hat. Das erste System kann sich jedoch dazu entscheiden, dass Prozessor 0 die Operation $(1, x, 2)$ wahrnimmt. Das zweite System hat diese Möglichkeit nicht, sodass für den Angreifer die resultierenden Speicher unterschiedlich sind. Es ist also notwendig, dass der Angreifer für jede öffentliche Variable die noch nicht gesehenen Schreiboperationen aus Sicht der beiden Prozessoren sieht.

Weiterhin ist die Ordnung der ungesehenen Schreiboperation in eine Variable x relevant. Dies geht aus den Speicherzuständen in Abbildung 5.2 hervor. Hierbei sind die ungesehenen Schreiboperation in x unterschiedlich angeordnet in beiden Speichern. Führt man ausgehend von diesen Speichern das Programm **skip**;**skip** aus, so kann das erste System im ersten Schritt Prozessor 0 $(0, x, 3)$ wahrnehmen lassen und anschließend $(0, x, 2)$. Das zweite System hingegen hat diese Möglichkeit nicht.

Zuletzt ist nicht nur die Ordnung der ungesehenen Schreiboperation in

Abbildung 5.2:

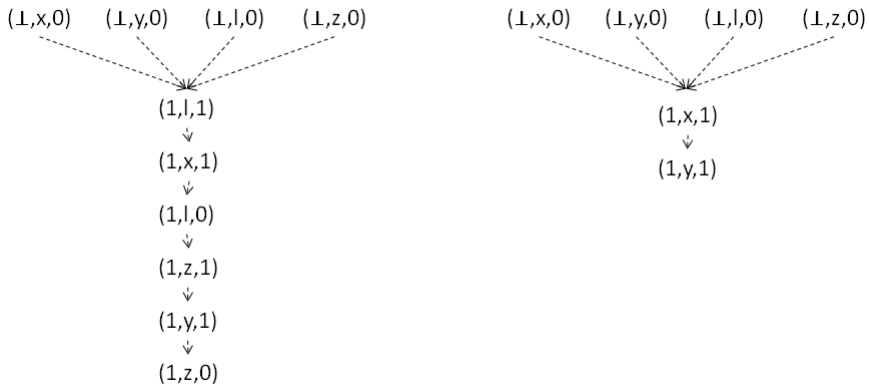


Abbildung 5.3:

einen Speicherort relevant, sondern auch die Bedingungen für die Sichtbarkeit von Schreiboperationen, die durch die Verwendung von Locks zustande gekommen sind. Hierzu sind in Abbildung 5.3 zwei Speicherzustände gegeben. Die Variablen l und z sind Lockvariablen. Im ersten Speicherzustand wird durch das Wahrnehmen von $(1, y, 1)$ das Wahrnehmen von $(1, x, 1)$ impliziert. Im zweiten Speicherzustand existiert diese Bedingung nicht. Jedoch sind die Werte der öffentlichen Variablen und der Lockzustand gleich.

Zusammenfassend sieht der Angreifer die ungesehenen Schreiboperationen in den Write Buffer sowie durch die Verwendung von Locks zustande gekommenen Bedingungen auf diesen. Dies wird als *Entscheidungsraum* eines Speichers bezeichnet. Formal ist er, wie folgt, definiert. Sei (wh, V, L, ls, i) ein Speicher. Die Menge der nicht gesehenen Schreiboperationen von p ist

$$E_{1-p} = \{k \in dom(wh) \mid lvl(var(wh(k))) = low, \neg seen(p, k)\}$$

Die Schreiboperationen in E_{1-p} in einen Speicherort x sind durch die Sicht

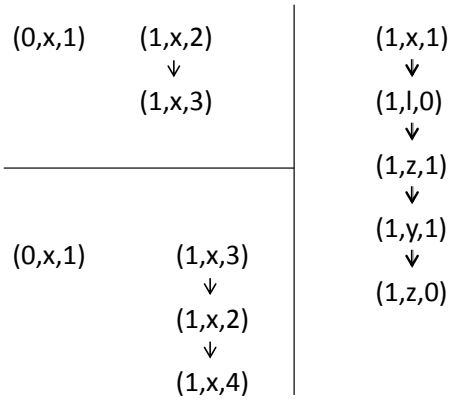


Abbildung 5.4:

von $1-p$ geordnet. Die Zusammenfassung all dieser Ordnungsrelationen ist

$$O_{1-p} = \{(k,j) \mid var(wh(k)) = var(wh(j)), k, j \in E_{1-p}, (k,j) \in V(1-p)\}$$

Die Menge aller Lockoperationen, für die gilt, dass mindestens eine Operation aus $E = E_0 \cup E_1$ kleiner ist, ist

$$\Lambda = \{k \in dom(wh) \mid var(wh(k)) \in Lock, \exists j \in E(jLk)\}$$

Die durch die Verwendung von Locks entstandenen Bedingungen bestehen aus O_0 und O_1 sowie dem relevanten Teil der Lockordnung

$$T = \{(k,j) \mid k, j \in E \cup \Lambda, (k,j) \in L\} \cup O_0 \cup O_1$$

In Abbildung 5.4 sind jeweils die Entscheidungsräume für die jeweils linken Speicher in den obigen drei Beispielen. Sie enthalten jeweils die ungesehenen Schreiboperationen und die Lockoperationen. Zudem ist die Ordnung T durch Pfeile gekennzeichnet.

Definition 5.1 (Ununterscheidbarkeit Entscheidungsraum)**.** (E_0, E_1, Λ, T) und $(E_0', E_1', \Lambda', T')$ sind ununterscheidbar, wenn gilt

1. $|E_0| = |E_0'|, |E_1| = |E_1'|$ und $|\Lambda| = |\Lambda'|$
2. es existiert eine injektive Funktion $\varphi : E \cup \Lambda \longrightarrow E' \cup \Lambda'$, sodass gilt
   (a) $\forall (k,j) \in T \quad (\varphi(k), \varphi(j)) \in T'$)
   (b) $\forall (k,j) \in T' \quad (\varphi^{-1}(k), \varphi^{-1}(j)) \in T$)

(c) $\forall k \in E_p \quad (\varphi(k) \in E'_p \wedge var(wh(k)) = var(wh(\varphi(k))) \wedge val(wh(k)) = val(wh(\varphi(k))))$

(d) $\forall k \in \Lambda \quad (\varphi(k) \in \Lambda' \wedge var(wh(k)) = var(wh(\varphi(k))) \wedge val(wh(k)) = val(wh(\varphi(k))))$

Wie oben beschrieben sind für den Angreifer zwei Speicherzustände ununterscheidbar, wenn die Werte der öffentlichen Variablen und der Lockzustand übereinstimmen und die zugehörigen Entscheidungsräume ununterscheidbar sind.

Definition 5.2 (*low*-Gleichheit)**.** Zwei Speicher $s = (wh, V, L, ls, i)$ und $s' = (wh', V', L', ls', i')$ sind *low*-gleich ($s =_L^{MPS} s'$ bezeichnet) genau dann, wenn

1. $valid(s)$ und $valid(s')$
2. $Var(s) = Var(s')$
3. $ls = ls'$
4. (E_0, E_1, Λ, T) und $(E'_0, E'_1, \Lambda', T')$ ununterscheidbar sind
5. $\forall x \in Var(s) \quad (lvl(x) = low \implies values(s) = values(s'))$

Für die späteren Eigenschaften der Sicherheitseigenschaft ist notwendig die Menge der Ausdrücke zu klassifizieren, ob sie unter ununterscheidbaren Speicherzuständen zu gleichen Werte evaluieren.

Definition 5.3 (Ununterscheidbarkeit Ausdrücke)**.** Zwei Ausdrücke *expr* und *expr'* sind ununterscheidbar (bezeichnet $expr \equiv_L expr'$), wenn gilt

$$\begin{aligned}\forall s, s' \in MEM \big(s =_L^{MPS} s' \wedge eval_0(expr, s) = n \wedge eval_0(expr', s') = n' \wedge \\ eval_1(expr, s) = m \wedge eval_1(expr', s') = m' \implies \\ n = n' \wedge m = m'\big)\end{aligned}$$

Eine wesentliche Eigenschaft von low-gleichen Speichern ist, dass für eine gültige ε-Kette für den einen Speicher eine gültige Kette für den anderen Speicher existiert, sodass nach Anwendung beider Ketten die resultieren Speicher wieder low-gleich sind.

Lemma 5.4. Seien $s, s' \in MEM$ mit $s =_L^{MPS} s'$. Dann gilt

$$\begin{aligned}&\forall C_1, C_2, \varepsilon^{p_n}_{k_n,x_n,t_n} \dots \varepsilon^{p_1}_{k_1,x_1,t_1} \quad (validChain(s, C_1, C_2, \varepsilon^{p_n}_{k_n,x_n,t_n} \dots \varepsilon^{p_1}_{k_1,x_1,t_1})) \\ &\implies \exists \varepsilon^{p'_m}_{k'_m,x'_m,t'_m} \dots \varepsilon^{p'_1}_{k'_1,x'_1,t'_1} \quad (validChain(s', C_1, C_2, \varepsilon^{p'_m}_{k'_m,x'_m,t'_m} \dots \varepsilon^{p'_1}_{k'_1,x'_1,t'_1}) \\ &\qquad \wedge \varepsilon^{p_n}_{k_n,x_n,t_n} \dots \varepsilon^{p_1}_{k_1,x_1,t_1}(s) =_L^{MPS} \varepsilon^{p'_m}_{k'_m,x'_m,t'_m} \dots \varepsilon^{p'_1}_{k'_1,x'_1,t'_1}(s')))\end{aligned}$$

Beweis. Sei $C_1, C_2 \in Com_{MTL\text{-}L}$ und $\varepsilon^{p_n}_{k_n,x_n,t_n} \dots \varepsilon^{p_1}_{k_1,x_1,t_1}$ beliebig, sodass $validChain(s, C_1, C_2, \varepsilon^{p_n}_{k_n,x_n,t_n} \dots \varepsilon^{p_1}_{k_1,x_1,t_1})$.

Sei $\varepsilon^{p^\circ_q}_{k^\circ_q,x^\circ_q,t^\circ_q} \dots \varepsilon^{p^\circ_1}_{k^\circ_1,x^\circ_1,t^\circ_1}$ die Teilkette von $\varepsilon^{p_n}_{k_n,x_n,t_n} \dots \varepsilon^{p_1}_{k_1,x_1,t_1}$, mit $lvl(x^\circ_i) = low$. Da $s =^{MPS}_{L} s'$, sind die Anzahl der ungesehen low-Operationen, sowie deren Ordnung, gleich. Damit sind die ε-Funktionen von $\varepsilon^{p^\circ_q}_{k^\circ_q,x^\circ_q,t^\circ_q} \dots \varepsilon^{p^\circ_1}_{k^\circ_1,x^\circ_1,t^\circ_1}$ jeweils anwendbar auf s'. Zudem gilt nach Satz 4.14 1., dass durch die Anwendung einer ε-Funktion die Sichtbarkeit der übrigen getätigten Operation gleich bleibt. Damit ist die genannte Teilkette ebenfalls anwendbar auf s'. Weiterhin ist die Reihenfolge gültig, da in s und s' die Lockordnung bezüglich der nicht gesehenen low-Operationen die gleiche ist. Somit lässt sich die Teilkette zu einer gültigen Kette $\varepsilon^{p'_m}_{k'_m,x'_m,t'_m} \dots \varepsilon^{p'_1}_{k'_1,x'_1,t'_1}$ für s' vervollständigen.

