Karlsruhe Institute of Technology

Automatische Korrektur von Wettlaufbedingungen

Diplomarbeit
von

Atanas Dimitrov

An der Fakultät für Informatik
Institute for Program Structures
and Data Organization (IPD)

Erstgutachter:	Prof. Dr. Walter F. Tichy
Betreuende Mitarbeiter:	Dipl.-Inform. Korbinian Molitorisz
	Dipl.-Inform. Jochen Schimmel

Bearbeitungszeit: 01. Januar 2012 - 29. Juni 2012

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 29.06.2012

Kurzfassung

Mit dem steigenden Einsatz von Multikernprozessoren stoßen Entwickler mehr und mehr auf ein schwerwiegendes Problem: das Erzeugen von korrektem, parallelem Code. Wettlaufbedingungen sind vom Anwender und von der Ablaufsteuerung des Betriebssystems abhängig und bleiben oft während der Qualitätssicherung der Softwareprodukte unentdeckt. Zahlreiche automatische Erkenner setzen sich mit diesem Problem auseinander. Alle Ergebnisse sind aber nutzlos, wenn die gefundenen Probleme nicht oder nur teilweise korrigiert werden. Existierende Verfahren zur automatischen Korrektur von Datenwettläufen sind stark eingeschränkt und verwenden unsichere Korrekturmechanismen. In dieser Arbeit wird ein Verfahren zur automatischen Korrektur von Datenwettläufen in parallelem und getestetem Code vorgestellt und implementiert. Fehlerstellen im Programmcode und in externen Bibliotheken werden durch Komponententests erkannt und durch sichere Synchronisationsblöcke und Austausch von parallel unsicheren Datenstrukturen ohne Benutzerbeteiligung im Quelltext korrigiert. Das Verfahren wird von einer statischen Kontroll- und Datenflussanalyse unterstützt.

Zur Validierung wird ein Werkzeug zur automatischen Korrektur von .NET-Anwendungen implementiert. Microsoft CHESS wird zum Erkennen der Datenwettläufe verwendet. Das Werkzeug wird anhand von 7 parallelen .NET-Anwendungen evaluiert.

Danksagung

An dieser Stelle möchte ich Herrn Professor Tichy und meinen Betreuern Jochen Schimmel und Korbinian Molitorisz für die Unterstützung bei meiner Diplomarbeit und die gute Zusammenarbeit im ESC danken.

Ich danke Max Schroeder, Jan Gärtner und Michaela Davidkova für die schnelle Korrektur.

Mein besonderer Dank gilt meinen Eltern und meiner ganzen Familie für ihre große Unterstützung während des gesamten Studiums.

Inhaltsverzeichnis

1. Einleitung

Mit dem steigenden Einsatz von Multikernprozessoren stoßen Entwickler mehr und mehr auf ein schwerwiegendes Problem: Das Erzeugen von korrektem, parallelem Code. Parallelisierungsfehler wie Verklemmungen und Wettlaufbedingungen sind vom Anwender und von der Ablaufsteuerung des Betriebssystems abhängig und bleiben oft während der Qualitätssicherung der Softwareprodukte unentdeckt. Verschiedene Systeme, die Datenwettläufe enthalten, zeigen verschiedene Symptome. Das macht das Reproduzieren der Probleme und das Erkennen der Ursache schwer und nicht eindeutig.

Therac-25, ein Gerät zur Strahlentherapie der kanadischen Firma Atomic Energy of Canada Limited (AECL), kostete zwischen 1985 und 1987 drei Menschen das Leben und verletzte drei weitere schwer. Die schnelle Bedienung des Gerätes konnte zu einem Datenwettlauf zwischen zwei Prozessen führen. Die fehlerhafte Synchronisation war einer der Gründe für eine Überdosis an Strahlung. Das Problem konnte lange Zeit nicht reproduziert werden und der Hersteller vermutete die Existenz defekter Hardwareteile. In einem Bericht wurden unter anderem die folgenden Ursachen ermittelt: Nicht ausreichende Testphase und Qualitätssicherung der Software, fehlende Dokumentation, übermäßiges Vertrauen auf das Gerät. [LT93, Lev85].

2003 ließ ein Datenwettlauf 50 Millionen Menschen in Nordamerika ohne Elektrizität. Nach acht Wochen Systemuntersuchung wurde festgestellt, dass eine Reihe von Ereignissen zu einem parallelen Schreibzugriff in einer Datenstruktur geführt hatte. Der parallele Schreibzugriff ließ die Datenstruktur in einem ungültigen Zustand und verhinderte das rechtzeitige Melden des Problems. Der Datenwettlauf trat während einer Testphase von drei Millionen Stunden nicht auf und hatte ein Auftrittsfenster im Millisekundenbereich. [Pou04]

Eine Vielzahl an Erkennungswerkzeugen existiert (siehe Abschnitt 3.2), die Datenwettläufe anhand statischer und dynamischer Analyse automatisch erkennen. Alle diese Detektoren sind aber nutzlos, wenn die gefundenen Probleme nicht oder nur teilweise korrigiert werden. In komplexeren Fällen haben nicht nur unerfahrene Entwickler Schwierigkeiten, eine Lösung zu erkennen und korrekt zu implementieren, sondern auch Entwickler mit jahrelanger Erfahrung. Existierende Verfahren für automatische Korrektur und Vermeidung von Datenwettläufen (siehe Abschnitt 3.3) haben eine eingeschränkte Anzahl an Fällen, die sie abdecken, verwenden unsichere Korrekturverfahren oder befinden sich in einer frühen Entwicklungsphase.

1.1 Ziele der Arbeit

Ziel dieser Arbeit ist es, Entwickler bei der Korrektur von Datenwettläufen zu unterstützen, in dem ein automatisches Verfahren zum Erzeugen von Korrekturvorschlägen für Wettlaufbedingungen im parallelen und getesteten Code und ihren Einbau im Quelltext der Anwendung entwickelt wird. Das Verfahren soll Datenwettläufe, die sowohl im eigenen als auch im externen Code stattfinden, anhand sicherer Synchronisationsblöcke und Austausch von parallel unsicheren Datenstrukturen korrigieren. Dabei sollen die Performanz und die Verklemmungsfreiheit des erzeugten Codes berücksicht werden.

1.2 Aufbau der Arbeit

Diese Arbeit ist folgendermaßen aufgebaut:

Kapitel 2 erläutert die grundlegenden Begriffe der parallelen Programmierung und Technologien, die in dieser Arbeit zum Einsatz kommen.

In Kapitel 3 werden verwandte Arbeiten und Werkzeuge beschrieben, die die automatische Erkennung und Korrektur von Datenwettläufen in parallelen Anwendungen als Schwerpunkt haben.

In Kapitel 4 wird das im Rahmen dieser Arbeit entwickelte Verfahren zur automatischen Korrektur von Wettlaufsituationen durch Synchronisationsblöcke und Austausch von Datenstrukturen vorgestellt.

Kapitel 5 diskutiert eine Implementierung des vorgestellten Entwurfs in der .NET-Sprache C# sowie den Einsatz notwendiger externer Werkzeuge und Technologien. Es werden die Architektur sowie die einzelnen Komponenten des Werkzeuges erläutert.

In Kapitel 6 werden die Ergebnisse der Validierung des Verfahrens vorgestellt.

Zum Schluss wird die gesamte Arbeit in Kapitel 7 zusammengefasst und ein Ausblick auf zukünftige Erweiterungsmöglichkeiten gegeben.

2. Grundlagen

Dieses Kapitel erläutert grundlegende Begriffe und Technologien, die ein Leser braucht, um die Problematik und den Ansatz dieser Diplomarbeit zu verstehen.

Weitergehende Informationen über die in diesem Kapitel behandelten Themen sind in den vorgeschlagenen Arbeiten im Literaturverzeichnis zu finden.

2.1 Grundbegriffe

In diesem Abschnitt werden grundlegende Begriffe der parallelen Programmierung erklärt.

2.1.1 Prozesse, Ausführungfäden und Parallelität

Der Prozess ist eine Abstraktion eines laufenden Programms [Tan09]. Er enthält u. a. einen eigenen Adressraum: einen kontinuierlichen Speicherbereich, in dem er lesende und schreibende Speicherzugriffe ausführen kann. Prozesse dürfen nur auf den eigenen Adressraum zugreifen und sind somit voneinander isoliert. Der Adressraum enthält u. a. den Code des ausgeführten Programms sowie Programmdaten wie Variablenwerte und Konstanten. Ein Prozess enthält des Weiteren n ausführbare Einheiten, die Ausführungsfäden (Englisch: Threads) genannt werden. Ein Prozess besteht mindestens aus einem Thread, sodass er die geladenen Instruktionen ausführen kann[1].

Threads sind die kleinste ausführbare Einheit, die vom Betriebssystem gesteuert wird[2]. Ein Thread existiert innerhalb eines Prozesses und teilt sich den Adressraum mit allen anderen Threads dieses Prozesses. Er hat einen eigenen Instruktionszeiger und Aufrufstapel (Englisch: call stack). Mehrere Threads, die nicht unbedingt zum gleichen Prozess gehören, können vom Betriebssystem parallel ausgeführt werden, wobei die Scheduling Policy (auf Deutsch: Steuerungsvorgehensweise) nur beschränkt vom Anwendungsentwickler beeinflussbar ist. Scheduling Policies unterscheiden sich von einem Betriebssystem zum anderen und können auch zwischen Versionen desselben Betriebssystems abweichen. Die Anzahl der parallel laufenden Threads ist von der Hardware (Prozessor-, Prozessorkerneanzahl)

[1] Die Beschreibung ist architekturabhängig und bietet nur einen Überblick auf Prozesse und Threads unter Betriebssystemen wie Windows und Linux.

[2] Konzepte für Threads auf Benutzerebene (Englisch: user level) wie Green Threads oder Fibers in Windows werden in dieser Arbeit nicht berücksichtigt. In Abhängigkeit davon, welche Synchronisationsmechanismen von der Implementierung dieser Konzepte angeboten werden, könnte der vorgestellte Ansatz anwendbar sein.

abhängig. Beherrscht das Betriebssystem prämptives Scheduling[3], kann die Ausführung eines Threads an einer beliebigen Stelle (bei einer beliebigen Instruktion) unterbrochen werden. Ursachen für Unterbrechungen sind zum Beispiel:

- Hardware Interrupts
- der Thread hat die eigene Ausführungszeitspanne verbraucht
- der Thread wartet auf eine Ressource oder ein Ereignis
- der Thread hat eine Eingabe bzw. eine Ausgabe angefangen (Englisch: I/O)
- der Thread hat freiwillig die CPU freigegeben

In einem modernen Betriebssystem existieren gleichzeitig viele Threads, die zu verschiedenen Prozessen gehören. Die meisten davon befinden sich in einem blockierten Zustand (Englisch: Blocked) und warten auf ein Ereigniss. Wenn ein Thread lauffähig (Englisch: Ready) wird, kann er vom Betriebssystem wieder einem Prozessor zugewiesen werden. Dieses stark vereinfachte Zustandsmodell ist auf Abbildung 2.1 zu sehen. Weitere Zustände, wie zum Beispiel `New` oder `Terminated` existieren in manchen Implementierungen. Die Windows-Threads haben zum Beispiel 10 verschiedene Zustände [Net].