Es verbleibt zu zeigen, dass

$$\varepsilon^{p_n}_{k_n,x_n,t_n} \dots \varepsilon^{p_1}_{k_1,x_1,t_1}(s) =^{MPS}_{L} \varepsilon^{p'_m}_{k'_m,x'_m,t'_m} \dots \varepsilon^{p'_1}_{k'_1,x'_1,t'_1}(s')$$

Nach Definition einer ε-Funktion verändert diese nicht den Lockzustand, sodass diese trivialerweise gleich sind. Weiter gilt nach Satz 4.14 2., dass die Werte der low-Variablen entweder in beiden Speichern zum gleichen Wert geändert werden oder gleich bleiben durch die ε-Funktionen. Seien (E_0, E_1, Λ, T) und $(E'_0, E'_1, \Lambda', T')$ die Entscheidungsräume von s und s'. Die Anwendung einer Funktion $\varepsilon^{p^\circ_i}_{k^\circ_i,x^\circ_i,n^\circ_i}$ aus der Kette $\varepsilon^{p^\circ_q}_{k^\circ_q,x^\circ_q,t^\circ_q} \dots \varepsilon^{p^\circ_1}_{k^\circ_1,x^\circ_1,t^\circ_1}$ bewirkt das Entfernen aller Schreiboperation in x°_i aus $E_{p^\circ_i}$ und $E'_{p^\circ_i}$ und das Entfernen der ersten n°_i Schreiboperation in x°_i aus $E_{1-p^\circ_i}$ und $E'_{1-p^\circ_i}$. Somit werden die Mengen auf gleiche Weise reduziert und ebenso die Ordnungen T und T'. Daher sind die Entscheidungsräume nach Anwendung der Teilkette $\varepsilon^{p^\circ_q}_{k^\circ_q,x^\circ_q,t^\circ_q} \dots \varepsilon^{p^\circ_1}_{k^\circ_1,x^\circ_1,t^\circ_1}$ wieder ununterscheidbar und es gilt

$$\varepsilon^{p_n}_{k_n,x_n,t_n} \dots \varepsilon^{p_1}_{k_1,x_1,t_1}(s) =^{MPS}_{L} \varepsilon^{p'_m}_{k'_m,x'_m,t'_m} \dots \varepsilon^{p'_1}_{k'_1,x'_1,t'_1}(s')$$

□

5.2 Starke MPS Sicherheit

Aufbauend auf der Ansicht des Angreifers auf den Speicher durch die low-Gleichheit und der Wahl einer starken Bisimulation wird im Folgenden eine Sicherheitseigenschaft angegeben, die ähnlich zur starken Low-Bisimulation ist.

Definition 5.5 (Starke Low-MPS[1]-Bisimulation)**.** Eine Relation $R \subset \overrightarrow{Com_{MTL-L}} \times \overrightarrow{Com_{MTL-L}}$ ist eine starke Low-MPS-Bisimulation auf Programmen, wenn

1. $\forall \overrightarrow{C}, \overrightarrow{D} \in \overrightarrow{Com_{MTL-L}} (\overrightarrow{C}\ R\ \overrightarrow{D} \implies |\overrightarrow{C}| = |\overrightarrow{D}|)$
2. $$\begin{aligned}\forall \pi\ \forall s_1, s_2, s_1' \in MEM\ \forall C_i', C_j' \in Com_{MTL-L} (s_1 =_L^{MPS} s_2 \wedge \overrightarrow{C}\ R\ \overrightarrow{D} \wedge \\ \langle C_1 \dots C_i \dots C_j \dots C_n, \pi, s_1\rangle \twoheadrightarrow \langle C_1 \dots C_i' \dots C_j' \dots C_n, \pi, s_1'\rangle \\ \implies \exists s_2' \in MEM\ \exists D_i', D_j' \in Com_{MTL-L} \\ (\langle D_1 \dots D_i \dots D_j \dots D_n, \pi, s_2\rangle \twoheadrightarrow \langle D_1 \dots D_i' \dots D_j' \dots D_n, \pi, s_2'\rangle \\ \wedge s_1' =_L^{MPS} s_2' \wedge C_i'\ R\ D_i' \wedge C_j'\ R\ D_j'))\end{aligned}$$
3. R ist symmetrisch

Die Bisimulation erfasst *Noninterference*, indem für ein Programm $\overrightarrow{C}$ und startend in $s_1 =_L^{MPS} s_2$, aber $s_1 \neq s_2$, gefordert wird, dass die Ausführungen wieder in ununterscheidbaren Speichern resultieren und so durch die Anweisung des Programms kein Informationsfluss in den öffentlichen Speicher stattfand.

Die Vereinigung aller starken Low-MPS-Bisimulation auf Programmen wird mit $\approx_L^{MPS}$ bezeichnet.

Ein Programm $\overrightarrow{C} \in \overrightarrow{Com_{MTL-L}}$ ist *stark MPS sicher* genau dann, wenn $\overrightarrow{C} \approx_L^{MPS} \overrightarrow{C}$.

Korollar 5.6. Die Relation $\approx_L^{MPS}$ ist eine starke Low-MPS-Bisimulation auf Programmen.

Beweis. Seien $\overrightarrow{C}, \overrightarrow{D} \in \overrightarrow{Com_{MTL-L}}$ mit $\overrightarrow{C} \approx_L^{MPS} \overrightarrow{D}$. Dann existiert ein starke Low-MPS-Bisimulation R mit $\overrightarrow{C}\ R\ \overrightarrow{D}$. Daher gilt $|\overrightarrow{C}| = |\overrightarrow{D}|$. Weiterhin ist $\overrightarrow{D}\ R\ \overrightarrow{C}$ und somit $\overrightarrow{D} \approx_L^{MPS} \overrightarrow{C}$. Es verbleibt zu zeigen, dass $\approx_L^{MPS}$ die 2. Bedingung in Definition 5.5 erfüllt. Aus $\overrightarrow{C}\ R\ \overrightarrow{D}$ folgt die Existenz von D_i', D_j' mit $C_i'\ R\ D_i'$ und $C_j'\ R\ D_j'$. Da $R \subset \approx_L^{MPS}$, gilt ebenso $C_i' \approx_L^{MPS} D_i'$ und $C_j' \approx_L^{MPS} D_j'$. Somit ist $\approx_L^{MPS}$ eine starke Low-MPS-Bisimulation auf Programmen. □

Programme, die zu sich selbst in der Relation $\approx_L^{MPS}$ stehen, sind sicher gegen den obigen Angreifer. Daraus folgt, dass unsichere Programme nicht zu sich selbst und damit zu keinem anderen Programm in Relation stehen. Das bedeutet, dass $\approx_L^{MPS}$ nicht reflexiv und damit eine partielle Äquivalenzrelation ist. Dies zeigt folgender Satz.

[1]MPS = Multi-Prozessor-System

Satz 5.7. Die Relation $\approx_L^{MPS}$ ist nicht reflexiv.

Beweis. Angenommen $\approx_L^{MPS}$ ist reflexiv. Dann gilt $\langle l := h\rangle \approx_L^{MPS} \langle l := h\rangle$, wobei $lvl(l) = low$ und $lvl(h) = high$. Das heißt insbesondere, dass Bedingung (2) in Definition 5.5 erfüllt ist.

Sei π beliebig mit $\pi(1) = p$. Ferner seien s_1 und s_2 zwei Initialspeicher für das Programm $\langle l := h\rangle$, wobei l in beiden den Wert 0 hat und h in s_1 den Wert 1 und in s_2 den Wert 0. Dann gilt $s_1 =_L^{MPS} s_2$, da der Wert von l in beiden gleich ist.

Folgt man den Regeln der Semantik, so gilt

$$\langle l := h, \pi, s_1\rangle \rightarrowtriangle \langle\langle\rangle, \pi, s_1'\rangle$$
$$\langle l := h, \pi, s_2\rangle \rightarrowtriangle \langle\langle\rangle, \pi, s_2'\rangle$$

wobei jeweils als ε-Kette die Identität gewählt werden musste. Die resultierenden Speicher s_1' und s_2' sind jedoch aus Sicht des Angreifers nicht mehr ununterscheidbar, da in s_1 der aktuelle Wert für l 1 ist und in s_2 0. Da die Semantik keine weitere Möglichkeit für ein s_2' zulässt, kann $\langle l := h\rangle \approx_L^{MPS} \langle l := h\rangle$ nicht gelten und $\approx_L^{MPS}$ ist nicht reflexiv. □

Beobachtung 5.8. Um für eine starke Low-MPS-Bisimulation R zu zeigen, dass Bedingung 2 erfüllt ist, muss stets gezeigt werden, dass für einen Schritt im ersten System ein ähnlicher im zweiten System möglich ist. Ein Teil beider Schritte ist die Anwendung einer ε-Kette. Sei $s_1 =_L^{MPS} s_2$ und $\varepsilon^{p_n}_{k_n,x_n,t_n} \ldots \varepsilon^{p_1}_{k_1,x_1,t_1}$ die Kette, die im Schritt

$$\langle C_1 \ldots C_i \ldots C_j \ldots C_n, \pi, s_1\rangle \rightarrowtriangle \langle C_1 \ldots C_i' \ldots C_j' \ldots C_n, \pi, s_1'\rangle$$

angewandt wurde. Dann existiert nach Lemma 5.4 $\varepsilon^{p_m'}_{k_m',x_m',t_m'} \ldots \varepsilon^{p_1'}_{k_1',x_1',t_1'}$ mit

$$\varepsilon^{p_n}_{k_n,x_n,t_n} \ldots \varepsilon^{p_1}_{k_1,x_1,t_1}(s_1) =_L^{MPS} \varepsilon^{p_m'}_{k_m',x_m',t_m'} \ldots \varepsilon^{p_1'}_{k_1',x_1',t_1'}(s_2)$$

Somit lässt sich dieser Schritt in der Beweisführung, dass R Bedingung 2 erfüllt, auslassen.

Es werden nun eine Reihe von Sätzen bewiesen, die aufzeigen, welche Programme unter der obigen Eigenschaft als sicher klassifiziert werden. Die Vorgehensweise ist hierbei stets, dass eine Relation R definiert und anschließend gezeigt wird, dass R eine starke Low-MPS-Bisimulation auf Programmen ist. Hierbei ist vor allem die 2. Bedingung der Definition 5.5 der nicht triviale Teil des Beweises.

Lemma 5.9. Es gilt:

1. $\langle\rangle \approx_L^{MPS} \langle\rangle$
2. $\langle \mathbf{skip} \rangle \approx_L^{MPS} \langle \mathbf{skip} \rangle$
3. $lvl(x) = high \implies \langle x := expr \rangle \approx_L^{MPS} \langle x := expr' \rangle$

Beweis. 1. Folgt unmittelbar aus der Definition von $\approx_L^{MPS}$ und der Semantik.

2. Sei $R = \{(\langle \mathbf{skip} \rangle, \langle \mathbf{skip} \rangle), (\langle\rangle, \langle\rangle)\}$. Es wird nun gezeigt, dass R eine starke Low-MPS-Bisimulation auf Programmen ist.

   Aus der Definition von R folgt, dass R symmetrisch ist und R Threadpools gleicher Größe in Relation setzt.

   Sei π eine beliebige Verteilung. Weiter seien s_1,s_2 und s_1' Speicher mit $s_1 =_L^{MPS} s_2$ so, dass

   $$\langle \langle \mathbf{skip} \rangle, \pi, s_1 \rangle \rightarrowtriangle \langle \langle\rangle, \pi, s_1' \rangle$$

   mit zugehöriger ε-Kette $\varepsilon^{p_n}_{k_n,x_n,t_n} \ldots \varepsilon^{p_1}_{k_1,x_1,t_1}$. Da die lokalen und globalen Auswirkungen von **skip** keine Änderungen an einem Speicher vornehmen, gilt $s_1' = \varepsilon^{p_n}_{k_n,x_n,t_n} \ldots \varepsilon^{p_1}_{k_1,x_1,t_1}(s_1)$.

   Nach Lemma 5.4 existiert eine Kette $\varepsilon^{p_m'}_{k_m',x_m',t_m'} \ldots \varepsilon^{p_1'}_{k_1',x_1',t_1'}$, sodass

   $$s_1' =_L^{MPS} \varepsilon^{p_n}_{k_n,x_n,t_n} \ldots \varepsilon^{p_1}_{k_1,x_1,t_1}(s_1) =_L^{MPS} \varepsilon^{p_m'}_{k_m',x_m',t_m'} \ldots \varepsilon^{p_1'}_{k_1',x_1',t_1'}(s_2)$$

   Sei $s_2' = \varepsilon^{p_m'}_{k_m',x_m',t_m'} \ldots \varepsilon^{p_1'}_{k_1',x_1',t_1'}(s_2)$. Dann gilt

   $$\langle \langle \mathbf{skip} \rangle, \pi, s_2 \rangle \rightarrowtriangle \langle \langle\rangle, \pi, s_2' \rangle$$

   und $s_2' =_L^{MPS} s_1'$. Weiter gilt $(\langle\rangle, \langle\rangle) \in R$. Somit ist R eine starke Low-MPS-Bisimulation auf Programmen und $\langle \mathbf{skip} \rangle \approx_L^{MPS} \langle \mathbf{skip} \rangle$.