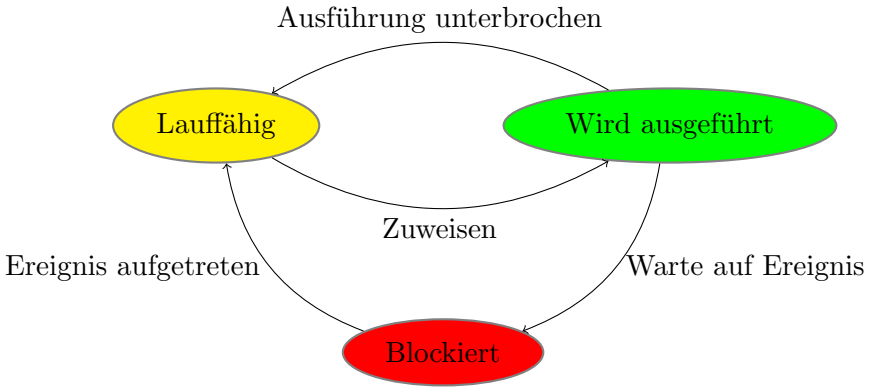

Abbildung 2.1: Dreistufiges Threadzustandmodell.

Threads, die zu verschiedenen Prozessen gehören, sind dank der Prozessisolation voneinander geschützt und können nur über vom Betriebssystem zur Verfügung gestellten Kommunikationsmittel interagieren. Threads, die zum selben Prozess gehören, können auch über den gemeinsamen Speicher (Adressraum) kommunizieren, was Probleme wie Verklemmungen und Datenwettläufe verursachen kann [Tan09].

Um die steigende Rechenleistung und Anzahl der Prozessoren/Prozessorkerne in einem Rechner sinnvoll auszunutzen, werden Anwendungen oftmals parallelisiert. Das ist dann möglich, wenn sich zum Beispiel die Funktionalität in kleine voneinander unabhängige Funktionen unterteilen lässt oder wenn dieselbe Funktion auf eine große Datenmenge unabhängiger Elemente angewendet wird. Die Parallelisierung einer Anwendung kann aber auch zu einer schlechteren Performanz führen, wenn parallele Aufgaben sich gegenseitig blockieren. Dies kann vor allem beim Zugriff auf gemeinsamen Daten schnell auftreten.

Parallel ist jede Anwendung, bei der neue Threads explizit durch den Entwickler oder indirekt durch Aufruf asynchroner Funktionen oder Aufruf der Methoden einer Bibliothek erzeugt werden. Das Hinzufügen von Parallelität macht eine Anwendung für Verklemmungen und Datenwettläufe anfällig. Diese werden in den nächsten Abschnitten vorgestellt.

[3] Im Unterschied zu präemptivem Scheduling, wird bei nicht präemptivem Scheduling die Ausführung eines Threads von außen (bei Hardware Interrupts) nicht unterbrochen. Die Threads laufen, bis sie die CPU freigeben [Tan09].

2.1.2 Atomizität

Wenn man ein sequenzielles Programm implementiert, geht man davon aus, dass Befehle ununterbrochen und in der gleichen sequenziellen Reihenfolge ausgeführt werden. Auch wenn der einzige Thread des sequenziellen Programms vom Betriebssystem unterbrochen wird, bleibt sein Zustand unverändert, bis die Ausführung fortgesetzt wird. Bei parallelen Programmen muss das nicht immer der Fall sein.

```
1. //C# Darstellung
2. j = i + 7;
3.
4. //CIL Darstellung
5. IL_0003: ldfld i
6. IL_0004: ldc.i4.7
7. IL_0005: add
8. IL_0006: stfld j
```

Abbildung 2.2: Ein C#-Statement und seine CIL Darstellung

Auf Abbildung 2.2 wird die Common Intermediate Language [Int10] Darstellung eines C#-Statements angezeigt. Wird der Thread vom Betriebssystem in Zeile 6 unterbrochen, kann es bei paralleler Ausführung passieren, dass x der anderen n Threads weiter ausgeführt werden und den Zustand des Programms ändern. Es können auch Daten geändert werden, die gerade vom angehaltenen Thread verwendet werden (die Werte die in i und j gespeichert sind). Solche Daten sind globale und statische Variablen, können aber auch Instanzvariablen sein. Im folgenden Abschnitt wird erläutert, wie eine unerwünschte Zustandsänderung bei paralleler Ausführung vermieden werden kann.

2.1.3 Synchronisation

Um unerwünschte Zustandsänderung bei paralleler Ausführung zu vermeiden, müssen alle Codebereiche, die diese Änderung verursachen können, synchronisiert werden. Wenn ein Thread (T1) einen der n Bereiche beigetreten hat, dann darf kein anderer Thread einen der Bereiche ausführen, bis T1 seinen verlässt oder bis er freiwillig blockiert. Dieses Konzept nennt man auch atomare Ausführung aller Bereiche. Betriebssysteme bieten verschiedene Synchronisationskonzepte an: zum Beispiel Semaphoren, Mutexe und Monitore. Im weiteren Verlauf dieses Abschnittes werden lediglich Monitore weiter erläutert, da diese zum weiteren Verständnis dieser Arbeit wichtig sind. Alle drei Synchronisationskonzepte werden ausführlich in [Tan09] beschrieben.

Monitore

> Ein Monitor ist eine Sammlung von Prozeduren, Variablen und Datenstrukturen, die in einem Modul oder in einer Klasse gruppiert sind [Tan09].

Codeteile, die nicht parallel ausgeführt werden dürfen, werden in Monitorprozeduren transformiert. Alle Prozeduren des Monitors können jederzeit aufgerufen werden, aber keine zwei Aufrufe werden parallel ausgeführt. Kann eine Monitorprozedur nicht ausgeführt werden, so wird der Thread im Monitor blockiert, bis der gerade ausgeführte Thread ihn verlässt. Dieses Verhalten wird mithilfe eines `lock`-Objekts (Deutsch: Sperre) implementiert. Die Sperre muss vor jeder Monitorprozedur vom Thread akquiriert werden und beim Verlassen des Monitors wieder freigegeben werden. Betriebssystemroutinen, die Sperren freigeben und akquirieren sind atomar, hierzu ist eine Hardware-seitige Unterstützung nötig. Wie diese auf Hardwareebene implementiert ist, wird in [Int12] erläutert.

In Java werden Monitorprozeduren mit dem Schlüsselwort `synchronized` definiert. C# bietet eine Methodenannotation an: `[MethodImpl(MethodImplOptions.Synchronized)]`. Beide Sprachen können auch Methodenteile als monitorgeschützt definieren, entsprechend mit den `synchronized(lockObject)` und `lock(lockObject)` Blöcken. Um beliebige Codebereiche zu schützen, bietet C# auch die `Monitor.Enter(lockObject)` und `Monitor.Exit(lockObject)` Methoden.

Für den Fall, dass Threads in einer Monitorprozedur warten müssen, weil eine Bedingung nicht erfüllt ist, bietet dieses Konzept Bedingungsvariablen an. Diese enthalten eine Warteschlange für wartende Threads und bieten die `Wait()`- und `Signal()`-Prozeduren zum Warten und Benachrichtigen an. Wenn ein Thread `Wait()` aufruft, dann gibt er die akquirierte Sperre frei und wartet auf eine Änderung des Programmzustandes. Ruft ein anderer Thread `Signal()` auf, dann benachrichtigt er einen der wartenden Threads, dass sich der Programmzustand geändert hat und gibt ihn frei. `SignalAll()` gibt alle wartenden Threads frei. Diese Prozeduren können nur innerhalb des Monitors aufgerufen werden. Java und C# bieten keine dedizierten Bedingungsvariablen an. Die Funktionalität wird in diesen Sprachen von der Sperre übernommen.

2.1.4 Datenwettläufe und Verklemmungen

In diesem Abschnitt werden drei Fehlertypen erläutert, die nur bei paralleler Ausführung beobachtet werden können:

- Data Races (Deutsch: Datenwettläufe)
- Deadlocks (Deutsch: Verklemmungen)
- Livelocks

Obwohl sich die Diplomarbeit nur mit der Analyse und mit dem Lösen von Datenwettläufen beschäftigt, werden hier alle drei Gruppen erläutert. Das kommt daher, dass alle drei stark zusammenhängen und das Lösen von einem Problem aus einer Gruppe ein Problem aus einer der anderen Gruppen verursachen kann. Dieses Phänomen wird im nächsten Abschnitt detaillierter erklärt.

Datenwettläufe

Datenwettläufe können dann entstehen, wenn mehrere Threads gleichzeitig auf dieselben Daten zugreifen und mindestens einer der Zugriffe ein Schreibzugriff ist [Tan09].

Datenwettläufe kann man in zwei Gruppen unterteilen:

- Harmlose (Englisch: benign): Diese haben keine negative Auswirkung auf die Programmausführung. Das sind zum Beispiel Zähler, die beim Logging verwendet werden. In diesem Fall macht es keinen Unterschied, ob der Zähler um 1 höher oder niedriger ist.
- Schwerwiegende: In diese Kategorie fallen alle Wettlaufbedingungen, bei denen es zu einer Dateninkonsistenz kommen kann. Als Auswirkung gerät das Programm in einen ungültigen oder indeterministischen Zustand. Dabei können Ergebnisse und Ausführungspfade geändert werden.

Zu welcher dieser Kategorien ein Datenwettlauf gehört, hängt von der Semantik des Codes ab und kann anhand eines Werkzeugs im Allgemeinen nicht automatisch erkannt werden. Aus diesem Grund behandelt diese Diplomarbeit alle gemeldeten Datenwettläufe gleich und versucht diese zu korrigieren.

```
1.  private int x = 0;

2.  public void Increment(int amount)
3.  {
4.      if (x >= 0 && x + amount <= 100)
5.      {
6.          x = x + amount;
7.      }
8.  }

9.  public void Decrement(int amount)
10. {
11.     if (x <= 100 && x - amount >= 0)
12.     {
13.         x = x - amount;
14.     }
15. }
```

Abbildung 2.3: Ausschnitt einer C#-Klasse. Die Instanzvariable `x` wird nur durch Aufrufe der zwei Instanzmethoden `Increment` und `Decrement` modifiziert. Die zwei Methoden überprüfen, ob der Wert von `x` vor und nach der Modifikation im $[0, 100]$ liegt. Ist er nicht im Intervall, so wird die Variable nicht modifiziert.

Das Beispiel auf Abbildung 2.3 zeigt eine typische Situation, in der ein Datenwettlauf entsteht: Wenn die Methoden `Increment` und `Decrement` n-Mal sequenziell ausgeführt werden, behält die Variable immer einen Wert im Intervall $[0, 100]$. Das Verhalten ist deterministisch und vorhersagbar für eine gegebene Reihenfolge von `Increment`/`Decrement`-Aufrufen.

Bei Ausführung mit mehreren Threads verhält sich dieser Code nicht deterministisch. Trotz der zwei `if`-Bedingungen kann der Variablen `x` ein Wert zugewiesen werden, der außerhalb des Intervalls liegt. Die Fehlerursache wird im Folgenden verdeutlicht:

- Fall 1: `x` hat einen Wert von 10. Thread A führt `Decrement` mit `amount = 6` und Thread B führt parallel dazu `Decrement` mit `amount = 7`. Thread A überprüft die if-Bedingung und diese stimmt; danach wird Thread A vom Betriebssystem unterbrochen und Thread B wird ausgeführt. Thread B überprüft die if-Bedingung und diese stimmt auch, weil Thread A die Variable noch nicht modifiziert hat. Thread B modifiziert die Variable und diese bekommt den Wert 3. Danach wird Thread A ausgeführt und er modifiziert den Variablenwert auch. Als Ergebnis bekommt `x` den Wert -3. Dieser liegt nicht mehr im Intervall $[0, 100]$.
- Fall 2: Das Problem von `Decrement` findet man auch in `Increment`: Wenn `Increment` von mehreren Threads gleichzeitig ausgeführt wird und das Betriebssystem die Threads an den geeigneten Stellen unterbricht, kann `x` einen Wert über 100 annehmen.
- Fall 3: Hierbei sind beide Methoden an der Modifikation der Variablen beteiligt: Thread A führt `Decrement` mit `amount = 6` aus und Thread B führt `Increment` mit `amount = 10` aus. Der Initialwert von `x` ist 50.

  Um diesen Fall besser zu erklären, betrachten wir die Darstellung einer Anweisung auf Maschinenebene. Analog zu Beispiel 2.2 wird `x = x + amount;` in mehrere Maschineninstruktionen übersetzt. Die genauen Ausführungsschritte können von Prozessorarchitektur zu Prozessorarchitektur abweichen, entsprechen aber im Allgemeinen dem folgenden Muster:

  1. Lade den ersten Operanden (`x`) in ein Register.
  2. Lade den zweiten Operanden (`amount`) in ein anderes Register.
  3. Addiere die Werte die in den beiden Registern liegen und speichere das Ergebnis in einem dritten Register.

4. Schreibe das Ergebnis zurück in die Variable `x`.

In diesem Fall haben die zwei Threads die `if`-Bedingung bereits überprüft. Thread A lädt den Wert von `x` in ein Register und wird danach vom Betriebssystem unterbrochen. Danach kommt Thread B und führt die Anweisung `x = x + amount;` aus. Anschließend hat `x` den Wert 60. Thread B wird unterbrochen und Thread A wird weiter ausgeführt. Thread A hat noch 50 als den Wert von `x`. Nach der vollständigen Ausführung von `x = x - amount;` von Thread A bekommt `x` den Wert 44. Die Änderung von Thread B ist verschwunden.

Führt man dieses Beispiel sequenziell aus, so wird immer ein korrekter Wert in die Variable `x` geschrieben. Der Wert wird auch bei paralleler Ausführung korrekt sein, wenn die Threads nicht an ungünstiger Stelle unterbrochen werden. Das macht das Erkennen und das Beheben von Multithreadingproblemen kompliziert.

```
1.  private int x = 0;

2.  public void Increment(int amount)
3.  {
4.      Monitor.Enter(this);
5.      if (x >= 0 && x + amount <= 100)
6.      {
7.          x = x + amount;
8.      }
9.      Monitor.Exit(this);
10. }

11. public void Decrement(int amount)
12. {
13.     Monitor.Enter(this);
14.     if (x <= 100 && x - amount >= 0)
15.     {
16.         x = x - amount;
17.     }
18.     Monitor.Exit(this);
19. }
```

Abbildung 2.4: Um die zwei kritischen Abschnitte gegen parallele Ausführung zu schützen, wird ein Monitor verwendet. Das Instanzobjekt (`this`) der Klasse, die die zwei Methoden enthält, wird als Sperre genommen. Eine dedizierte Sperre kann auch verwendet werden.

Die zwei `if`-Blöcke definieren zwei kritische Abschnitte. Kritische Abschnitte (Englisch: critical section) sind Codebereiche im parallelen Programm die auf gemeinsame Daten zugreifen. Solche Codebereiche müssen synchronisiert werden, um Probleme zu vermeiden. In Abbildung 2.4 ist dargestellt, wie das anhand des Monitorkonzepts in C# gemacht werden kann. Nach dieser Änderung können keine zwei Threads die kritischen Abschnitte für dieselbe Instanz der Klasse parallel ausführen. Selbst der gleiche Abschnitt kann nicht von zwei Threads gleichzeitig ausgeführt werden.

Das korrekte Einfügen von Sperren und anderen Synchronisationskonstrukten, um parallele Zugriffe auf gemeinsame Daten zu vermeiden, ist kompliziert und die genauen Stellen sind nur in einfachen Situationen offensichtlich. Falsche Lösungsansätze können zu weiteren Problemen wie Deadlocks und Livelocks führen.

Deadlocks

Wenn mindestens zwei Prozesse oder Threads exklusiven Zugriff auf mindestens zwei Ressourcen benötigen, diese aber nicht in derselben Reihenfolge akquirieren, kann ein Deadlock (Deutsch: Verklemmung) entstehen. Jeder Prozess bzw. jeder Thread hält nur eine Teilmenge der Ressourcen und kann keinen Fortschritt machen. Ressourcen können zum Beispiel Hardwaregeräte, Datenbankeinträge, Dateien oder Variablen sein. In diesem Abschnitt werden Verklemmungen zwischen Threads erläutert, weil nur diese für die Arbeit relevant sind. Die Sperren, die in Abschnitt 2.1.3 vorgestellt wurden, sind die Ressourcen.

Damit ein Deadlock entsteht, müssen 4 Bedingungen erfüllt sein [Tan09].

1. Jede Ressource kann von nur einem Thread gehalten werden.
2. Threads können exklusiven Zugriff auf mehrere Ressourcen anfordern.
3. Ressourcen können von einem Thread nicht weggenommen werden. Der Thread selbst muss sie freigeben.
4. Es muss ein Kreis von mindestens zwei Threads entstehen, in dem jeder Thread eine Ressource braucht, die zu der Zeit von einem anderen Thread gehalten wird.

```
1.  public void Method1()
2.  {
3.      Monitor.Enter(A);
4.      Monitor.Enter(B);
5.      //Critical Region 1
6.      Monitor.Exit(B);
7.      Monitor.Exit(A);
8.  }
```

```
9.  public void Method2()
10. {
11.     Monitor.Enter(B);
12.     Monitor.Enter(A);
13.     //Critical Region 2
14.     Monitor.Exit(A);
15.     Monitor.Exit(B);
16. }
```

Abbildung 2.5: Die zwei Methoden werden parallel ausgeführt. Thread1 führt `Method1` und Thread2 `Method2` aus. Eine Verklemmung entsteht, wenn Thread1 nach der Ausführung von `Monitor.Enter(A)` unterbrochen und Thread2 ausgeführt wird. Thread2 führt `Monitor.Enter(B)` aus und sperrt Variable `B`. Ab diesem Zeitpunkt befinden sich die zwei Threads in einem Deadlock.

Auf Abbildung 2.5 ist ein C# Codebeispiel gegeben, das bei einer parallelen Ausführung eine Verklemmung verursachen könnte. Diese Verklemmung lässt sich anhand eines Graphen darstellen (siehe Abbildung 2.6).

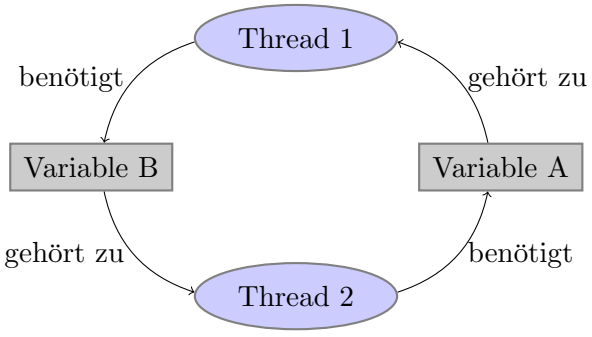

Abbildung 2.6: Eine Verklemmungssituation zwischen zwei Threads, die beide exklusiven Zugriff auf zwei Variablen benötigen. Thread1 hat exklusiven Zugriff auf `A` erhalten und benötigt einen exklusiven Zugriff auf `B`. Thread2 hat exklusiven Zugriff auf `B` und benötigt `A`.

Livelocks

Livelocks sind eine spezielle Form von Deadlocks, bei denen die Threads nicht blockiert sind, trotzdem aber keinen Fortschritt machen können. Als Beispiel für ein Livelock kann man sich zwei Personen vorstellen, die sich in einem Gang entgegenkommen. Sie versuchen fortwährend einander auszuweichen, aber immer in der gleichen Richtung.

Der in dieser Diplomarbeit vorgestellte Ansatz kann als Seiteneffekt der Quelltextkorrektur Verklemmungen im Code einbauen. Um die Auftrittschance zu minimieren, werden Mechanismen zum Erkennen und Vermeiden von Verklemmungen vorgestellt: Ändern der Sperrenreihenfolge und Anpassung der Regionen (siehe Abschnitt 4.12).

Mehr Informationen über Verklemmungen findet man in [Tan09].

2.1.5 Komponententests

Komponententests (Englisch: Unit Test) sind Softwaretests, die die Funktionalität der kleinsten Komponente, der Methode, einer Anwendung auf Korrektheit überprüfen. Ein Komponententest testet üblicherweise genau eine Methode, die sogenannte Method under Test (MUT). Je nach Komplexität werden viele Tests pro Methode geschrieben, um verschiedene Kontrollflüsse und Eingabewerte zu verifizieren. Mit Komponententests lassen sich u. a. Rückgabewerte, Fehler und Aufrufreihenfolgen mit vom Entwickler vordefinierten Werten vergleichen.

Bei Komponententests wird die getestete Methode vom Rest der Anwendung isoliert, sodass die Testergebnisse sich nur auf die MUT beziehen. Benötigte Teile des ungetesteten Codes können mit sogenannten Mock-Objekten ersetzt werden. Ruft die getestete Methode eine andere auf, so kann im Mock-Objekt das Verhalten (Rückgabewert, Fehler) der aufgerufenen Methode festgelegt werden. Während des Tests wird nur das, im Mock-Objekt definierte, Verhalten ausgeführt und nicht die echte Methode.

Komponententests werden in Testpakete nach dem getesteten Codebereich gruppiert.

Parallele Komponententest

Parallele Komponententests sind Komponententests, die das Zusammenspiel von mehreren parallel laufenden Methoden auf Fehler überprüfen. Dafür werden im Test neue Threads erstellt, die die parallelen Codeteile ausführen.

In dieser Arbeit werden parallele Komponententests verwendet, um Datenwettläufe zu erkennen. Dabei muss Folgendes beim Testaufbau betrachtet werden:

- Testabdeckung: Wenn die Tests nicht den kompletten Code abdecken, dann können nicht alle Wettlaufsituationen erkannt werden.
- Korrektheit: Die Tests müssen korrekt laufen. D.h., bevor der Code auf parallele Probleme getestet wird, sind alle anderen gemeldeten Probleme zu beheben. Die entwickelten Konzepte können mit Problemen wie falsche Rückgabewerte und Exceptions nicht umgehen.
- Werden Rückgabewerte im Test verifiziert, so muss sichergestellt werden, dass alle Threads ihre Arbeit erledigt haben, bevor eine Verifikation stattfindet.
- Auch wenn keine Werte verifiziert werden, muss sichergestellt werden, dass der Test erst dann beendet wird, wenn alle erzeugten Threads ihre Arbeit erledigt haben.

Wegen der indeterministischen Natur von Threads können Testergebnisse in Abhängigkeit vom zur Laufzeit verwendeten Scheduling abweichen. Deswegen reichen die parallelen Komponententests nicht aus, um den Code zu testen und alle existierenden Wettlaufsituationen zu finden. Eine spezielle Testumgebung, wie zum Beispiel Microsoft CHESS (siehe Abschnitt 2.2.5) ist notwendig, um die Tests unten verschiedenen Schedules auszuführen.