3. Sei $R = \{(\langle h := expr \rangle, \langle h := expr' \rangle) \mid lvl(h) = high\} \cup \{(\langle\rangle, \langle\rangle)\}$. Es wird nun gezeigt, dass R eine starke Low-MPS-Bisimulation auf Programmen ist.

   Aus der Definition von R folgt, dass R symmetrisch ist und R Threadpools gleicher Größe in Relation setzt.

Sei π eine beliebige Verteilung. Weiter seien s_1,s_2 und s_1' Speicher mit $s_1 =_L^{MPS} s_2$ so, dass

$$\langle\langle h := expr\rangle, \pi, s_1\rangle \rightarrowtriangle \langle\langle\rangle, \pi, s_1'\rangle$$

mit zugehöriger ε-Kette $\varepsilon^{p_n}_{k_n,x_n,t_n} \dots \varepsilon^{p_1}_{k_1,x_1,t_1}$.

Es wird nun argumentiert, dass die lokalen und globalen Auswirkungen einer Schreiboperation in eine *high*-Variable keinen Effekt auf den vom Angreifer beobachtbaren Teil des Speichers hat.

Die lokalen Auswirkungen einer Schreiboperation j ordnen diese größer (nach $V(p)$) als alle gesehenen Schreiboperationen von p an. Dadurch ändert sich nur der Wert in h für p, aber nicht den Wert der anderen Variablen. Zudem verändern die weiteren Auswirkungen nicht den Lockzustand. Da die neu hinzugefügte Schreiboperation weder eine *low*-Variable noch eine Lockvariable betrifft, ändert sich nichts an den Mengen E und Λ des Entscheidungsraums von s_1. Somit gilt $s_1' =_L^{MPS} s_1$.

Gleiches gilt daher auch für s_2 und s_2', also $s_2 =_L^{MPS} s_2'$.

Somit ist $s_2' =_L^{MPS} s_1'$. Da weiter $(\langle\rangle, \langle\rangle) \in R$, ist R eine starke Low-MPS-Bisimulation auf Programmen und $\langle x := expr\rangle \approx_L^{MPS} \langle x := expr'\rangle$.

□

Lemma 5.10. Es gilt:

1. $\langle l.acquire\rangle \approx_L^{MPS} \langle l.acquire\rangle$
2. $\langle l.release\rangle \approx_L^{MPS} \langle l.release\rangle$
3. $expr \equiv_L expr' \implies \langle x := expr\rangle \approx_L^{MPS} \langle x := expr'\rangle$

Beweis. 1. Sei $R = \{(\langle l.acquire\rangle, \langle l.acquire\rangle), (\langle\rangle, \langle\rangle)\}$. Es wird nun gezeigt, dass R eine starke Low-MPS-Bisimulation auf Programmen ist.

Aus der Definition von R folgt, dass R symmetrisch ist und R Threadpools gleicher Größe in Relation setzt.

Sei π eine beliebige Verteilung mit $\pi(1) = p$. Weiter seien s_1,s_2 und s_1' Speicher mit $s_1 =_L^{MPS} s_2$ so, dass

$$\langle\langle l.acquire\rangle, \pi, s_1\rangle \rightarrowtriangle \langle\langle\rangle, \pi, s_1'\rangle$$

Nach Beobachtung 5.8 wird der ε-Schritt ausgelassen.

Es verbleibt zu zeigen, dass die lokalen und globalen Auswirkungen von l.acquire beide Speicher s_1 und s_2 wieder auf ununterscheidbare Speicher abbilden.

Die lokalen Auswirkungen von l.acquire sind die selben wie die eines Schreibzugriffs. Daher gilt, wie im Beweis von 5.9, dass die Werte aller anderen Variablen, also auch der low-Variablen, unverändert bleiben. Der Lockzustand und die Lockordnung werden in diesem Schritt nicht verändert und sind demnach auch weiterhin ununterscheidbar. Es gilt also, dass die Speicher s_1^+ und s_2^+, die aus s_1 respektive s_2 durch die lokalen Auswirkungen resultieren, wieder ununterscheidbar sind.

Ein Teil der globalen Auswirkungen von l.acquire ist die Abänderung des Lockzustands, sodass l als angeeignet markiert wird. Da die Lockzustände in s_1^+ und s_2^+ vorher gleich waren und in gleicher Weise verändert werden, sind die Lockzustände nach den globalen Auswirkungen erneut gleich. Ferner ändern die globalen Auswirkungen nicht die Werte der Variablen.

Seien (E_0, E_1, Λ, T) und $(E_0', E_1', \Lambda', T')$ die Entscheidungsräume und L_1' und L_2' die Lockordnungen von $s_1' = globalAcquire(p, l, s_1^+)$ respektive $s_2' = globalAcquire(p, l, s_2^+)$. Seien i und i' die Indizes der zuletzt getätigten Schreiboperation (und damit die Operationen $(p, l, 1)$).

Angenommen $i \in \Lambda$ und $i' \notin \Lambda'$. Dann existiert ein $e \in E$ und $j \in \Lambda$, sodass $e\ L_1^+\ j\ L_1^+\ i$. Hierbei ist j eine Release-Operation. Da beide Entscheidungsräume vorher ununterscheidbar waren, existieren ebenso $e' \in E$ und $j' \in \Lambda$, sodass $e'\ L_2^+\ j'$. Ferner ist j' eine Release-Operation. Da i mit j synchronisiert, muss auch i' mit j' synchronisieren (entweder sind die Variablen die gleichen oder die Prozessoren). Daher gilt $e'\ L_2^+\ j'\ L_2^+\ i'$ und $i' \in \Lambda$. Die Argumentation folgt analog, wenn man annimmt, dass $i \notin \Lambda$ und $i \in \Lambda$. Somit ist $i \in \Lambda \iff i' \in \Lambda$.

Im Fall $i \notin \Lambda$ und $i' \notin \Lambda'$ sind die Entscheidungsräume die selben wie vor der Anwendung der globalen Auswirkungen und somit ununterscheidbar.

Sei nun $i \in \Lambda$ und $i' \in \Lambda'$. Die Anpassung der Lockordnungen durch die globalen Auswirkungen ordnen i und i' größer als die zuvor getätigte Lockoperationen von p und größer als die letzte Release-Operation in l. Da die Entscheidungsräume vorher ununterscheidbar waren, werden in beiden Fällen die gleiche Anzahl an Relationen zu den Lockordnungen hinzugefügt. Damit sind beide Entscheidungsräume wieder ununterscheidbar. Es gilt also $s_1' =_L^{MPS} s_2'$ und s_2' ist der gesuchte Speicher.

Da $(\langle\rangle, \langle\rangle) \in R$, ist die zweite Bedingung einer starken Low-MPS-Bisimulation erfüllt und R eine eben solche. Somit gilt
$\langle l.acquire\rangle \approx_L^{MPS} \langle l.acquire\rangle$.

2. Sei $R = \{(\langle l.release\rangle, \langle l.release\rangle), (\langle\rangle, \langle\rangle)\}$. Es wird nun gezeigt, dass R eine starke Low-MPS-Bisimulation auf Programmen ist.

   Aus der Definition von R folgt, dass R symmetrisch ist und R Threadpools gleicher Größe in Relation setzt.

   Sei π eine beliebige Verteilung mit $\pi(1) = p$. Weiter seien s_1,s_2 und s_1' Speicher mit $s_1 =_L^{MPS} s_2$ so, dass

   $$\langle\langle l.release\rangle, \pi, s_1\rangle \rightarrowtriangle \langle\langle\rangle, \pi, s_1'\rangle$$

   Wie im Fall von l.acquire folgt, dass man eine geeignete ε-Kette finden kann, sodass man nach Anwendung beider Ketten auf s_1 und s_2 und der lokalen Auswirkungen zwei ununterscheidbare Speicher $s_1^+ =_L^{MPS} s_2^+$ erreicht. Es verbleibt demnach noch zu zeigen, dass die globalen Auswirkungen von l.release beide Speicher wieder auf ununterscheidbare Speicher abbilden.

   Seien (E_0, E_1, Λ, T) und $(E_0', E_1', \Lambda', T')$ die Entscheidungsräume und L_1' und L_2' die Lockordnungen von $s_1' = globalRelease(p, l, s_1^+)$ respektive $s_2' = globalRelease(p, l, s_2^+)$. Seien i und i' die Indizes der zuletzt getätigten Schreiboperation (und damit die Operationen $(p, l, 0)$). Die globalen Auswirkungen ordnen i und i' größer als alle von p getätigten Operationen an. Das bedeutet, dass $i \in \Lambda$, wenn $E_p \neq \varnothing$. Da die vorherigen Entscheidungsräume von s_1^+ und s_2^+ ununterscheidbar sind und die globalen Auswirkungen von l.release die Mengen E und E' nicht ändern, gilt $E_p = \varnothing \iff E_p' = \varnothing$. Damit gilt $i \in \Lambda \iff i' \in \Lambda'$.

   Im Fall $i \notin \Lambda$ sind daher die Entscheidungsräume die selben wie vor den globalen Auswirkungen und somit ununterscheidbar.

   Angenommen $i \in \Lambda$ und $i' \in \Lambda'$. Dann sind beide Operationen größer (nach L_1^+ und L_2^+) als die zuletzt getätigte Operation von p. Damit sind sie auch größer als die zuletzt getätigten Operation in low-Variablen von p (diese existieren, da $E_p \neq \varnothing$ und $E_p' \neq \varnothing$). Da dies die einzige Änderung bezüglich der Lockordnung ist, wurden T und T' die gleichen Relationen hinzugefügt, sodass beide Entscheidungsräume ununterscheidbar sind. Es gilt also $s_1' =_L^{MPS} s_2'$.

   Da $(\langle\rangle, \langle\rangle) \in R$, ist die zweite Bedingung einer starken Low-MPS-Bisimulation erfüllt und R eine eben solche. Somit gilt $\langle l.release\rangle \approx_L^{MPS} \langle l.release\rangle$.

3. Sei

   $$R = \{(\langle x := expr\rangle, \langle x := expr'\rangle) \mid lvl(x) = low, expr \equiv_L expr'\} \cup \{(\langle\rangle, \langle\rangle)\}$$

Es wird nun gezeigt, dass R eine starke Low-MPS-Bisimulation auf Programmen ist.

Aus der Definition von R folgt, dass R symmetrisch ist und R Threadpools gleicher Größe in Relation setzt.

Sei π eine beliebige Verteilung mit $\pi(1) = p$. Weiter seien s_1,s_2 und s_1' Speicher mit $s_1 =_L^{MPS} s_2$ so, dass

$$\langle\langle x := expr\rangle, \pi, s_1\rangle \rightarrowtriangle \langle\langle\rangle, \pi, s_1'\rangle$$

Die lokalen Auswirkungen der Schreiboperation ändern in beiden Fällen den Wert für x aus Sicht von p. Die Werte der restlichen Variablen bleiben die selben. Da $expr \equiv_L expr'$ sind die Werte n,n', die x zugewiesen werden, die selben ($n = n'$). Somit sind nach den lokalen Auswirkungen die Speicher s_1^+ und s_2^+ wieder ununterscheidbar

Seien (E_0, E_1, Λ, T) und $(E_0', E_1', \Lambda', T')$ die Entscheidungsräume und L_1' und L_2' die Lockordnungen von $s_1' = globalWrite(p, s_1^+)$ respektive $s_2' = globalWrite(p, s_2^+)$. Seien i und i' die Indizes der zuletzt getätigten Schreiboperation (und damit die Operationen (p, x, n) und (p, x, n')). Die globalen Auswirkungen ordnen i und i' größer (bezüglich der Lockordnung) als die letzte getätigte Acquire-Operationen von Prozessor p an. Das bedeutet zum Einen, dass $i \in E_p$ und $i' \in E_p'$. Zum Anderen wurden, da beide Entscheidungsräume zuvor ununterscheidbar waren, die gleichen Relationen hinzugefügt, sodass die Entscheidungsräume wieder ununterscheidbar sind. Damit gilt $s_1' =_L^{MPS} s_2'$. Da $(\langle\rangle, \langle\rangle) \in R$, ist R eine starke low-MPS-Bisimulation auf Programmen und
$\langle x := expr\rangle \approx_L^{MPS} \langle x := expr'\rangle$.