In der Arbeit werden nur die generierten Ergebnisse verwendet. Das bedeutet, dass das vorgestellte Verfahren auch manuell beschriebene Wettlaufsituationen korrigieren kann.

2.1.6 Aufrufstapel, Stack Trace und Stack Frame

Der Aufrufstapel (Englisch: call stack) ist eine thread-lokale Datenstruktur, die zur Laufzeit als Threadgedächnis benutzt wird. Hier wird die Hierarchie der bereits aufgerufenen Methoden mit lokalen Variablen, Argumenten und Rücksprungadressen gespeichert, sodass der Thread nach Verlassen einer Methode den Weg zurück zur Aufrufermethode findet.

Ein Stack Frame ist ein einzelnes Element des Aufrufstapels. Es liefert Informationen über die aufgerufene Methode, das genaue Methodenoffset, lokal gespeicherte Variablen und andere.

Der Stack Trace ist eine zusammengefasste Darstellung des Aufrufstapels zu einem bestimmten Zeitpunkt. Oft werden Stack Traces bei der Fehlersuche und Analyse verwendet. Ein Stack Trace enthält normalerweise den vollqualifizierten Namen der aufgerufenen Methoden und Quellcodezeile falls vorhanden. Variableninformationen werden an dieser Stelle ignoriert.

Informationen des Aufrufstapels werden in dieser Diplomarbeit als Teil der Analyse verwendet, die die genaue Fehlerstelle im Code sucht.

2.1.7 Code-Mutation

Zu der automatischen Änderung des Quelltextes eines Programms, die Code-Mutation, gehören das Hinzufügen, Editieren und Löschen von Anweisungen, Variablen, Methoden, Klassen und weiterer Elemente.

Ziel einer Mutation ist, Fehler im vorhandenen Code zu korrigieren. Eine Mutation wird als erfolgreich bezeichnet, wenn die resultierende Anwendung danach fehlerfrei kompiliert, ausgeführt werden kann und seine Semantik erhält.

2.2 Verwendete Technologien

In diesem Abschnitt werden die zur Implementierung verwendeten Technologien vorgestellt.

2.2.1 Microsoft .NET und die Common Language Infrastructure

Die Common Language Infrastructure (CLI) ist eine in ECMA-335 [Int10] standardisierte Beschreibung einer virtuellen Maschine. Teile dieses Standards sind:

1. das verwendete Typsystem (Common Type System oder CTS).
2. der Instruktionssatz und seine Regeln (Common Language Specification oder CLS).
3. sprachunabhängige Metadaten, die die Arbeit mit CLI-konformen Bibliotheken (in einer beliebigen CLI-konforme Programmiersprache geschriebene Bibliotheken) ermöglichen.
4. ein Laufzeitsystem (Virtual Execution System oder VES), das für das Ausführen CLI-konformer Anwendungen verantwortlich ist.

Der Standard wurde von Microsoft, Intel und anderen Partnern entwickelt. Ziel der Common Language Infrastructure ist es, eine uniforme, sprach- und systemneutrale Plattform für die Anwendungsentwicklung anzubieten. Dieses Ziel wird durch die Zwischensprache Common Intermediate Language (CIL) erreicht. Alle Anwendung, die in einer CLI-konformen Sprache geschrieben werden, werden zu CIL-Code übersetzt und in der Form von Assemblies in Portable Executable (PE)-Dateien gespeichert.

Eine Assembly ist eine konfigurierte Menge von Modulen und weiterer Ressourcen. Ein Modul ist eine vom VES ausführbare Einheit. Die Assembly selbst liefert Informationen für die Ausführung der enthaltenen Module. Typen werden nur im Kontext einer Assembly definiert. Der gesamte Ausführungsprozess bei der Common Language Infrastructure wird von dem Common Language Runtime (CLR) übernommen.

Microsoft .NET

.NET [Micc] ist die Implementierung der Common Language Infrastructure von Microsoft für das Windows Betriebssystem und andere Microsoft Produkte wie XBox 360 und Windows Phone.

.NET bietet Compiler für verschiedene Programmiersprachen, die alle zu CIL kompiliert werden, beispielsweise C#, Visual Basic.Net, C++ und F#.

C# ist eine generische, objektorientierte, typsichere Programmiersprache und ist die einzige .NET-Sprache die ECMA-standardisiert ist [Int06].

2.2.2 Visual Studio

Visual Studio 2010 [Mica] ist ein Werkzeug von Microsoft, welches Entwickler während des ganzen Entwicklungszyklus einer Anwendung unterstützen soll. Mit Visual Studio 2010 werden Anwendungen für Windows, Windows Phone, XBox 360, Web und Cloud entwickelt, in dem eine der .NET-basierten Programmiersprachen oder natives C++ eingesetzt wird.

Während des Übersetzungsprozesses von Visual Studio werden auch die sogenannten Programm Database Dateien (PDB) [Pdb] erstellt, die das Debuggen einer Anwendung ermöglichen. In diesen Dateien werden Zuordnungen gespeichert. Die genau benötigten Informationen werden in späteren Kapiteln genauer beschrieben. Einige davon sind:

1. Zuordnung IL/Native Instruktion zu Quellcodezeile
2. Namen von Variablen, Argumenten und weiterer Konstrukte
3. Pfade zu Quelltextdateien

Mithilfe von pdb-Dateien kann man feststellen, welche Instruktionen zu welcher Quelltextzeile gehören. Die Arbeit mit Program Database Dateien wird durch die im nächsten Abschnitt beschriebenen Common Compiler Infrastructure Frameworks realisiert.

2.2.3 Common Compiler Infrastructure

Common Compiler Infrastructure (CCI) [BFVa] ist eine Menge von Open Source Frameworks von Microsoft Research. Die Frameworks unterstützen das Auslesen, Editieren und Erzeugen von Metadaten und Intermediate Language Code von bereits kompilierten .NET Assemblies.

Die von CCI angebotene Funktionalität beinhaltet und erweitert die Funktionalität der folgenden .NET Frameworks und Werkzeugen:

1. Metadata APIs (zum Beispiel IMetaDataEmit)
2. System.Reflection und System.Reflection.Emit
3. System.CodeDom
4. Die .NET Werkzeuge ilasm (IL Assembler) und ildasm (IL Disassembler)

Common Compiler Infrastructure besteht aus zwei Teilen:

Common Compiler Infrastructure: Metadata

Common Compiler Infrastructure: Metadata [BFVc] ist der Teil der Frameworks, der den Umgang mit Metadaten implementiert. Zusätzlich werden auch das Erzeugen und das Editieren von IL-Code unterstützt. Methodenkörper werden von CCI Metadata als Liste von IL-Instruktionen dargestellt.

Common Compiler Infrastructure: Code und AST

Common Compiler Infrastructure: Code & AST [BFVb] bietet eine IL-Verarbeitung auf einer höheren Abstraktionsebene als CCI: Metadata. Der Entwickler hat bei diesem Framework die Möglichkeit mit sprachunabhängigen Konstrukten, wie zum Beispiel Blöcken und Statements zu arbeiten. Ziel dieses Projektes ist es, das Arbeiten mit IL-Code zu erleichtern und in eine Quellcode-ähnliche Form zu bringen.

2.2.4 Roslyn

Roslyn [NWGH] ist ein Microsoft Projekt, dessen Ziel es ist, das gesamte Quelltextwissen des Übersetzers und seine interne Funktionalität als APIs zur Verfügung zu stellen und so die Entwicklung externer Werkzeuge für Codeanalyse, Refaktorisierung und Formatieren zu erleichtern. Roslyn bietet folgende APIs an:

1. Compiler APIs: Stellen die gesamte Information, die jede Stufe des Übersetzungsprozesses generiert, zur Verfügung.
2. Scripting APIs: Bieten eine Ausführungsumgebung für C# und Visual Basic Code.
3. Workspace APIs: Startpunkt für Codeanalyse und Refaktorisierung von ganzen Projekten.
4. Services APIs: Enthält alle Visual Studio IDE Features, wie z.B. IntelliSense, Refaktorisierung und Formatierung. Enthält auch APIs, die eine Erweiterung der Entwicklungsumgebung ermöglichen.

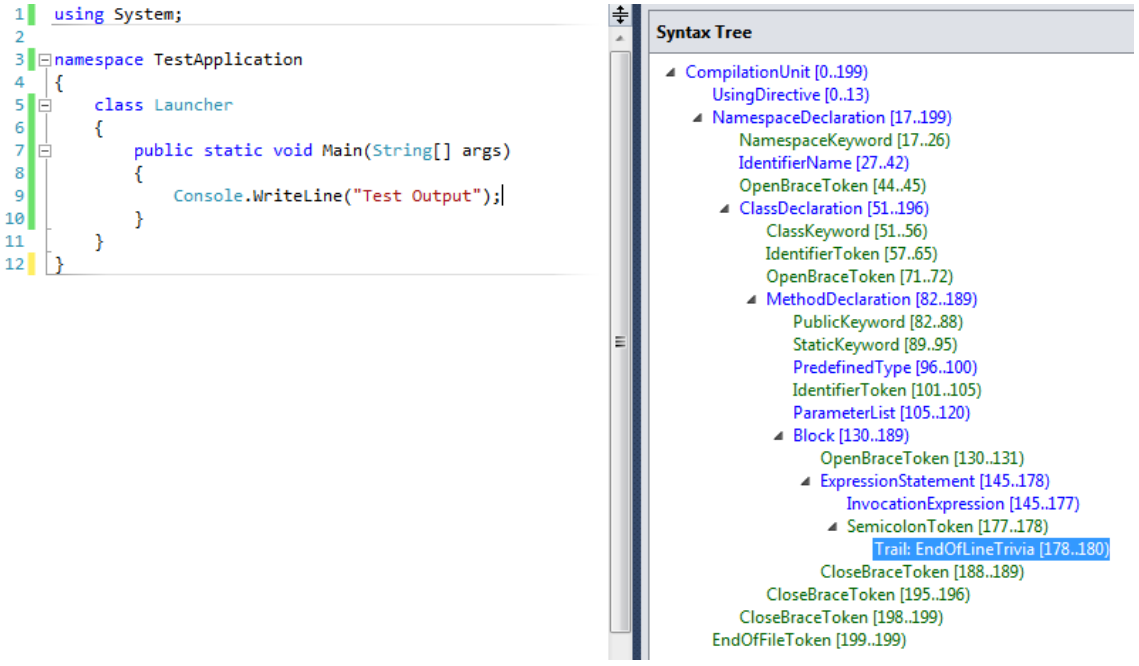

Abbildung 2.7: Syntaxbaum einer Queltextdatei in Roslyn

Die Compiler-APIs eines Roslyn CTP (Community Technology Preview) werden in dieser Diplomarbeit für das Parsen des Quelltextes und beim Erkennen von Anweisungen und Schleifen verwendet. Der nach dem Parsen entstehende Syntaxbaum bietet höheren Komfort und Sicherheit als das Arbeiten mit Klartext (siehe Abbildung 2.7).

2.2.5 Microsoft Research CHESS

Microsoft Research CHESS [BBM+], [MQB+08], [BBMQ08], [BBC+09] ist ein Werkzeug, das das Ausführen paralleler Komponententests unter Anwendung verschiedener Schedules ermöglicht.