□

Korollar 5.11. Es gilt:

1. $l.acquire \approx_L^{MPS} C \implies C = l.acquire$
2. $l.release \approx_L^{MPS} C \implies C = l.release$
3. $x := expr \approx_L^{MPS} C \wedge lvl(x) = low \implies C = x := expr' \wedge expr' \equiv_L expr$

Beweis. 1. Eine Acquire-Anweisung in l ist die einzige Anweisung, die den Lockzustand eines Speichers dahingehend ändert, dass der Lock l als angeeignet gesetzt wird. Daher muss $C = l.acquire$, da ein Angreifer andernfalls beide Programme nach einem Schritt unterscheiden kann. $\langle l.acquire\rangle \approx_L^{MPS} C$ folgt dann aus Lemma 5.10.

2. Eine Release-Anweisung in l ist die einzige Anweisung, die einen Lock wieder freigeben kann. Daher muss $C = l.release$ gelten, da ansonsten ein Angreifer beide Programme anhand des Lockzustands unterscheiden kann. $\langle l.release \rangle \approx_L^{MPS} C$ folgt dann aus Lemma 5.10.

3. Sei $x := expr \approx_L^{MPS} C$ und $lvl(x) = low$. Angenommen $C \neq x := expr'$. Dann lassen sich beide Programme anhand der Werte von x unterscheiden. Daher muss $C = x := expr'$ gelten.

   Angenommen $\neg(expr' \equiv_L expr)$. Dann existieren zwei ununterscheidbare Speicher für die beide Ausdrücke für mindestens einen Prozessor zu unterschiedlichen Werten evaluieren. Somit lassen sich beide Programme anhand der unterschiedlichen Werte in x unterscheiden. $\langle x := expr \rangle \approx_L^{MPS} C$ folgt dann aus Lemma 5.10.

□

In den obigen Beweisen wurde stets die Bisimulationsrelation R explizit benannt. Für die folgenden Resultate ist die Angabe einer Relation R schwierig, da eine Vielzahl an Threadpoolpaaren benannt werden muss. Um die Beweise zu vereinfachen, wird die sogenannte Up-To-Technik oder Bis-Zu-Technik verwendet. Dadurch lässt sich eine kleinere Bisimulationsrelation definieren, die jedoch die Eigenschaft hat, dass Threadpoolpaare in dieser Relation auch in $\approx_L^{MPS}$ enthalten sind.

Definition 5.12. Eine Relation $R \subset \overrightarrow{Com_{MTL\text{-}L}} \times \overrightarrow{Com_{MTL\text{-}L}}$ ist eine *starke Low-MPS-Bisimulation auf Programmen bis zu* $\approx_L^{MPS}$, wenn gilt

1. $\forall \vec{C}, \vec{D} \in \overrightarrow{Com_{MTL\text{-}L}} (\vec{C}\ R\ \vec{D} \implies |\vec{C}| = |\vec{D}|)$

2. $$\begin{aligned}
   &\forall \pi\, \forall s_1, s_2, s_1' \in MEM\ \forall C_i', C_j' \in Com_{MTL\text{-}L} (s_1 =_L^{MPS} s_2 \wedge \vec{C}\ R\ \vec{D} \wedge \\
   &\quad \langle C_1 \ldots C_i \ldots C_j \ldots C_n, \pi, s_1 \rangle \twoheadrightarrow \langle C_1 \ldots C_i' \ldots C_j' \ldots C_n, \pi, s_1' \rangle \\
   &\quad \implies \exists s_2' \in MEM\ \exists D_i', D_j' \in Com_{MTL\text{-}L} \\
   &\quad (\langle D_1 \ldots D_i \ldots D_j \ldots D_n, \pi, s_2 \rangle \twoheadrightarrow \langle D_1 \ldots D_i' \ldots D_j' \ldots D_n, \pi, s_2' \rangle \\
   &\quad \wedge s_1' =_L^{MPS} s_2' \wedge \langle C_i' \rangle\ (R \cup \approx_L^{MPS})^+\ \langle D_i' \rangle \wedge \langle C_j' \rangle\ (R \cup \approx_L^{MPS})^+\ \langle D_j' \rangle))
   \end{aligned}$$

3. R ist symmetrisch

Der einzige Unterschied zu Definition 5.5 ist, dass die noch auszuführenden Threads C_i', D_i', C_j' und D_j' nicht mehr nur in Relation R, sondern in der größeren Relation $(R \cup \approx_L^{MPS})^+$ stehen. Dadurch reduziert sich die Anzahl der Paare, die in R enthalten sein müssen. Der folgende Satz liefert die Rechtfertigung für die Nutzung dieser kleineren Relation.

Satz 5.13. Sei $R \subset \overrightarrow{Com_{MTL-L}} \times \overrightarrow{Com_{MTL-L}}$ eine *starke Low-MPS-Bisimulation auf Programmen bis zu* $\approx_L^{MPS}$.

Wenn $\overrightarrow{C}\ R\ \overrightarrow{C'}$, dann gilt $\overrightarrow{C} \approx_L^{MPS} \overrightarrow{C'}$.

Beweis. Sei $Q = (R \cup \approx_L^{MPS})^+$. Es ist zu zeigen, dass $Q \subset \approx_L^{MPS}$ und damit $R \subset Q \subset \approx_L^{MPS}$. Hierzu wird gezeigt, dass Q eine starke Low-MPS-Bisimulation auf Programmen ist.

Da R und $\approx_L^{MPS}$ Threadpools gleicher Größe in Relation setzen und symmetrisch sind, gilt dies ebenso für Q. Die 2. Bedingung der Bisimulation wird mittels Induktion über die Q-Distanz zweier Threadpools in Q bewiesen.

Die Q-Distanz zwischen $\overrightarrow{C}, \overrightarrow{D} \in \overrightarrow{Com_{MTL-L}}$ mit $\overrightarrow{C}\ Q\ \overrightarrow{D}$ ist die minimale Länge einer Folge $\overrightarrow{V_1}, \ldots, \overrightarrow{V_n}$, für die gilt

$$\forall i \in \{0, \ldots, n\} \quad (\overrightarrow{V_i}(R \cup \approx_L^{MPS})\overrightarrow{V_{i+1}})$$

wobei $\overrightarrow{V_0} = \overrightarrow{C}$ und $\overrightarrow{V_{n+1}} = \overrightarrow{D}$.

Sei $\overrightarrow{C}\ Q\ \overrightarrow{D}$ mit Distanz 0. Dann gilt $\overrightarrow{C}\ R\ \overrightarrow{D}$ oder $\overrightarrow{C} \approx_L^{MPS} \overrightarrow{D}$. Daher gilt für die Implikation in Bedingung 2, dass $C_i'\ Q\ D_i'$ und $C_j'\ Q\ D_j'$ oder $C_i' \approx_L^{MPS} D_i'$ und $C_j' \approx_L^{MPS} D_j'$. Aus $\approx_L^{MPS} \subset Q$ folgt, dass die 2. Bedingung damit erfüllt ist. Es wird angenommen, dass die Behauptung für n gilt.

Sei $\overrightarrow{C}\ Q\ \overrightarrow{D}$ mit Distanz $n+1$. Somit existiert eine Folge $\overrightarrow{V_1}, \ldots, \overrightarrow{V_{n+1}}$, sodass

$$\forall i \in \{0, \ldots, n+1\} + \overrightarrow{V_i}(R \cup \approx_L^{MPS})\overrightarrow{V_{i+1}}$$

wobei $\overrightarrow{V_0} = \overrightarrow{C}$ und $\overrightarrow{V_{n+1}} = \overrightarrow{D}$. Nach Induktionshypothese gilt die 2. Bedingung für $\overrightarrow{C}\ Q\ \overrightarrow{V_{n+1}}$. Weiterhin gilt $\overrightarrow{V_{n+1}}\ R\ \overrightarrow{D}$ oder $\overrightarrow{V_{n+1}} \approx_L^{MPS} \overrightarrow{D}$. Sei π eine beliebige Verteilung und $s_1, s_2, s_3 \in MEM$ mit $s_1 =_L^{MPS} s_2 =_L^{MPS} s_3$. Weiterhin sei s_1' so, dass

$$\langle\langle C_1 \ldots C_i \ldots C_j \ldots C_n\rangle, \pi, s_1\rangle \rightarrowtriangle \langle\langle C_1 \ldots C_i' \ldots C_j' \ldots C_n\rangle, \pi, s_1'\rangle$$

Dann existiert s_2', $V_{n+1,i}'$ und $V_{n+1,j}'$ mit $s_1' =_L^{MPS} s_2'$ und

$$\begin{aligned}&\langle\langle V_{n+1,1} \ldots V_{n+1,i} \ldots V_{n+1,j} \ldots V_{n+1,n}\rangle, \pi, s_2\rangle\\ &\rightarrowtriangle\\ &\langle\langle V_{n+1,1} \ldots V_{n+1,i}' \ldots V_{n+1,j}' \ldots V_{n+1,n}\rangle, \pi, s_2'\rangle\end{aligned}$$

und $\langle C_i'\rangle\ Q\ \langle V_{n+1,i}'\rangle$ und $\langle C_j'\rangle\ Q\ \langle V_{n+1,j}'\rangle$. Für s_2 und s_2' existiert weiter s_3', D_i' und D_j' mit $s_3' =_L^{MPS} s_2'$ und

$$\langle\langle D_1 \ldots D_i \ldots D_j \ldots D_n\rangle, \pi, s_3\rangle \rightarrowtriangle \langle\langle D_1 \ldots D_i' \ldots D_j' \ldots D_n\rangle, \pi, s_3'\rangle$$

und $\langle V'_{n+1,i}\rangle\ Q\ \langle D'_i\rangle$ und $\langle V'_{n+1,j}\rangle\ Q\ \langle D'_j\rangle$ (wieder da $\approx_L^{MPS} \subset Q$). Da Q transitiv ist, folgt $C'_i\ Q\ D'_i$ und $C'_i\ Q\ D'_i$, sowie $s'_1 =_L^{MPS} s'_3$. Damit ist Q eine starke Low-MPS-Bisimulation auf Programmen und $Q \subset \approx_L^{MPS}$. □

Eine erste Anwendung des obigen Satzes liefert, dass sichere Threadpools aus sicheren Threads bestehen. Die Umkehrung wird später bewiesen.

Lemma 5.14. Seien $\langle C_1 \dots C_n\rangle, \langle D_1 \dots D_n\rangle \in \overrightarrow{Com_{MTL-L}}$.
Wenn $\langle C_1 \dots C_n\rangle \approx_L^{MPS} \langle D_1 \dots D_n\rangle$, dann gilt $\forall i \leq n(\langle C_i\rangle \approx_L^{MPS} \langle D_i\rangle)$

Beweis. Sei $\langle C_1 \dots C_n\rangle \approx_L^{MPS} \langle D_1 \dots D_n\rangle$ und $i \leq n$.
Sei $R = \{(\langle C_i\rangle, \langle D_i\rangle)\}$.
Sei π eine beliebige Verteilung mit $\pi(1) = p$. Weiter seien s_1,s'_1 und s_2 Speicher mit $s_1 =_L^{MPS} s_2$ und

$$\langle\langle C_i\rangle, \pi, s_1\rangle \rightarrowtriangle \langle\langle C'_i\rangle, \pi, s'_1\rangle$$

Nach Regel **Non-Par** gilt für gleiches s_1,s'_1 und C'_i mit

$$\langle\langle C_1 \dots C_i \dots C_n\rangle, \pi, s_1\rangle \rightarrowtriangle \langle\langle C_1 \dots C'_i \dots C_n\rangle, \pi, s'_1\rangle$$

dass ein s'_2 und D'_i existiert mit

$$\langle\langle D_1 \dots D_i \dots D_n\rangle, \pi, s_2\rangle \rightarrowtriangle \langle\langle D_1 \dots D'_i \dots D_n\rangle, \pi, s'_2\rangle$$

wobei $s'_1 =_L^{MPS} s'_2$ und $\langle C'_i\rangle \approx_L^{MPS} \langle D'_i\rangle$.
Aus der gleichen Regel folgt, dass für gleiches s'_2 und D'_i

$$\langle\langle D_i\rangle, \pi, s_2\rangle \rightarrowtriangle \langle\langle D'_i\rangle, \pi, s'_2\rangle$$

Damit ist R eine starke Low-MPS-Bisimulation auf Programmen bis zu $\approx_L^{MPS}$ und $\langle C_i\rangle \approx_L^{MPS} \langle D_i\rangle$. □

Die Sicherheitseigenschaft erfüllt zahlreiche Kompositionalitätsresultate, die zeigen, dass man sichere Programme aus sicheren Teilprogrammen zusammensetzen kann. Dadurch lässt sich ähnlich wie für *strong security* ein Typsystem angeben.

Satz 5.15. Sei $C_1 \approx_L^{MPS} D_1$ und $C_2 \approx_L^{MPS} D_2$. Dann gilt

1. $C_1; C_2 \approx_L^{MPS} D_1; D_2$
2. $B \equiv_L B' \vee C_1 \approx_L^{MPS} C_2 \implies$
   if B **then** C_1 **else** C_2 **fi** $\approx_L^{MPS}$ **if** B' **then** D_1 **else** D_2 **fi**

3. $B \equiv_L B' \implies \textbf{while } B \textbf{ do } C_1 \textbf{ od} \approx_L^{MPS} \textbf{while } B' \textbf{ do } D_1 \textbf{ od}$

Beweis. 1. Sei

$$R = \{(\langle C_1; C_2\rangle, \langle D_1; D_2\rangle \mid C_1 \approx_L^{MPS} D_1, C_2 \approx_L^{MPS} D_2\}$$

Es wird nun gezeigt, dass R eine starke Low-MPS-Bisimulation auf Programmen bis zu $\approx_L^{MPS}$ ist. Aus der Definition von R folgt, dass R symmetrisch ist und Threadpools gleicher Größe in Relation setzt.

Sei $L\ R\ L'$. Dann ist $L = \langle C_1; C_2\rangle$ und $L' = \langle D_1; D_2\rangle$ mit $C_1 \approx_L^{MPS} D_1$ und $C_2 \approx_L^{MPS} D_2$.

Sei π eine beliebige Verteilung mit $\pi(1) = p$. Weiter seien s_1,s_2 und s_1' Speicher mit $s_1 =_L^{MPS} s_2$ so, dass

$$\langle\langle C_1; C_2\rangle, \pi, s_1\rangle \rightarrowtriangle \langle E, \pi, s_1'\rangle$$

Man kann nach Beobachtung 5.8 annehmen, dass die Speicher nach der Anwendung der ε-Kette wieder s_1 und s_2 sind.

Nach den Regeln der operationellen Semantik gilt entweder $E = \langle C_2\rangle$ oder $E = \langle C_3; C_2\rangle$.

Angenommen $E = \langle C_2\rangle$. Daraus folgt, dass C_1 in einem Schritt terminiert. Aus $C_1 \approx_L^{MPS} D_1$ folgt für s_1,s_2 und s_1', dass s_2' existiert mit

$$\langle\langle D_1\rangle, \pi, s_2\rangle \rightarrowtriangle \langle\langle\rangle, \pi, s_2'\rangle$$

mit $s_2' =_L^{MPS} s_1'$. Somit gilt

$$\langle\langle D_1; D_2\rangle, \pi, s_2\rangle \rightarrowtriangle \langle\langle D_2\rangle, \pi, s_2'\rangle$$

wobei $C_2 \approx_L^{MPS} D_2$ nach Annahme.

Angenommen $E = \langle C_3; C_2\rangle$. Dann folgt, dass C_1 nicht in einem Schritt terminiert und damit ebenso D_1. Aus $C_1 \approx_L^{MPS} D_1$ folgt für s_1,s_2 und s_1', dass s_2' und D_3 existiert mit

$$\langle\langle D_1\rangle, \pi, s_2\rangle \rightarrowtriangle \langle\langle D_3\rangle, \pi, s_2'\rangle$$

mit $s_2' =_L^{MPS} s_1'$ und $C_3 \approx_L^{MPS} D_3$. Somit gilt nach Regel **Non-Par**

$$\langle\langle D_1; D_2\rangle, \pi, s_2\rangle \rightarrowtriangle \langle\langle D_3; D_2\rangle, \pi, s_2'\rangle$$

Da $C_3 \approx_L^{MPS} D_3$ und $C_2 \approx_L^{MPS} D_2$ ist $\langle C_3; C_2\rangle\ R\ \langle D_3; D_2\rangle$.

Damit ist R eine starke Low-MPS-Bisimulation auf Programmen bis zu $\approx_L^{MPS}$.

2. Sei

$$R = \{(\langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}\rangle, \langle \textbf{if } B' \textbf{ then } D_1 \textbf{ else } D_2 \textbf{ fi}\rangle) \mid B \equiv_L B' \vee C_1 \approx_L^{MPS} C_2\}$$

Es wird nun gezeigt, dass R eine starke Low-MPS-Bisimulation auf Programmen bis zu $\approx_L^{MPS}$ ist. Aus der Definition von R folgt, dass R symmetrisch ist und Threadpools gleicher Größe in Relation setzt.

Sei $L\ R\ L'$. Dann ist $L = \langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}\rangle$ und $L' = \langle \textbf{if } B' \textbf{ then } D_1 \textbf{ else } D_2 \textbf{ fi}\rangle$ mit $B \equiv_L B'$ oder $C_1 \approx_L^{MPS} C_2$.

Sei π eine beliebige Verteilung mit $\pi(1) = p$. Weiter seien s_1,s_2 und s_1' Speicher mit $s_1 =_L^{MPS} s_2$ so, dass

$$\langle\langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}\rangle, \pi, s_1\rangle \rightarrowtriangle \langle D, \pi, s_1'\rangle$$

Nach Beobachtung 5.8 wird der ε-Schritt ausgelassen.

Ähnlich zu **skip** hat die Ausführung von L und L' keine lokalen und globalen Auswirkungen, sodass $s_1' =_L^{MPS} s_2'$.

Angenommen $B \equiv_L B'$. Dann evaluieren beide Ausdrücke unter s_1' und s_2' zu den selben Werten.

Im Fall, dass B zu $True$ evaluiert, machen beide Zustände einen Schritt, sodass

$$\langle\langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}\rangle, \pi, s_1\rangle \rightarrowtriangle \langle\langle C_1\rangle, \pi, s_1'\rangle$$
$$\langle\langle \textbf{if } B' \textbf{ then } D_1 \textbf{ else } D_2 \textbf{ fi}\rangle, \pi, s_2\rangle \rightarrowtriangle \langle\langle D_1\rangle, \pi, s_2'\rangle$$

Nach Annahme gilt $\langle C_1\rangle \approx_L^{MPS} \langle D_1\rangle$.

Im Fall, dass B zu $False$ evaluiert, machen beide Zustände einen Schritt, sodass $\langle C_2\rangle$ und $\langle D_2\rangle$ als noch auszuführende Programme verbleiben. Diese stehen ebenfalls nach Annahme in $\approx_L^{MPS}$ in Relation.

Angenommen $C_1 \approx_L^{MPS} C_2$. Dann gilt ebenso $D_1 \approx_L^{MPS} D_2$. Demnach stehen die noch verbleibenden Programme nach Ausführung von L und L' unabhängig vom Ergebnis der Evaluation von B und B' wieder in $\approx_L^{MPS}$ in Relation.

Damit ist R eine starke Low-MPS-Bisimulation auf Programmen bis zu $\approx_L^{MPS}$ und nach Satz 5.13 die Behauptung bewiesen.

3. Sei

$$R = \{(\langle \textbf{while } B \textbf{ do } C_1 \textbf{ od}\rangle, \langle \textbf{while } B' \textbf{ do } C_2 \textbf{ od}\rangle) \mid B \equiv_L B'\} \cup$$
$$\{(\langle C_1; \textbf{while } B \textbf{ do } C_1 \textbf{ od}\rangle, \langle C_2; \textbf{while } B' \textbf{ do } C_2 \textbf{ od}\rangle) \mid B \equiv_L B'\}$$

Es wird nun gezeigt, dass R eine starke Low-MPS-Bisimulation auf Programmen bis zu $\approx_L^{MPS}$. Aus der Definition von R folgt, dass R symmetrisch ist und Threadpools gleicher Größe in Relation setzt.

Sei $L\ R\ L'$. Nach Definition von R gibt es zwei Möglichkeiten für L und L'.

Angenommen $L = \langle \textbf{while } B \textbf{ do } C_1 \textbf{ od} \rangle$ und $L' = \langle \textbf{while } B' \textbf{ do } C_2 \textbf{ od} \rangle$ mit $B \equiv_L B'$.

Sei π eine beliebige Verteilung mit $\pi(1) = p$. Weiter seien s_1,s_2 und s_1' Speicher mit $s_1 =_L^{MPS} s_2$ so, dass

$$\langle \langle \textbf{while } B \textbf{ do } C_1 \textbf{ od} \rangle, \pi, s_1 \rangle \twoheadrightarrow \langle D, \pi, s_1' \rangle$$

Nach Beobachtung 5.8 wird der ε-Schritt ausgelassen.

Man unterscheidet nun zwei Fälle:

(1) $eval_p(B, s_1) = False$: In diesem Fall ist $D = \langle\rangle$. Da $B \equiv_L B'$, gilt $eval_p(B', s_2) = False$ und somit

$$\langle \langle \textbf{while } B' \textbf{ do } C_2 \textbf{ od} \rangle, \pi, s_2 \rangle \twoheadrightarrow \langle \langle\rangle, \pi, s_2' \rangle$$

Schließlich ist $\langle\rangle \approx_L^{MPS} \langle\rangle$.

(2) $eval_p(B, s_1) = True$: Hierbei ist

$$D = \langle C_1; \textbf{while } B \textbf{ do } C_1 \textbf{ od} \rangle$$

Aus $B \equiv_L B'$ folgt $eval_p(B', s_2) = True$ und

$$\langle \langle \textbf{while } B' \textbf{ do } C_2 \textbf{ od} \rangle, \pi, s_2 \rangle \twoheadrightarrow \langle \langle C_2; \textbf{while } B' \textbf{ do } C_2 \textbf{ od} \rangle, \pi, s_2' \rangle$$

Da $C_1 \approx_L^{MPS} C_2$, gilt

$$\langle C_1; \textbf{while } B \textbf{ do } C_1 \textbf{ od} \rangle\ R\ \langle C_2; \textbf{while } B' \textbf{ do } C_2 \textbf{ od} \rangle$$

Die zweite Möglichkeit für L und L' ist, dass
$L = \langle C_1; \textbf{while } B \textbf{ do } C_1 \textbf{ od} \rangle$ und $L' = \langle C_2; \textbf{while } B \textbf{ do } C_2 \textbf{ od} \rangle$.

Sei wieder π eine beliebige Verteilung mit $\pi(1) = p$. Weiter seien s_1,s_2 und s_1' Speicher mit $s_1 =_L^{MPS} s_2$ so, dass

$$\langle \langle C_1; \textbf{while } B \textbf{ do } C_1 \textbf{ od} \rangle, \pi, s_1 \rangle \twoheadrightarrow \langle D, \pi, s_1' \rangle$$

Nach Beobachtung 5.8 wird der ε-Schritt ausgelassen.