Wenn ein Programm mit CHESS getestet wird, übernimmt CHESS die Kontrolle über das Thread-Scheduling, in dem es einen Wrapper über die verwendeten parallelen Bibliotheken bildet. So steuert CHESS die Ausführungsreihenfolge der einzelnen Threads. Unter Verwendung verschiedener Suchtechniken werden unterschiedliche Schedules erzeugt und der Code wird mit diesen getestet.

Chess kann Probleme im Code erkennen, die bei normaler Ausführung kaum zu erkennen sind: Hierzu gehören Datenwettläufe, Verklemmungen und Atomizitätsverletzungen. Die Schedules, die einen Fehler verursacht haben, werden aufgenommen und können später reproduziert werden. So kann man das Problem auch mehrmals beobachten und weitere Informationen gewinnen (Tracing): zum Beispiel Stack Trace, betroffene Methode, Quelltextzeile.

Eine leicht angepasste Version von CHESS wird im Rahmen der Diplomarbeit als Testumgebung zum Erkennen von Datenwettläufen verwendet. Die Trace-Ergebnisse wurden um zusätzlichen Stack Trace-Informationen ergänzt.

Concurrency Explorer und die grafische Oberfläche CHESS Board sind zwei Werkzeuge, die die Verwendung von CHESS erleichtern.

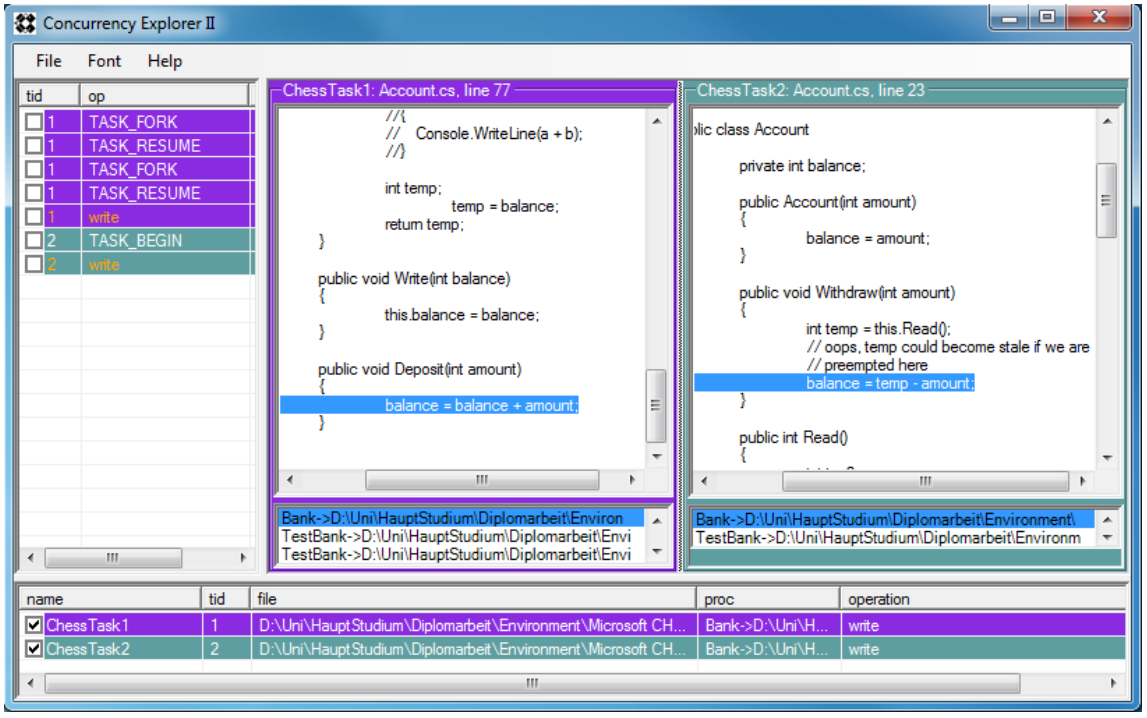

Abbildung 2.8: Ein Datenwettlauf in Concurrency Explorer

Concurrency Explorer

Concurrency Explorer kann Trace-Ergebnisse eines, mit CHESS ausgeführten Tests visualisieren. Wurde beispielsweise ein Datenwettlauf gefunden, so kann das Werkzeug die zwei Trace-Einträge bestimmen, die ihn verursacht haben (siehe Abbildung 2.8). In dieser Diplomarbeit wird für die primäre Ergebnisanalyse eine angepasste Version des Concurrency Explorers verwendet. Dafür wurde das Werkzeug in einer Bibliothek transformiert und die oben beschriebene Funktionalität wurde nach außen zur Verfügung gestellt.

3. Verwandte Arbeiten

In diesem Kapitel werden verwandte Arbeiten und Werkzeuge vorgestellt, die die automatische Erkennung und Korrektur von Datenwettläufen in parallelen Anwendungen zum Schwerpunkt haben.

3.1 Automatische Testgenerierung für parallele Anwendungen

Als Teil eines großen Projekts für automatische Parallelisierung sequenzieller Software und die automatische Qualitätssicherung der dabei entstandenen Anwendungen wird in diesem Kapitel auch eine weitere Diplomarbeit vorgestellt, deren Ausgabe als Eingabedaten für die vorliegende Arbeit verwendet werden kann.

Im Rahmen seiner Diplomarbeit "Automatische Testgenerierung für parallele .NET-Anwendungen" [Dim11] am IPD Tichy hat Filip Dimitrov ein Verfahren für das automatische Generieren von Testfällen zum Erkennen von Datenwettläufen in parallelen Programmen vorgestellt. Dabei wurde ein Testfallgenerator für .NET Anwendungen entwickelt, der Microsoft CHESS (siehe Abschnitt 2.2.5) verwendet, um die generierten Komponententests auszuführen und den Code auf Datenwettläufe zu durchsuchen.

Das Verfahren profitiert von einer mehrstufigen Softwareanalyse. Eine statische Voranalyse ermittelt anhand Datenabhängigkeiten die Methoden, in denen Datenwettläufe auftreten können. Danach wird anhand einer dynamischen Analyse bestimmt, welche von diesen Methoden parallel ausgeführt werden. Anschließend werden für diese Methoden relevante Variablenbelegungen und Programmzustände aufgezeichnet. Diese Werte werden dann als Eingabeparameter der generierten Testfälle verwendet.

3.2 Automatisches Erkennen von Wettlaufbedingungen

Microsoft CHESS ist das Werkzeug, das in dieser Arbeit zum Erkennen von Wettlaufbedingungen verwendet wird. Dieses wird in Abschnitt 2.2.5 beschrieben. In diesem Abschnitt werden einige Alternativen vorgestellt. Diese können in drei verschiedene Gruppen aufgeteilt werden:

3.2.1 Statische Werkzeuge

Als statische Erkenner werden Werkzeuge bezeichnet, die nur auf der Basis einer statischen Analyse auf Quelltext- und/oder auf Assemblyebene die Anwendung auf Fehler überprüfen. Das Programm wird nicht ausgeführt und Informationen, die nur zur Laufzeit existieren, liegen nicht vor: zum Beispiel Eingabedaten. Schwierig und eingeschränkt ist die Analyse von Referenzen und dynamischen Methodenaufrufen. Statisch kann aber die ganze Anwendung und nicht nur der gerade ausgeführte Ablaufpfad analysiert werden. Diese Werkzeuge sind oft in der Lage viele Fehler zu erkennen, liefern aber auch eine große Anzahl an falsch positiven Ergebnissen. In den nächsten Abschnitten werden einige statische Datenwettlauferkenner vorgestellt.

RacerX

RacerX [EA03] ist ein statischer Verklemmungs- und Datenwettlaufdetektor, der von Engler und Ashcraft am Stanford University entwickelt wurde. Das Werkzeug ist speziell für komplexe parallele Systeme aufgebaut. RacerX verwendet starkes Inferieren auf der Basis von Informationen wie:

- welche Sperre welche Operationen schützt
- welche Codeteile werden parallel ausgeführt
- welche Zugriffe auf gemeinsame Daten sind gefährlich

Jlint

Jlint [Kni] ist ein Werkzeug zum automatischen Erkennen von Fehlern in Java-Programmen auf der Basis von statischer Daten- und Kontrollflussanalyse sowie dem dabei entstandenen Sperrengraphen. Zu den erkannten Fehlertypen gehören auch Synchronisationsprobleme wie Verklemmungen und Wettlaufbedingungen. Jlint wurde ursprünglich von Konstantin Knizhnik an der Moscow State University entwickelt und ist frei verfügbar.

Extended Static Checker for Java

Extended Static Checker for Java (ESC/Java) [FLL+02] ist ein Werkzeug, das Java-Anwendungen zur Übersetzungszeit auf Fehler überprüft. ESC/Java ist auf Generierung von Verifikationsbedingungen und Techniken zum Beweisen von Theoremen aufgebaut. Der Entwickler verwendet eine Annotationssprache, um Entwurfsentscheidungen formal zu definieren. ESC/Java vergleicht die Annotationen mit dem implementierten Code und warnt, wenn Unterschiede erkannt sind. Das Werkzeug kann auch Datenwettläufe und Verklemmungen erkennen. ESC/Java wurde am Compaq Systems Research Center entwickelt.

3.2.2 Dynamische Werkzeuge

Dynamische Datenwettlaufdetektoren sind Werkzeuge, die zur Laufzeit das Verhalten einer Anwendung analysieren und nach Fehlern suchen. Die getestete Anwendung wird instrumentiert, um Speicherzugriffe und Synchronisationsinformationen zu gewinnen. Problematisch bei dynamischer Analyse ist die Codeabdeckung. Es wird nur der ausgeführte Ablaufpfad analysiert, was zu suboptimalen Ergebnissen führt. Zusätzlich verändert die Instrumentierung das Programm und sorgt für eine langsamere Ausführung. In den nächsten Abschnitten werden einige dynamische Datenwettlauferdetektoren vorgestellt.

Coverity Static Analysis and Coverity Dynamic Analysis

Coverity Static Analysis [Inc11b] und Coverity Dynamic Analysis [Inc11a] sind Softwareprodukte der Firma Coverity Inc. Beide Werkzeuge unterstützten u. a. auch automatisches Erkennen von Wettlaufbedingungen und Verklemmungen, in dem entsprechend statische und dynamische Analysen verwendet werden. Coverity Static Analysis kann C/C++, C# und Java-Code analysieren, Coverity Dynamic Analysis ist momentan auf Java-Programme eingeschränkt.

Eraser

Eraser [SBN+97] ist ein dynamischer Datenwettlauferkenner für sperrenbasierte parallele Anwendungen. Dieses Werkzeug verwendet Codeinstrumentierung, um jeden Zugriff auf gemeinsamem Speicher und das Verhalten beim Akquirieren und Freigeben von Sperren zur Laufzeit aufzuzeichnen. Wettlaufbedingungen werden anhand des in [SBN+97] definierten Lockset-Algorithmus erkannt.

Helgrind und Helgrind+

Helgrind [Devb] ist ein dynamischer Erkenner von Synchronisationsfehlern für C/C++ und Fortran Programme, die die POSIX pthreads-Bibliothek verwenden. Der implementierte Algorithmus zum Erkennen von Datenwettläufen basiert auf der Happens-Before-Relation. Helgrind ist Teil von Valgrind [NS07], ein Instrumentierungsrahmenswerk zum Implementieren von dynamischen Werkzeugen für Codeanalyse.

Helgrind+ [JBPT09] ist ein experimenteller Datenwettlauferkenner, der auf der Basis von Helgrind entwickelt worden ist und die Lockset- und Happens-Before-Algorithmen kombiniert. Unter den neuen Features sind Techniken zum Reduzieren der falsch gemeldeten Datenwettläufe sowie das Erkennen von Wettlaufbedingungen unabhängig von den verwendeten Synchronisationsbibliotheken.

3.2.3 Testbasierte Werkzeuge

Testbasierte Werkzeuge versuchen, anhand von Komponententests den Code auf Fehler zu überprüfen. Die Komponententests werden mehrmals unter verschiedenen Schedules ausgeführt. Dabei wird eine dynamische Analyse durchgeführt, die Datenwettläufe und andere Fehlertypen erkennt. Eins dieser Werkzeuge ist Microsoft CHESS. Andere werden in den nachfolgenden Abschnitten vorgestellt:

Relacy Race Detector

Relacy Race Detector (RRD) [Vyu], [Kar09] ist ein Werkzeug zur Verifikation paralleler C++-Anwendungen auf der Basis von Komponententests. RRD wird von Dmitriy Vyukov entwickelt, mit dem Ziel Synchronisationsalgorithmen auf parallele Probleme zu überprüfen. Das Werkzeug kontrolliert den Ablaufplan während der Ausführung der Komponententests. Hierfür muss die Codeparallelisierung anhand der RRD-eigenen Wrapper-APIs erfolgen. RRD erkennt folgende Fehlertypen:

- Wettlaufbedingungen
- Zugriff auf nicht-inizialisierten Variablen
- Zugriff auf freigegebenen Speicher
- Speicherleck
- Verklemmungen
- Live Locks
- Fehler bei benutzerdefinierten Asserts und Invarianten

Concurrent Testing

Concurrent Testing (ConTest) [EFN+02b, EFN+02a, EFN+03] ist ein von der IBM Forschungs- und Entwicklungscommunity "alphaWorks" entwickeltes Werkzeug zum mehrfachen Ausführen von Java-Anwendungen und dafür geschriebener Tests. ConTest instrumentiert den Java-Bytecode mit heuristisch-kontrollierten sleep() und yield() Instruktionen, die den Ablaufplan beeinflussen. Somit wird versucht, die Wahrscheinlichkeit zum Auftreten von Synchronisationsproblemen, wie Datenwettläufe und Verklemmungen, zu erhöhen. ConTest ist ein offenes System. Es können Plugins geschrieben werden, die in den Bereichen Testen, Debuggen oder Vermeidung von Fehlern neue Funktionalität implementieren.

Unit Testing for Multi-Threaded Java Programs

In [Sze09] wird ein Rahmenwerk zur Ausführung paralleler Komponententests für Java-Anwendungen vorgestellt. Programmfehler werden erkannt, in dem alle möglichen Ausführungspfade eines Komponententests anhand Techniken zur Modellüberprüfung untersucht werden. Ein Test wird als bestanden markiert, wenn in keinem Ablaufplan Fehler entdeckt werden. Das Rahmenwerk ist auf der Basis von JUnit [JUn] und dem Modellprüfer Java PathFinder [Pat, VHBP00] implementiert.

3.3 Korrektur und Vermeidung von Datenwettäufen

In diesem Abschnitt werden Arbeiten vorgestellt, die verschiedene Ansätze zum Verhindern von Datenwettläufen beschreiben. Diese Arbeiten teilen sich in zwei Gruppen. Zu der ersten Gruppe gehören Arbeiten, die den Quelltext der Anwendung verändern, um die Datenwettläufe zu entfernen. Die vorliegende Diplomarbeit gehört auch dazu. Zu der zweiten Gruppe gehören Arbeiten, die versuchen, das Auftreten von Datenwettläufen zur Laufzeit zu vermeiden.

3.3.1 Automatische Quellcodekorrektur von Wettlaufsituationen

Im Rahmen seiner Diplomarbeit "Automatische Quellcodekorrektur von Wettlaufsituationen in parallelen Programmen" [Bon11] am IPD Tichy hat Felix Bondarenko ein Verfahren erarbeitet, das für mit Helgrind erkannte Datenwettläufe in C-Programmen Lösungsvorschläge generiert und im Quellcode einbaut. Dabei werden Lese-und Schreibzugriffe, bedingte Verzweigungen und Schleifen berücksichtigt.

Viele der verwendeten Konzepte werden in ähnlicher Form für diese Diplomarbeit übernommen. Obwohl sich aber die Themen der beiden Arbeiten teilweise überdecken, existieren zahlreiche Unterschiede, Erweiterungen und Verbesserungen. Die Wichtigsten sind in der Tabelle in Abbildung 3.1 aufgezählt.

3.3.2 Automatische Synchronisationskorrektur

In "Automatic Synchronization Correction" [FF05] präsentieren Cormac Flanagan und Stephen N. Freud ihren Ansatz zur Korrektur von Synchronisationsfehlern anhand zusätzlicher Synchronisationsblöcke. Es wird eine Rahmenbedingung erstellt, die die Relation zwischen eine definierte Methodenatomizität (zum Beispiel `atomic`) und einen auf der Basis des Methodenkörpers generierten Atomizitatsausdruck darstellt. Der Ansatz inferiert die zusätzliche Synchronisation, in dem die Rahmenbedingung aufgelöst wird. Das beschriebene Verfahren verwendet die Sperreninferenz, die in [FF07] vorgestellt wird, um die notwendigen Sperren zu bestimmen. Der Ansatz ignoriert die beim Einbau zusätzlicher Synchronisationsblöcke bestehende Verklemmungsgefahr. Tabelle 3.1 beschreibt die Unterschiede zwischen den beiden Arbeiten.

Konzept	Bondarenko [Bon11]	Flanagan und Freud [FF05]	Diese Arbeit
Allgemeine Codeannahme	Betrachtet den Code als eine isolierte Einheit die einen eindeutigen Startpunkt hat und mit dem Rest der Welt nicht interagiert.	Arbeitet nach vorgegebenen oder geschätzten Standardvorgaben für die Methodenatomizität.	Betrachtet den Code als Teil eines komplexen Systems. Alle öffentlichen Methoden werden als Startpunkte interpretiert, die parallel aufgerufen werden können.
Bestimmen der verwendeten Sperren.	Unterstützt Wiederverwenden von Sperren durch eine Sperreninferenz.	Unterstützt Wiederverwenden von Sperren durch eine Sperreninferenz.	Unterstützt Wiederverwenden von Sperren nur durch Codeannotationen. Der Entwickler kann durch die Codeannotation auch Sperren definieren.
Korrektur von Datenwettläufen im Programmcode durch:	Explizites Aufrufen der Methoden zum Akquirieren und Freigeben von Sperren.	Synchronisationsblöcke. Sperrenfreigabe bei Laufzeitfehlern wird nicht diskutiert, diese ist aber für Synchronisationsblöcke der Standard. Das Verfahren reagiert nicht auf gemeldete Datenwettläufe, sondern korrigiert die Synchronisation, um eine vorgegebene oder geschätzte Methodenatomizität zu erfüllen.	Einsatz eines Synchronisationsblocks, der eine Freigabe der Sperre auch bei geworfenem Laufzeitfehler garantiert. Wird die Sperre nicht freigegeben, so können Probleme wie Verklemmungen entstehen. Korrekturmuster, die passend für die LOCK-Anweisung sind.
Korrektur von Datenwettläufen im externen Code durch:	Keine	Keine	Unter der Annahme, dass der Datenwettlauf vom eigenen Code korrigierbar ist. Im allgemeinen Fall durch Synchronisation mit Lock-Blöcken. Für Datenwettläufe in Thread-unsicheren Datenstrukturen wird ein Austausch gegen eine Thread-sichere Datenstruktur vorgestellt.
Korrekturvorschläge	Generiert eine Liste zahlreicher Korrekturvarianten. Der Benutzer muss danach die Entscheidung treffen, welche davon im Code eingebaut wird. Die Anzahl der Vorschläge kann auch bei wenigen Datenwettläufen im schlimmsten Fall exponentiell wachsen	Generiert eine Lösung, in dem eine generierte Rahmenbedingung aufgelöst wird. Die Rahmenbedingung ist die Relation zwischen der erwarteten und der im Methodenkörper vorhandenen Atomizität.	Generiert eine Lösung. Eine Kontroll- und Datenflussanalyse und das vorgestellte Stopper-Konzept entscheiden zum Beispiel ob mehrere Zugriffe auf dieselbe Variable zusammen oder getrennt geschützt werden. Weitere Entscheidungen werden entweder direkt gelöst oder für weiterführende Arbeiten gelassen.
Codeanalyse	Verwendet nur Quelltext.	Verwendet nur Quelltext.	Verwendet sowohl Quelltext als auch nativen Code, um die Vorteile der beiden Programmdarstellungen zu kombinieren. Beispiel: keine gesonderte Betrachtung von Parameterübergabe an Methodenaufrufe. Auf nativer Ebene ist sie als Leseoperation vorhanden.
Behandlung von Verklemmungen	Es wird eine Sperrenordnung verwendet, um die Verklemmungsgefahr zu minimieren.	Die Verklemmungsgefahr wird ignoriert und für weiterführende Arbeiten gelassen.	Es wird eine Anpassung (Expandieren) der geschützten Regionen, sowie eine Sperrenordnung verwendet, um die Verklemmungsgefahr zu minimieren.

Abbildung 3.1: Liste der größeren Unterschiede zwischen der Diplomarbeit "Automatische Quellcodekorrektur von Wettlaufsituationen in parallelen Programmen" [Bon11], "Automatic Synchronization Correction" [FF05] und der vorliegende Diplomarbeit.

3.3.3 GenProg: Genetische Software-Reparatur

Genetic Program Repair (GenProg) [LGNFW12, LGDVFW12] ist eine automatisierte Methode zur Korrektur von Legacy-Anwendungen ohne formale Spezifikation und Programmanotationen. GenProg verwendet eine erweiterte Form der genetischen Programmierung, um eine Programmvariation zu entwickeln, die die Funktionalität des Originals behält, aber ein vorgegebenes Problem nicht. Dabei werden Software-Tests verwendet, um nicht nur die Probleme zu erkennen, sondern auch Informationen über die Funktionalität der Anwendung zu gewinnen. GenProg unterstützt zurzeit keine Korrektur von Datenwettläufen, ist aber erfolgreich in der Reparatur paralleler Anwendungen mit voneinander unabhängigen Threads.

In den nachfolgenden Abschnitten werden Arbeiten und Werkzeuge vorgestellt, die sich mit der Vermeidung von Datenwettläufen zur Laufzeit beschäftigen. Diese Ansätze stehen im Gegensatz zur in dieser Arbeit eingesetzten quelltextbasierten Datenwettlaufskorrektur auf der Basis von Synchronisationsblöcken und Austausch von Datenstrukturen.

3.3.4 Selbstheilen zur Laufzeit

In "Automatic Synchronization Correction" [KLT+07] werden die ersten Schritte eines Ansatzes zum Selbstheilen von Datenwettläufen auf der Basis des in Abschnitt 3.2.3 vorgestellten Werkzeugs ConTest vorgestellt. Es werden drei Mechanismen vorgeschlagen:

1. Ändern der Wahrscheinlichkeit für Thread-Verschränkung mithilfe von `sleep`, `yield`-Anweisungen und Prioritäten.
2. Entfernen der Thread-Verschränkung mithilfe von Synchronisation.
3. Korrektur der Ergebnisse einer schlechten Thread-Verschränkung.