Man unterscheidet auch hier zwei Fälle:

(1) $D = \langle \textbf{while } B \textbf{ do } C_1 \textbf{ od} \rangle$: Damit terminiert die Ausführung von C_1 und C_2 nach einem Schritt. Aus $C_1 \approx_L^{MPS} C_2$ folgt für s_1, s_2 und s_1' mit

$$\langle\langle C_1 \rangle, \pi, s_1\rangle \rightarrowtriangle \langle\langle\rangle, \pi, s_1'\rangle$$

die Existenz von s_2', sodass

$$\langle\langle C_2 \rangle, \pi, s_2\rangle \rightarrowtriangle \langle\langle\rangle, \pi, s_2'\rangle$$

Damit gilt für gleiches s_2' nach den Regeln **Non-Par** und **While**, dass

$$\langle\langle C_2; \textbf{while } B' \textbf{ do } C_2 \textbf{ od} \rangle, \pi, s_2\rangle \rightarrowtriangle \langle\langle \textbf{while } B' \textbf{ do } C_2 \textbf{ od} \rangle, \pi, s_2'\rangle$$

wobei

$$\langle \textbf{while } B \textbf{ do } C_1 \textbf{ od} \rangle \, R \, \langle \textbf{while } B' \textbf{ do } C_2 \textbf{ od} \rangle$$

(2) $D = \langle C_1'; \textbf{while } B \textbf{ do } C_1 \textbf{ od} \rangle$: Da $C_1 \approx_L^{MPS} C_2$ terminieren C_1 und C_2 nicht in einem Schritt. Weiterhin folgt für s_1, s_2 und s_1' mit

$$\langle\langle C_1 \rangle, \pi, s_1\rangle \rightarrowtriangle \langle\langle C_1' \rangle, \pi, s_1'\rangle$$

die Existenz von s_2', sodass

$$\langle\langle C_2 \rangle, \pi, s_2\rangle \rightarrowtriangle \langle\langle C_2' \rangle, \pi, s_2'\rangle$$

und $C_1' \approx_L^{MPS} C_2'$. Damit gilt für gleiches s_2' nach Regel **Non-Par** und **While**, dass

$$\langle\langle C_2; \textbf{while } B' \textbf{ do } C_2 \textbf{ od} \rangle, \pi, s_2\rangle \rightarrowtriangle \langle\langle C_2'; \textbf{while } B' \textbf{ do } C_2 \textbf{ od} \rangle, \pi, s_2'\rangle$$

Da $C_1' \approx_L^{MPS} C_2'$ folgt somit

$$\langle C_1'; \textbf{while } B \textbf{ do } C_1 \textbf{ od} \rangle \, R \, \langle C_2'; \textbf{while } B' \textbf{ do } C_2 \textbf{ od} \rangle$$

Es folgt, dass R eine starke Low-MPS-Bisimulation auf Programmen bis zu $\approx_L^{MPS}$ ist und damit $\textbf{while } B \textbf{ do } C_1 \textbf{ od} \approx_L^{MPS} \textbf{while } B' \textbf{ do } C_2 \textbf{ od}$. □

Das letzte Kompositionalitätsresultat ist die Umkehrung von Lemma 5.14. Das bedeutet, dass sich sichere Multithread-Programme aus sicheren Threads zusammenstellen lassen. Der entscheidende Punkt des Beweises ist zu zeigen, dass die parallele Ausführung zweier sicherer Threads die 2. Bedingung der starken Low-MPS-Bisimulation erfüllen.

Hierzu wird beschrieben, wie sich der Entscheidungsraum eines Speichers s^{VI} bei der Komposition der Zwischenspeicher s^{III} und s^{V} der Regel **Par** aus deren Entscheidungsräumen zusammensetzt. Sei s der anfängliche Speicher und s^{III} und s^{V} jeweils die Speicher, die durch die globalen und lokalen Auswirkungen der Ausführung eines Threads auf Prozessor 0 und 1 resultieren. Für s^{VI} gilt

$$s^{VI} = composeMemories(s, s^{III}, s^{V})$$

Seien (E_0, E_1, Λ, T),$(E_0^{III}, E_1^{III}, \Lambda^{III}, T^{III})$ und $(E_0^{V}, E_1^{V}, \Lambda^{V}, T^{V})$ die Entscheidungsräume von s, s^{III} und s^{V}. Der Entscheidungsraum kann sich nur durch die Ausführung einer Acquire- oder Releaseanweisung oder einer Zuweisung in eine low-Variable verändert werden. Zudem kann Prozessor p nicht die Menge E_{1-p} verändern. Daher gilt $E_1^{III} = E_1$ und $E_0^{V} = E_0$. Weiterhin gilt im Fall einer Änderung des Entscheidungsraums entweder $E_0^{III} = E_0 \cup \{j_1\}$ oder $\Lambda^{III} = \Lambda \cup \{j_1\}$. Ebenso gilt entweder $E_1^{V} = E_1 \cup \{j_2\}$ oder $\Lambda^{V} = \Lambda \cup \{j_2\}$.

Führte mindestens einer der Prozessoren eine Acquire- oder Releaseanweisung oder einer Zuweisung in eine low-Variable durch, so enthalten die Ordnungen T^{III} und T^{V} im Gegensatz zu T neue Relationen (k, t). Hierbei gilt stets, dass $t = j_1$ oder $t = j_2$.

Der Entscheidungsraum für s^{VI} setzt sich daher, wie folgt, zusammen

$$(E_0^{III}, E_1^{V}, \Lambda^{III} \cup \Lambda^{V}, T^{III} \cup T^{V})$$

Hierzu wird zuerst folgendes Lemma bewiesen.

Lemma 5.16. Sei $\langle C_1 \rangle \approx_L^{MPS} \langle D_1 \rangle$ und $\langle C_2 \rangle \approx_L^{MPS} \langle D_2 \rangle$. Weiterhin sei π eine Verteilung und $s_1, s_2, s_1' \in MEM$ mit $s_1 =_L^{MPS} s_2$ so, dass

$$\langle\langle C_1 C_2 \rangle, \pi, s_1 \rangle \twoheadrightarrow \langle\langle C_1' C_2' \rangle, \pi, s_1' \rangle$$

Dann existiert $s_2' \in MEM$ und $D_1', D_2' \in Com_{MTL\text{-}L}$, sodass

$$\langle\langle D_1 D_2 \rangle, \pi, s_2 \rangle \twoheadrightarrow \langle\langle D_1' D_2' \rangle, \pi, s_2' \rangle$$

mit $s_1' =_L^{MPS} s_2'$.

Beweis. Nach Beobachtung 5.8 wird der ε-Schritt ausgelassen. O.B.d.A sei $\pi(1) = 0$ und $\pi(2) = 1$. Weiter seien ξ und ρ Verteilungen mit $\xi(1) = 0$ und $\rho(1) = 2$. Schließlich sei s_1,$s_2 \in MEM$ mit $s_1 =_L^{MPS} s_2$.

Aus $\langle C_1 \rangle \approx_L^{MPS} \langle D_1 \rangle$ folgt für ξ, s_1 und s_1° mit

$$\langle\langle C_1 \rangle, \xi, s_1 \rangle \twoheadrightarrow \langle\langle C_1' \rangle, \xi, s_1^{\circ} \rangle$$

die Existenz von s_2° mit

$$\langle\langle D_1 \rangle, \xi, s_2 \rangle \twoheadrightarrow \langle\langle D_1' \rangle, \xi, s_2^{\circ} \rangle$$

und $s_1^\circ =_L^{MPS} s_2^\circ$. Ebenso folgt aus $\langle C_2 \rangle \approx_L^{MPS} \langle D_2 \rangle$ für ρ, s_1 und s_1^+ mit

$$\langle\langle C_2 \rangle, \rho, s_1 \rangle \rightarrowtriangle \langle\langle C_2' \rangle, \rho, s_1^+ \rangle$$

die Existenz von s_2^+ mit

$$\langle\langle D_2 \rangle, \rho, s_2 \rangle \rightarrowtriangle \langle\langle D_2' \rangle, \rho, s_2^+ \rangle$$

und $s_1^+ =_L^{MPS} s_2^+$.

Sei s_1' so, dass

$$\langle\langle C_1 C_2 \rangle, \pi, s_1 \rangle \rightarrowtriangle \langle\langle C_1' C_2' \rangle, \pi, s_1' \rangle$$

Dann gilt $s_1' = composeMemories(s_1, s_1^\circ, s_1^+)$. Weiterhin gilt für

$$\langle\langle D_1 D_2 \rangle, \pi, s_2 \rangle \rightarrowtriangle \langle\langle D_1' D_2' \rangle, \pi, s_2' \rangle$$

dass $s_2' = composeMemories(s_2, s_2^\circ, s_2^+)$. Es wird nun argumentiert, dass die Entscheidungsräume von s_1' und s_2' ununterscheidbar sind. Hierzu wird für jede Komponente des Entscheidungsraums argumentiert. Sei φ eine injektive Funktion, sodass die Entscheidungsräume von s_1 und s_2 mittels φ ununterscheidbar sind.

E_0: Man unterscheidet zwei Fälle. Im ersten Fall ist die Menge E_0 in s_1' und in s_1 gleich. Dann wurde keine Zuweisung durch C_1 in eine low-Variable x getätigt. Aus $C_1 \approx_L^{MPS} D_1$ und Korollar 5.11 folgt, dass auch D_1 keine Zuweisung in eine low-Variable tätigte. Somit ist auch hier E_0 in s_2 und s_2' gleich. Im zweite Fall enthält die Menge E_0 in s_1' einen Index j mehr als in s_1. Die zugehörige Schreiboperation sei $(0, x, n)$. Die Menge E_0 in s_2' enthält ebenfalls nach $C_1 \approx_L^{MPS} D_1$ und Korollar 5.11 einen Index k mehr als in s_2 mit gleicher Schreiboperation. Somit lässt sich φ erweitern zu $\varphi(j) = k$.

E_1: Analog zu E_0.

Λ: Man unterscheidet hier die Fälle, dass sich die Menge von s_1 zu s_1' nicht verändert hat oder bis zu zwei neue Indizes hinzugekommen sind. Angenommen die Menge enthält zusätzlich einen Index j, die auf eine Schreiboperation $(0, l, 1)$ mit $l \in Lock$ verweist. Dann ist $C_{1,0} = l.acquire$. Aus $C_1 \approx_L^{MPS} D_1$ und Korollar 5.11 folgt, dass Λ in s_2' ebenfalls ein k zusätzlich enthält, der auf die gleiche Schreiboperation verweist. Damit lässt sich φ erweitern zu $\varphi(j) = k$. Für den Fall, dass zwei Indizes hinzugekommen sind, oder Release-Operationen getätigt wurden, folgt die Erweiterung von φ analog.

T: Auch hier unterscheidet man die Fälle, dass sich keine Änderung von T von s_1 zu s_1' ergeben hat oder dass neue Paare (t_1, t_2) hinzugefügt wurden, wobei t_2 einer der neuen Indizes aus E_0,E_1 oder Λ ist. Angenommen E_0 hat einen Index j mehr. So existiert auch ein k in E_0 in s_2' mit $\varphi(j) = k$. und es wurde eine Zuweisung getätigt. Sei L die Lockordnung in s_1'. Dann

ist j größer nach L in s_1' als die letzte getätigte Acquire-Operation m von Prozessor 0. T in s_1' enthält also die Paare (m, j) und (m', j) für $m' \in E \cup \Lambda$ und $m'\ L\ m$. Da $s_1 =_L^{MPS} s_2$, existieren für m und alle m' geeignete Indizes in s_2, sodass ebenso für s_2' gilt, dass genau die Paare $(\varphi(m), \varphi(j))$ und $(\varphi(m'), \varphi(j))$ in T in s_2' zusätzlich enthalten sind. Für den Fall, dass Acquire- oder Releaseanweisungen getätigt wurden, folgt die Argumentation analog.

Es verbleibt zu zeigen, dass die Werte der low-Variablen und die Lockzustände von s_1' und s_2' gleich sind.

Angenommen ls in s_1' unterscheidet sich zu ls in s_1. Dann folgt, wie oben, dass die gleichen Synchronisationanweisungen in beiden Systemen ausgeführt wurden und so die gleichen Änderungen an ls in s_2' zu sehen sind. Damit sind die Lockzustände gleich.

Eine Änderung der Werte der low-Variablen erfolgt nur durch die Zuweisung eines Werts in diese. Wie oben erwähnt, geschieht dies in beiden Systemen auf gleiche Weise, d.h. für eine low-Variable x wird entweder in beiden Systemen der Wert zu n geändert oder er bleibt gleich. Somit sind auch die Werte der low-Variablen in s_1' und s_2' gleich.

Es gilt demnach $s_1' =_L^{MPS} s_2'$. □

Es folgt nun der Beweis für die Komposition eines Threadpools und einem Thread.

Satz 5.17. Sei $\overrightarrow{C} \approx_L^{MPS} \overrightarrow{D}$ und $\langle E \rangle \approx_L^{MPS} \langle F \rangle$. Dann gilt $\langle C_1 \dots C_n E \rangle \approx_L^{MPS} \langle D_1 \dots D_n F \rangle$.

Beweis. Sei

$$R = \{(\langle C_1 \dots C_n E \rangle, \langle D_1 \dots D_n F \rangle) \mid \overrightarrow{C} \approx_L^{MPS} \overrightarrow{D}, \langle E \rangle \approx_L^{MPS} \langle F \rangle\}$$

Es ist nun zu zeigen, dass R eine starke Low-MPS-Bisimulation auf Programmen bis zu $\approx_L^{MPS}$ ist. Nach obiger Definition ist R symmetrisch und setzt Threadpools gleicher Größe in Relation.