Die drei Mechanismen werden mittels einer Instrumentierung und Beobachtung der laufenden Anwendung realisiert.

3.3.5 TachoRace

TachoRace [Sch10] ist ein Datenwettlauferkenner, der zur Laufzeit Wettlaufbedingungen erkennt und automatisch vermeidet. Dieses Verfahren beobachtet, mithilfe der von den Hardware-Performanzzählern gesammelten Information, die Kommunikation zwischen den Primär-Caches der Prozessorkerne, um ein leicht gewichtetes Erkennen zu implementieren. Zwei verschiedene Strategien werden angeboten. Eine für speziell annotierte Anwendungen und eine weniger präzise für Standardanwendungen.

Datenwettläufe werden durch eine Verzögerung der Thread-Ausführung und somit des Datenzugriffes vermieden. Die Ausführungsverzögerung wurde mit einer Erweiterung des MESI-Protokols [Che08] um die `RaceWait` und `RaceContinue` Interrupt-Nachrichten erreicht. Bei `RaceWait` blockiert der Prozessorkern die Ausführung des Threads bis eine `RaceContinue`-Nachricht ankommt.

3.3.6 Erkennen und Tolerieren asymmetrischer Datenwettläufe

In "Detecting and tolerating asymmetric races" [RBK+09] wird das Werkzeug ToleRace vorgestellt. ToleRace ist ein Laufzeitsystem, das anderer Anwendungen ermöglicht, asymmetrische Datenwettläufe zu erkennen und zu tolerieren. Asymmetrisch sind Datenwettläufe, die zwischen ein korrekt- und ein nichtgeschützter Codebereich entstehen. ToleRace isoliert die durch Sperren geschützte kritische Region, in dem lokale Kopien aller gemeinsamen Variablen beim Eintritt in die Region erstellt werden.

3.3.7 Vermeidung von Datenwettläufen durch Sichten

In "Maotai 2.0: Data Race Prevention in View-Oriented Parallel Programming" [LHHW09] wird ein Ansatz zur Vermeidung von Datenwettläufen vorgestellt, der auf das View-Oriented Parallel Programming-Modell (VOPP) basiert. VOPP verwendet Sichten, um Speicherzugriffe mit Mutexe (wechselseitiger Ausschluss) zu kombinieren. Die Daten werden in verschiedene Sichten verteilt. Ein Thread darf zu den Daten aus einer Sicht erst dann zugreifen, wenn er die Sicht akquiriert hat. Zusätzliche Features wie automatische Verklemmungsvermeidung, Erzeuger-Verbraucher Sicht und andere sind in der VOPP-Implementierung Maotai 2.0 enthalten.

3.4 Symbolische Ausführung

Bei einer symbolischen Ausführung [Kin76, BEL75, HSS09] einer Anwendung werden echte Eingabedaten durch Symbole ausgetauscht. Während der Ausführung werden Programmwerte durch Formeln auf die Eingabesymbole repräsentiert. Das erlaubt die Ausführung aller Kontrollpfade im Programm und das Sammeln von Informationen über die Eingabewerte. Die Anwendung wird bei dieser Analyse interpretiert. Symbolische Ausführung wird oftmals bei der Generierung von Testdaten verwendet.

Die in dieser Arbeit vorgestellte simulierte Ausführung (siehe Abschnitt 4.9.2) adaptiert viele Eigenschaften der symbolischen Ausführung um unter anderem das Verfolgen von Objekten im Code zu ermöglichen. Das erlaubt unter anderem das Bestimmen des genauen Typs und der Erstellungsquellen eines Referenzobjekts.

4. Entwurf

Dieses Kapitel erläutert das im Rahmen dieser Arbeit entwickelte Verfahren zum automatischen Generieren von Lösungensvorschlägen für Wettlaufsituationen und deren Einbau im Quelltext einer Anwendung. Ziel ist es, ein vollständig automatisiertes und benutzerfreundliches Verfahren zu definieren.

Struktur des Kapitels

Im nächsten Abschnitt werden zunächst die Zielsetzung, die Anforderungen und die Einschränkungen des Verfahrens definiert. Danach wird in Abschnitt 4.2 ein Überblick über das Verfahren geschaffen und die notwendigen Ausgangsinformationen werden vorgestellt. Abschnitt 4.3 beschäftigt sich mit der verfahrenseigenen Programmabbildung (Code Object Model), auf deren Basis die Korrekturvorschläge generiert werden. Anschließend werden in Abschnitt 4.4 die Analyse und die Verarbeitung der Testergebnisse erläutert. Als Nächstes wird in Abschnitt 4.5 die Auswahl der notwendigen Sperren erklärt. Dabei werden die Themen Wiederverwendung und Erstellung neuer Sperren detailliert beschrieben. In Abschnitte 4.6, 4.7, 4.8 und 4.9 wird der Aufbau von einigen Hilfsstrukturen vorgestellt. Diese liefern Informationen über die Methodenaufrufe, die Sperren, den Kontroll- und den Datenfluss in der Anwendung. Abschnitt 4.10 stellt die Analyse der Fehlerstellen und deren Schützen, mithilfe von Lock-Blöcken und dem Austausch von Datenstrukturen, vor. Bevor die Korrekturen in Abschnitt 4.13 im Code eingebaut werden, werden in Abschnitt 4.11 die zu schützenden Regionen in ihrer finalen Form gebracht und in Abschnitt 4.12 die Ordnung der Sperren definiert. Zum Schluss wird das Kapitel in Abschnitt 4.14 zusammengefasst.

4.1 Zielsetzung und Anforderungen

Ziel dieser Arbeit ist das Entwickeln eines automatischen Verfahrens zum Erzeugen von Korrekturvorschlägen für Wettlaufbedingungen im parallelen und getesteten Code und ihren Einbau im Quelltext der Anwendung. Die konkreten Anforderungen sind folgendermaßen definiert:

A1: Das Verfahren funktioniert unabhängig von der Testumgebung, die die Datenwettläufe erkennt. Eine Spezifikation der notwendigen Informationen wird zur Verfügung gestellt.

A2: Wettlaufsituationen im Programmcode werden mittels Sperren geschützt.

A2.1: Es muss garantiert werden, dass jede akquirierte Sperre auch freigegeben wird.

A2.2: Die Korrektur verursacht keine Verklemmungen.

A2.3: Der Entwickler hat die Möglichkeit eigene Sperren zu definieren, die vom Verfahren bei der Korrektur benutzt werden können. In diesem Fall wird das Werkzeug keine automatisch generierten neuen Sperren verwenden, sondern diese, die vom Entwickler für das Schützen der bestimmten Objekte vorgesehen sind. Das gibt dem Entwickler die Möglichkeit die Korrektur von außen zu beeinflussen.

A3: Wettlaufsituationen in externen Bibliotheken werden im Programmcode durch Lock-Blöcke oder Austausch von Datenstrukturen korrigiert.

A4: Voneinander abhängige Zugriffe auf dieselbe Variable werden zusammen geschützt, um eine vom Datenwettlauf verursachte Atomizitätsverletzung zu vermeiden. Mehrere Lese- und Schreibzugriffe auf dieselbe Variable können in einem Codebereich auftreten. Diese können Teil derselben Anweisung sein (`x++;`, `x = x + 1;`) aber auch von mehreren Zeilen und sogar Methodenaufrufen getrennt werden. Die Existenz eines Datenwettlaufes auf die Variable kann die Abhängigkeit brechen und die Atomizität des Bereiches verletzen. Das Zasammenschützen abhängiger Variablenzugriffe vermeidet dieses Problem.

A5: Bedingte Sprünge und Schleifen werden bei der Korrektur mit beachtet. Sie unterscheiden sich in ihrer Funktionalität und Quelltextdarstellung von den einfachen Anweisungen und steuern den Kontroll- und den Datenfluss im Code. Aus diesem Grund müssen sie gesondert betrachtet werden. Bei der Mutation des Quelltextes müssen bedingte Sprünge und Schleifen auch mitbetrachtet werden, um eine fehlerfreie Änderung zu erzeugen. Zum Beispiel kann die Bedingung einer IF-Anweisung nicht direkt mit einem Synchronizationsblock geschützt werden.

A6: Es werden möglichst wenige und kleine Codestellen geschützt.

A7: Eine Stelle wird maximal einmal von derselben Sperre geschützt.

A8: Die errechneten Lösungsvorschläge werden im Quelltext eingebaut.

Einschränkungen

Die Konzepte, die in diesem Kapitel entwickelt werden, werden unter den folgenden Einschränkungen definiert:

E1: Der Code, der getestet und korrigiert wird, enthält keine Verklemmungen: Während der Analyse wird die Ordnung der Sperren betrachtet. Ein Zyklus im Sperrgraphen deutet darauf hin, dass eine mögliche Verklemmung im Code existiert. Die Zyklusfreiheit des Graphen nach der Bestimmung der geschützten Regionen und der neuen Sperren wird als ein Hinweis auf eine Verklemmungsfreiheit des generierten Codes erkannt. Diese Voraussetzung limitiert die Anzahl der Anwendungen, die mit dem Verfahren korrigiert werden können.

E2: Alle vorhandenen Datenwettläufe werden auch erkannt: Das Verfahren kann nur Datenwettläufe korrigieren, die gemeldet werden. Zwischen gemeldeten Fehlerstellen können Abhängigkeiten existieren. Das Verfahren kann aber bereits eingebaute Korrekturen nicht editieren und setzt deswegen voraus, dass alle existierenden Datenwettläufe erkannt und am Anfang übergeben werden. Das iterative Vorgehen kann zu mehrfachem Schützen von Regionen führen. Weiterführende Arbeiten können das Verfahren um das Anpassen und Editieren bereits vorhandener Synchronisierung erweitern, um diese Einschränkung zu überwinden.

E3: Eine Fehlerstelle verursacht Code-Änderung nur in ihrer Methode. Diese Einschränkung wird eingeführt, um eine Grenze für die Korrekturmöglichkeiten zu setzen. Das Auslagern der Synchronisationsblöcke in Aufrufermethoden kann zum Beispiel eine Kettenreaktion bei der Korrektur verursachen. Eine weiterführende Arbeit kann den Fall untersuchen, bei dem diese Einschränkung nicht existiert.

E4: Virtuelle Methoden werden nicht gesondert betrachtet. In Abschnitt 7.2 wird einen Vorschlag gemacht, wie man während der Analyse virtuelle Methoden bearbeiten könnte. Dieses Thema wird aber in der Arbeit nicht betrachtet.

Weitere kleine Einschränkungen werden in den einzelnen Abschnitten definiert.