Sei $L\ R\ L'$. Dann ist $L = \langle C_1 \dots C_n E \rangle$ und $L' = \langle D_1 \dots D_n F \rangle$ mit $\overrightarrow{C} \approx_L^{MPS} \overrightarrow{D}$ und $\langle E \rangle \approx_L^{MPS} \langle F \rangle$.

Sei π eine beliebige Verteilung und $s_1, s_2 \in MEM$ mit $s_1 =_L^{MPS} s_2$. Man unterscheidet nun vier Fälle:

1. es wird ein einzelner Thread von $\overrightarrow{C}$ ausgeführt
2. es wird E einzeln ausgeführt
3. es werden zwei Threads von $\overrightarrow{C}$ parallel ausgeführt

4. es wird ein Thread von $\overrightarrow{C}$ ein E parallel ausgeführt

(1): Sei s_1' so, dass

$$\langle\langle C_1 \dots C_i \dots C_n E\rangle, \pi, s_1\rangle \rightarrow \langle\langle C_1 \dots C_i' \dots C_n E\rangle, \pi, s_1'\rangle$$

Aus $\overrightarrow{C} \approx_L^{MPS} \overrightarrow{D}$ folgt für s_1, s_2 und s_1' mit

$$\langle\langle C_1 \dots C_i \dots C_n\rangle, \pi, s_1\rangle \rightarrow \langle\langle C_1 \dots C_i' \dots C_n\rangle, \pi, s_1'\rangle$$

die Existenz von s_2' und D_i', sodass

$$\langle\langle D_1 \dots D_i \dots D_n\rangle, \pi, s_2\rangle \rightarrow \langle\langle D_1 \dots D_i' \dots D_n\rangle, \pi, s_2'\rangle$$

und $s_1' =_L^{MPS} s_2'$ und $\langle C_i'\rangle \approx_L^{MPS} \langle D_i'\rangle$. Nach den Regeln der operationellen Semantik gilt dann auch

$$\langle\langle D_1 \dots D_i \dots D_n F\rangle, \pi, s_2\rangle \rightarrow \langle\langle D_1 \dots D_i' \dots D_n F\rangle, \pi, s_2'\rangle$$

(2): Sei s_1' so, dass

$$\langle\langle C_1 \dots C_i \dots C_n E\rangle, \pi, s_1\rangle \rightarrow \langle\langle C_1 \dots C_i \dots C_n E'\rangle, \pi, s_1'\rangle$$

Aus $\langle E\rangle \approx_L^{MPS} \langle F\rangle$ folgt für s_1, s_2 und s_1' mit

$$\langle\langle E\rangle, \pi, s_1\rangle \rightarrow \langle\langle E'\rangle, \pi, s_1'\rangle$$

die Existenz von s_2' und F', sodass

$$\langle\langle F\rangle, \pi, s_2\rangle \rightarrow \langle\langle F'\rangle, \pi, s_2'\rangle$$

und $s_1' =_L^{MPS} s_2'$ und $\langle E'\rangle \approx_L^{MPS} \langle F'\rangle$. Nach den Regeln der operationellen Semantik gilt dann auch

$$\langle\langle D_1 \dots D_i \dots D_n F\rangle, \pi, s_2\rangle \rightarrow \langle\langle D_1 \dots D_i \dots D_n F'\rangle, \pi, s_2'\rangle$$

(3): Sei s_1' so, dass

$$\langle\langle C_1 \dots C_i \dots C_j \dots C_n E\rangle, \pi, s_1\rangle \rightarrow \langle\langle C_1 \dots C_i' \dots C_j' \dots C_n E\rangle, \pi, s_1'\rangle$$

Aus $\overrightarrow{C} \approx_L^{MPS} \overrightarrow{D}$ folgt für s_1, s_2 und s_1' mit

$$\langle\langle C_1 \dots C_i \dots C_j \dots C_n\rangle, \pi, s_1\rangle \rightarrow \langle\langle C_1 \dots C_i' \dots C_j' \dots C_n\rangle, \pi, s_1'\rangle$$

die Existenz von s_2', D_i' und D_j', sodass

$$\langle\langle D_1 \dots D_i \dots D_j \dots D_n\rangle, \pi, s_2\rangle \twoheadrightarrow \langle\langle D_1 \dots D_i' \dots D_j' \dots D_n\rangle, \pi, s_2'\rangle$$

und $s_1' =_L^{MPS} s_2'$, $\langle C_i'\rangle \approx_L^{MPS} \langle D_i'\rangle$ und $\langle C_j'\rangle \approx_L^{MPS} \langle D_j'\rangle$. Nach den Regeln der operationellen Semantik gilt dann auch

$$\langle\langle D_1 \dots D_i \dots D_j \dots D_n F\rangle, \pi, s_2\rangle \twoheadrightarrow \langle\langle D_1 \dots D_i' \dots D_j' \dots D_n F\rangle, \pi, s_2'\rangle$$

(4): Sei s_1' so, dass

$$\langle\langle C_1 \dots C_i \dots C_n E\rangle, \pi, s_1\rangle \twoheadrightarrow \langle\langle C_1 \dots C_i' \dots C_n E'\rangle, \pi, s_1'\rangle$$

Es gilt $\langle C_i\rangle \approx_L^{MPS} \langle D_i\rangle$ nach Lemma 5.14 und $\langle E\rangle \approx_L^{MPS} \langle F\rangle$. Nach Lemma 5.16 gilt dann für s_1,s_2 und s_1', dass ein s_2' existiert, sodass

$$\langle\langle D_i F\rangle, \pi, s_1\rangle \twoheadrightarrow \langle\langle D_i' F'\rangle, \pi, s_1'\rangle$$

und $s_1' =_L^{MPS} s_2'$. Damit folgt nach den Regeln der Semantik, dass auch

$$\langle\langle D_1 \dots D_i \dots D_n F\rangle, \pi, s_2\rangle \twoheadrightarrow \langle\langle D_1 \dots D_i' \dots D_n F'\rangle, \pi, s_2'\rangle$$

Somit gilt in allen vier Fällen, dass die 2. Bedingung erfüllt ist und R daher eine starke Low-MPS-Bisimulation auf Programmen bis zu $\approx_L^{MPS}$ ist. □

Korollar 5.18. Sei $\overrightarrow{C}_1 \approx_L^{MPS} \overrightarrow{D}_1$ und $\overrightarrow{C}_2 \approx_L^{MPS} \overrightarrow{D}_2$. Dann gilt $\overrightarrow{C}_1\overrightarrow{C}_2 \approx_L^{MPS} \overrightarrow{D}_1\overrightarrow{D}_2$.

Beweis. Folgt aus Lemma 5.14 und der mehrfachen Anwendung des vorherigen Satzes. □

5.3 Typsystem

Es wird nun ein Typsystem für die obige Sicherheitseigenschaft vorgestellt. Damit hat es die Eigenschaft, dass ein typisierbares Programm stark Low-MPS sicher ist.

Für Ausdrücke, die in beiden Sprachen MTL und MTL-L gleich sind, wird das gleiche Typsystem verwendet. Zur besseren Übersicht ist es hier erneut in Abbildung 5.5 angeführt. Um die obigen Sätze für Bedingungen und Schleifen verwenden zu können, wird folgendes Lemma angegeben.

Lemma 5.19. Sei $B \in \mathit{Expr}$. Wenn $\vdash B : \mathit{low}$ nach dem Typsystem in Abbildung 5.5, dann gilt $B \equiv_L B$.

Beweis. Der Beweis erfolgt mittels Induktion über die Typregeln des Typsystems.

$$\text{Val}\frac{}{\vdash val : low} \qquad \text{Var}\frac{lvl(x) = D}{\vdash x : D}$$

$$\text{Op}\frac{\vdash expr_1 : D_1 \ldots \vdash expr_n : D_n \qquad \forall i \quad D_i \leq D}{\vdash op(expr_1, \ldots, expr_n) : D}$$

Abbildung 5.5: Typsystem für Ausdrücke

Val: Angenommen $B = v$, wobei $v \in Val$. Dann ist die Evaluation des Ausdrucks unabhängig von einem Speicher und somit $B \equiv_L B$.

Var: Angenommen $B = x$, wobei $x \in Var$ und $lvl(x) = low$. Seien $s, s' \in MEM$ mit $s \equiv_L s'$. Dann sind insbesondere die Werte von x in beiden Speichern für jeweils einen Prozessor gleich. Daher gilt $B \equiv_L B$.

Op: Angenommen $B = op(expr_1, \ldots, expr_n)$ mit $\vdash expr_i : low$. Dann gilt $expr_i \equiv_L expr_i$. Seien $s, s' \in MEM$ mit $s \equiv_L s'$. Es gilt $eval_p(expr_i, s) = eval_p(expr_i, s')$. Somit gilt ebenso $eval_p(B, s) = eval_p(B, s')$ und damit $B \equiv_L B$. □

Den Typregeln in Abbildung 5.6 liegt die gleiche Intuition zugrunde wie in Kapitel 2. Der Unterschied zum Typsystem dort ist jedoch die fehlende Regel für **fork**. Dafür existiert eine Regel für die parallele Komposition, da das System mit mehreren Thread gestartet wird. Zudem existieren Regeln für l.acquire und l.release. Ähnlich zu **skip**, sind diese immer typisierbar. Ebenso wie im Typsystem für *strong security* hat die Regel für Bedingungen mit Ausdrücken, die als high typisiert sind, eine semantische Nebenbedingung. Hierzu ließe sich ebenfalls eine syntaktische Approximation finden, die hier aber nicht angegeben wird.

Satz 5.20 (Korrektheit des Typsystems)**.** Wenn das obige Typsystem $\vdash \overrightarrow{C}$ herleitet, dann ist $\overrightarrow{C}$ stark MPS-sicher.

Beweis. Der Satz wird mittels Induktion über die Typregeln des Typsystems bewiesen.

Skip: Nach Lemma 5.9 gilt $\langle \textbf{skip} \rangle \approx_L^{MPS} \langle \textbf{skip} \rangle$.

Acq: Nach Lemma 5.10 gilt $\langle l.acquire \rangle \approx_L^{MPS} \langle l.acquire \rangle$.

$$\text{Skip}\frac{}{\vdash \textbf{skip}} \qquad \text{Seq}\frac{\vdash C_1 \quad \vdash C_2}{\vdash C_1; C_2} \qquad \text{Par}\frac{\vdash C_1 \ldots \vdash C_n}{\vdash \langle C_1 \ldots C_n \rangle}$$

$$\text{While}\frac{\vdash B : low \quad \vdash C}{\vdash \textbf{while } B \textbf{ do } C \textbf{ od}} \qquad \text{Ass-L}\frac{\vdash expr : low \quad \vdash x : low}{\vdash x := expr}$$

$$\text{Ass-H}\frac{\vdash x : high}{\vdash x := expr} \qquad \text{Acq}\frac{}{\vdash l.acquire} \qquad \text{Rel}\frac{}{\vdash l.release}$$

$$\text{If-L}\frac{\vdash B : low \quad \vdash C_1 \quad \vdash C_2}{\vdash \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}}$$

$$\text{If-H}\frac{\vdash B : high \quad \vdash C_1 \quad \vdash C_2 \quad \langle C_1 \rangle \approx_L^{MPS} \langle C_2 \rangle}{\vdash \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}}$$

Abbildung 5.6: Typsystem

Rel: Nach Lemma 5.10 gilt $\langle l.release \rangle \approx_L^{MPS} \langle l.release \rangle$.

Ass-L: Aus $\vdash expr : low$ folgt $expr \equiv_L expr$. Daher gilt nach Lemma 5.10 $\langle x := expr \rangle \approx_L^{MPS} \langle x := expr \rangle$.

Ass-H: Aus Lemma 5.9 folgt unmittelbar, dass $\langle x := expr \rangle \approx_L^{MPS} \langle x := expr \rangle$.

Seq: Nach Induktionsannahme gilt, dass $\langle C_1 \rangle \approx_L^{MPS} \langle C_1 \rangle$ und $\langle C_2 \rangle \approx_L^{MPS} \langle C_2 \rangle$. Nach Satz 5.15 gilt somit $\langle C_1; C_2 \rangle \approx_L^{MPS} \langle C_1; C_2 \rangle$.

Par: Nach Induktionsannahme gilt, dass $\langle C_i \rangle \approx_L^{MPS} \langle C_i \rangle$ mit $i \in \{1, \ldots, n\}$. Die mehrfache Anwendung von Satz 5.17 liefert damit $\langle C_1 \ldots C_n \rangle \approx_L^{MPS} \langle C_1 \ldots C_n \rangle$.

While: Nach Induktionsannahme gilt $\langle C \rangle \approx_L^{MPS} \langle C \rangle$. Weiterhin ist der Typ von B low. Daher gilt $B \equiv_L B$. Aus Satz 5.15 folgt somit

$$\langle \textbf{while } B \textbf{ do } C \textbf{ od} \rangle \approx_L^{MPS} \langle \textbf{while } B \textbf{ do } C \textbf{ od} \rangle$$

If-L: Nach Induktionsannahme gilt, dass $\langle C_1 \rangle \approx_L^{MPS} \langle C_1 \rangle$ und $\langle C_2 \rangle \approx_L^{MPS} \langle C_2 \rangle$. Weiterhin gilt $B \equiv_L B$. Aus Satz 5.15 folgt dann

$$\langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi} \rangle \approx_L^{MPS} \langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi} \rangle$$

If-H: Nach Induktionsannahme gilt, dass $\langle C_1 \rangle \approx_L^{MPS} \langle C_1 \rangle$. $\langle C_2 \rangle \approx_L^{MPS} \langle C_2 \rangle$. Weiterhin gilt $\langle C_1 \rangle \approx_L^{MPS} \langle C_2 \rangle$. Aus Satz 5.15 folgt dann

$$\langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi} \rangle \approx_L^{MPS} \langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi} \rangle$$

□

5.4 Analyse von Beispielprogrammen

Die in Kapitel 3 vorgestellten Programme, die einen unerlaubten Informationsfluss zulassen, werden von der obigen Sicherheitseigenschaft trivialerweise als unsicher eingestuft. Für das Programm in Abbildung 3.7 folgt dies aufgrund der ersten Zeile des Threads, der auf Prozessor 0 ausgeführt wird. Hier wird einer Low-Variable der Wert einer High-Variable zugewiesen. Nach Satz 5.7 ist das Programm daher nicht sicher. Gleiches gilt für das Programm in Abbildung 3.8. Hier ist Zeile 2 des zweiten Threads der Grund, dass das Programm nicht stark MPS-sicher ist.

5.5 Alternativen und Erweiterungen

Für den obigen Angreifer wurde eine starke Approximation im Modell gewählt, um Ein- und Ausgabekanäle zu modellieren. Hierdurch ergab sich eine restriktive Sicherheitseigenschaft, die jedoch eine vollständige Komposition aus sicheren Teilprogrammen erlaubt und voraussichtlich Scheduler-unabhängig ist. Eine Alternative zur Approximation, dass der Angreifer den gesamten relevanten Speicher einschließlich der ungesehenen Schreiboperationen sieht, ist, dass ein Teil der Low-Variablen ausschließlich zur Ausgabe dienen und eine Ausgabe immer dann stattfindet, wenn eine Zuweisung in eine solche Variable erfolgt. Dadurch wäre der Angreifer im Modell zwar schwächer, aber immer noch stärker als der informelle Angreifer. Zudem wäre das nachstehende Programm mit diesem Angreifer sicher, sofern man annimmt, dass temp und z Low-Variablen sind, aber nur z zur Ausgabe dient. Durch die Verwendung von Locks ist hier ein unerlaubter Informationsfluss ausgeschlossen.

l.acquire	l.acquire
temp := h_1	z := temp
h_1 := h_2	l.release
h_2 := temp	
temp := 0	
l.release	

Somit ließe sich hierdurch die Präzision der Sicherheitseigenschaft erhöhen, da starke MPS-Sicherheit das Programm ablehnt (temp := h_1). Wie bereits erwähnt wird in [MSS11] ein Ansatz vorgestellt, der z.B. obiges Programm als sicher klassifizieren kann. Eine Übertragung der Resultate auf den Multi-Prozessor-Fall ist daher wünschenswert, insbesondere da MTL-L Locks bereitstellt. Hierzu wäre es ebenso wünschenswert, wenn die Semantik explizit Ein- und Ausgabekanäle modelliert, um dem Angreifer im Modell nicht unnötig viel Wissen über den Speicher zugestehen zu müssen.

Ähnlich wie *strong security* betrachtet der obige Angreifer eine große Klasse an Schedulers. In [MS10] wird ein Ansatz vorgestellt, der die Klasse der Schedulers verkleinert und so eine präzisere Eigenschaft definiert. Hierzu ist jedoch, wie bereits im vorherigen Kapitel erwähnt, notwendig, dass die Systemzustände, um eine Scheduler erweitert werden. Weiterhin ist die Integration von Deklassifikation, wie in [LM09b] und [LM09a], wünschenswert.

Kapitel 6

Zusammenfassung

In dieser Arbeit wurde die in Sabelfeld [SS00] verwendete Beispielsprache verändert, sodass die Verwendung von Locks ermöglicht wurde. Für diese Sprache wurde anschließend eine neue Semantik entwickelt, die die Ausführung auf einem Dual-Prozessor-System modelliert. Hierbei wurde darauf geachtet, dass die Modellierung Ähnlichkeiten zu bestehenden Arbeiten in der Informationsflusssicherheit hat, um so besser Vergleiche ziehen zu können.

Anschließend wurde eine Sicherheitseigenschaft auf der obigen Semantik aufgestellt. Die Eigenschaft orientierte sich dabei an *strong security*. Sie basiert ebenfalls auf einer starken Bisimulation, die keine Deklassifikation zulässt. Der Ansatz wurde hierbei gewählt, um eine zu Anfangs einfache Eigenschaft zu betrachten, von der man ausgehen kann, dass sie ähnlich zu *strong security* Schedular-unabhängig ist. Dies stellt den ersten Versuch dar, Informationsflusssicherheit im Umfeld von Multi-Prozessor-Systemen zu betrachten. Die Ähnlichkeiten zwischen dem Typsystem für *strong security* und dem für *strong MPS security* zeigen auf, dass sich für diese Art von Sicherheit von Programmen hinsichtlich der Verwendung von Multi-Prozessor-Systemen keine zusätzlichen Anforderungen ergeben.

6.1 Verwandte Arbeiten

Im Bereich der Multi-Prozessor-Systeme existieren zahlreiche Arbeiten, um die Ausführung von Programmen auf diesen Systemen zu formalisieren.

In [SSO+10] und [SSA+11] stellen die Autoren axiomatische Semantiken für die Maschinensprachen von x86 und PowerPC Multi-Prozessor-Systemen vor. Ebenso wird in [MPA05] vorgestellt, welche Ausführungen im Java Memory Model gültig sind. Für das aktuelle C++ Speichermodell des C++11 Standards findet sich in [BOS+11] eine Formalisierung der gültigen Aus-

führungen. Allen vier gemein ist, dass sie beschreiben, welche Ausführungsspuren im Speichermodell gültig sind.

[DRD10] stellt eine operationelle Semantik für eine imperative GOTO-Sprache vor, die Register verwendet. Die Semantik unterteilt die Ausführung einer Anweisung in eine *issue*- und *commit*-Phase, wobei erstere die Ausführung der Anweisung durch den Prozessor darstellt und die zweite die Bekanntmachung der Anweisung an die anderen Prozessoren ist. Die einzelnen *issue*- und *commit*-Aktionen sind in einer Ausführung total geordnet und bieten daher keine echt parallele Ausführung von Anweisungen.

Die operationelle Semantik für eine Teilsprache von Java in [JPR10] nutzt eine Historie der getätigten Schreiboperationen und Lockoperationen, um bei einer Leseoperation zu ermitteln, von welchen Schreiboperationen ein Prozessor lesen kann. Weiterhin nutzt die Semantik sogenannte *speculations*, um spekulative Leseoperationen der Prozessoren zu modellieren. Dadurch ist das zugrundeliegende schwache Speichermodell schwächer als das Java Memory Model. Die Semantik unterstützt hingegen nicht die echt parallele Ausführung von Anweisungen.

6.2 Zukünftige Arbeiten

Die in dieser Arbeit vorgestellte Semantik nutzt ein schwaches Speichermodell, das jedoch nicht alle möglichen Relaxierungen umfasst. Damit ist sie z.B. nicht in der Lage alle möglichen Ausführungen eines Programms auf einer PowerPC-Maschine zu simulieren. Eine Erweiterung der Semantik in Hinsicht auf das Speichermodell ist daher wünschenswert. Hierbei kann man dem Ansatz von Jagadeesan folgen und *speculations* für $R \rightarrow R/W$-Relaxierungen nutzen. Zudem ist es wünschenswert, die Anzahl der Prozessoren von 2 auf eine beliebige Anzahl zu erhöhen. Hierbei könnten weitere Effekte von Multi-Prozessor-Systemen auftreten, die für die Informationflusssicherheit relevant sein könnten.

Weiterhin fehlt eine Modellierung von Schedulers in der Semantik. Die Nutzung von Schedulers im Modell würde es erlauben, schwächere Sicherheitseigenschaften zu definieren, die aber eine größere Präzision haben ([MS10]). Zudem ist eine Modellierung von Ein- und Ausgabekanälen sinnvoll, um Sicherheitseigenschaften auf Basis dieser formulieren zu können. Schließlich ist es wünschenswert, die Sprache um eine fork- oder spawn-Anweisung zu erweitern.

Die vorgestellte Sicherheitseigenschaft *strong MPS security* orientiert sich an *strong security*. Damit ist sie sehr restriktiv und lehnt viele Programme ab. Dazu zählen auch Programme, wie in 5.5, die durch die Verwendung

von Locks im Prinzip keine Sicherheitslücke enthalten. Ein Einbeziehen der Verwendung von Locks in Programmen ist daher sehr wünschenswert. Hierzu ist es möglich den Ansatz aus [MSS11] auf die vorgestellte Semantik zu übertragen. Weiterhin nutzt die Eigenschaft keine Deklassifikation, sodass auch hier eine Integration der bestehenden Arbeiten [LM09a] und [LM09b] interessant ist.

Literaturverzeichnis

[BOS+11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing c++ concurrency. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 55–66, New York, NY, USA, 2011. ACM.

[DRD10] A. De, A. Roychoudhury, and D. D'Souza. Womm: a weak operational memory model. *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 519–534, 2010.

[JPR10] R. Jagadeesan, C. Pitcher, and J. Riely. Generative operational semantics for relaxed memory models. *Programming Languages and Systems*, pages 307–326, 2010.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

[LM09a] Alexander Lux and Heiko Mantel. Declassification with explicit reference points. In Michael Backes and Peng Ning, editors, *14th European Symposium on Research in Computer Security*, volume 5789 of *LNCS*, pages 69–85. Springer, 2009.

[LM09b] Alexander Lux and Heiko Mantel. Who can declassify? In P. Degano, J. Guttman, and F. Martinelli, editors, *Proceedings of the Workshop on Formal Aspects in Security and Trust, FAST 2008*, volume 5491 of *LNCS*, pages 35–49. Springer, 2009.

[MPA05] J. Manson, W. Pugh, and S.V. Adve. *The Java memory model*, volume 40. ACM, 2005.

[MS04] Heiko Mantel and David Sands. Controlled Declassification based on Intransitive Noninterference. In *Proceedings of the 2nd ASIAN Symposium on Programming Languages and Systems, APLAS*

2004, LNCS 3302, pages 129–145, Taipei, Taiwan, November 4–6 2004. Springer-Verlag.

[MS10] Heiko Mantel and Henning Sudbrock. Flexible scheduler-independent security. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, LNCS 6345, pages 116–133. Springer, 2010.

[MSS11] Heiko Mantel, David Sands, and Henning Sudbrock. Assumptions and guarantees for compositional noninterference. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium (CSF)*, pages 218–232, Cernay-la-Ville, France, 2011. IEEE Computer Society.

[SS00] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Computer Security Foundations Workshop, 2000. CSFW-13. Proceedings. 13th IEEE*, pages 200–214. IEEE, 2000.

[SSA+11] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding power multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 175–186, New York, NY, USA, 2011. ACM.

[SSO+10] P. Sewell, S. Sarkar, S. Owens, F.Z. Nardelli, and M.O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, 2010.