4.2 Das Verfahren im Überblick

Die folgende Liste erläutert die einzelnen Schritte des Verfahrens und ihre Beziehungen:

- Das Verfahren bekommt von der Testumgebung (in dieser Arbeit Microsoft CHESS 2.2.5) eine Liste mit Datenwettläufen zwischen jeweils zwei Stellen im Code.
- Auf Basis vom Quelltext und Maschinencode wird ein eigenes Modell der Anwendung, das sogenannte Code Object Model, erstellt. Dieses bietet gleichzeitig eine Abstraktion von dem Quelltext und der Maschinendarstellung, eine 1-zu-1 Abbildung zwischen deren Elementen und eigene Objekte zur Codeanalyse. Ziel des Code Object Models ist es, die Bereitstellung einer einheitlichen Informationsquelle für die weitere Analyse.
- Die Testergebnisse werden analysiert, im Code lokalisiert und in eine passende Form gebracht. Danach werden diese nach dem betroffenen Objekt in Datenwettlaufgruppen eingeordnet.
- Die Sperren für alle Datenwettläufe werden bestimmt. Existiert keine Sperre für eine Datenwettlaufgruppe, dann wird eine neue erstellt und so annotiert, dass das Verfahren diese auch in weiteren Iterationen erkennt.
- Das Code Object Model wird analysiert, um notwendige Informationen über die Methodenaufrufe, die Sperrenbeziehungen, den Daten- und den Kontrollfluss zu gewinnen. Diese werden im nächsten Schritt verwendet, um die Korrekturvorschläge zu generieren.
- Ein Korrekturvorschlag wird für jede gemeldete Fehlerstelle generiert. Während des Korrekturaufbaus werden bedingte Anweisungen, Schleifen, Metodenaufrufe, Kontrollflüsse und Datenabhängigkeiten betrachtet. Die Korrektur erfolgt nach dem vorliegenden Fall, mittels Einbau von Lock-Blöcken und Austausch von Datenstrukturen.
- Die Regionen, die mit Lock-Blöcken zu schützen sind, werden aufgebaut und evaluiert. Überlappende und geschachtelte Regionen werden behandelt.
- Eine Sperrenordnung wird für die Fälle definiert, bei denen mehrere Lock-Blöcke dieselbe Stelle schützen. Danach wird anhand des Sperrgraphen geprüft, ob die vorbereiteten Korrekturen eine Verklemmung verursachen können.
- Alle Korrekturen werden im Quelltext eingebaut und der resultierende Code wird auf Fehlerfreiheit anhand der Testumgebung überprüft.

Im Rest des Kapitels werden alle diesen Einzelschritte detailliert und mit Beispielen beschrieben.

Ausgangsinformationen

Damit das vorgestellte Verfahren korrekt funktioniert, werden die folgenden Informationen und Daten benötigt:

- übersetzbarer Quelltext.
- Beschreibung der Datenwettläufe. Diese Beschreibung kommt von der Testumgebung (siehe Abschnitt 5.4), kann aber auch manuell erstellt werden, und enthält folgende Informationen:
  - Bibliothek
  - Typ mit entsprechendem Namespace oder Paket
  - Vollqualifizierter Methodenname, mit Beschreibung der Parameter, damit eine Unterscheidung zwischen überladenen Methoden möglich ist.
  - Abstand der Instruktion, vom Methodenanfang. Anhand der Instruktion kann danach auch die genaue Fehlerstelle im Quelltext identifiziert werden.

Zusätzlich sind Informationen wie zum Beispiel Art des Zugriffes auch hilfreich aber nicht unbedingt notwendig, weil diese anhand der Instruktion auch später bestimmt werden können.

4.3 Code Object Model

Ist in den Ausgangsdaten mindestens eine Wettlaufsituation beschrieben, kann das Verfahren gestartet werden. Diese benötigt:

1. den Quelltext, der unter Anderen für Folgendes notwendig ist:
   - Erkennen und Betrachten von bedingten Sprüngen und Schleifen. [A5] Die Information, die die Maschinendarstellung und die vorhandene Debug-Information (in .NET liegt diese in der pdb-Datei) reichen nicht aus, um u. a. die `FOR`-Schleife von der `WHILE`-Schleife zu unterscheiden.
   - Einbau der Korrekturen im Quelltext. [A8]
2. die Maschinendarstellung, die unter Anderen für Folgendes verwendet wird:
   - Lokalisieren der Datenwettläufe und Erkennen des betroffenen Feldes.
   - Wegen ihrer Eindeutigkeit ist die Maschinendarstellung in diesem Verfahren für die simulierte Ausführung des Codes bevorzugt. Diese ist Bestandteil einer nötigen Kontroll- und Datenflussanalyse, die eine Voraussetzung für [A4] ist.

Viele Teile des Verfahrens nutzen diese zwei Informationsquellen gleichzeitig, um ihre Aufgaben zu erledigen. Aus diesem Grund ist eine eindeutige Abbildung zwischen Quelltext und Maschinencode mit einer einheitlichen Zugriffsschnittstelle notwendig.

Es wurde kein Werkzeug gefunden, das diese Aufgabe erledigt.

Das Common Compiler Infrastructure 2.2.3 bietet einen Decompiler an. Der erzeugte Quelltext weicht aber stark vom echten Quelltext ab, weil dieser direkt vom Maschinencode abgeleitet ist. Aus diesem Grund kann er nicht für das Speichern der Ergebnisse verwendet werden, sondern nur für die Arbeit mit dem Assembly-Code. Wäre der vom Decompiler erzeugte Code eine perfekte Kopie des Quelltextes, so könnte man sich den Aufbau des Code Object Models sparen.

Das speziell entwickelte Code Object Model wird auf Basis des Quelltexts und der Maschineninstruktionen aufgebaut. Es bietet eigene Abstraktionsobjekte, die nah an der Quelltextstruktur liegen, trotzdem aber Zugriff auf die unterste Maschinenebene erlauben. Dabei entsteht eine eindeutige Abbildung zwischen Maschineninstruktionen und Quelltext (siehe Abbildung 4.1).

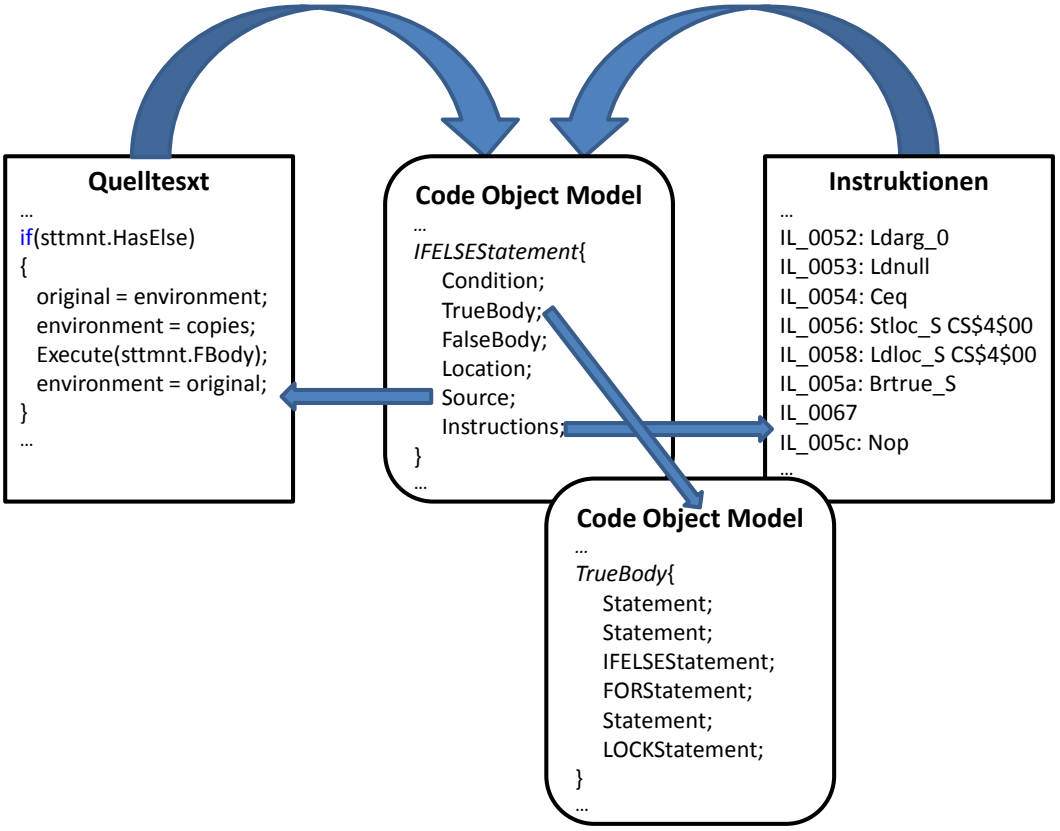

Abbildung 4.1: Code Object Model: Informationsfluss und Bestandteile.

Das Code Object Model ist eine Abstraktion über den tatsächlichen Quelltext und die Maschinendarstellung. Im weiteren Verlauf dieser Arbeit beziehen sich die Verwendung von Begriffen wie Schleifen oder bedingte Anweisungen auf die Abstraktionsobjekte des Code Object Models.

4.3.1 Aufbau und Bestandteile des Code Object Model

Es gibt zwei Möglichkeiten das Modell aufzubauen:

- Zuerst wird der Quelltext geparst. Dann wird nach den entsprechenden Instruktionen für jedes Quelltextelement gesucht. So können Teile der Anwendung, die keinen entsprechenden Quelltext haben, nicht oder nur schwer betrachtet werden. Das führt später zu Ungenauigkeiten bei der Kontroll-und Datenflussanalyse und somit zu suboptimalen Korrekturen.
- Finde zu jeder Instruktion den entsprechenden Quelltextteil. Instruktionen, die zusammengehören, werden zu Anweisungen kombiniert. Bei diesem Vorgehen ist ein tieferes Verständnis über den internen Aufbau der Maschinensprache und des Übersetzers notwendig. Nur so können zum Beispiel quelltextlose Instruktionen korrekt bearbeitet werden. In vielen Fällen erzeugen die Übersetzer Code, der für die Entwickler transparent bleibt. Zusätzlich können Teile des Codes vom Compiler weg-

optimiert werden. Gute Voraussetzungen für dieses Verfahren sind das Vermeiden nicht erreichbarer Quelltextteile und das Übersetzen in Debug Modus.

Das zweite Vorgehen wurde während dieser Arbeit angewandt. Dabei mussten viele technologiespezifische Probleme gelöst werden. Diese sind detailliert in Kapitel 5.5 beschrieben.

Die errechneten Korrekturen der Fehlerstellen werden im Quelltext eingebaut. Das macht ihn nicht nur zur Informationsquelle, sondern auch Speicherungsziel. Aus diesem Grund ähnelt das Code Object Model in seiner Code-Darstellung und Aufbau dem Quelltext. Es werden Highlevel-Objekte angeboten, die Quelltextanweisungen (engl. Statements) abbilden. Diese Objekte machen den Zugriff auf die unterste Instruktionsebene strukturierter und einfacher.

Die Statements sind im Code Object Model die Entsprechung der Anweisungen, die die high-level Sprache (in dieser Arbeit C#) definiert:

- Einfache Anweisungen:
  - Variablenzuweisungen
  - Methodenaufrufe
  - Instruktionen ohne dazugehörigen Quellcode
- Komplexe, zusammengesetzte Anweisungen: Das sind Anweisungen, die mehrere anderen Anweisungen enthalten (siehe Abschnitt 4.3.2):

Diese sind im COM-Kontext von einer Menge von nativen Instruktionen aufgebaut. Nach der Bearbeitung des Methodenkörpers ist seine COM-Repräsentation ein Baum, in dem zusammengesetzte Statements die Knoten und einfache Statements die Blätter sind.

Jede Anweisung enthält die folgenden Informationen:

- Liste der Instruktionen, die das Statement aufbauen.
- Quelltext, falls vorhanden.
- genaue Position der Anweisung im Quelltext.
- Typ des einfachen Statements:
  - Sprungbedingung für IF, IF/ELSE und für die Schleifen
  - Lock-Enter und Lock-Exit
  - For-Initialisierung und For-Update
  - ...

Im nächsten Abschnitt wird der Aufbau der zusammengesetzten Anweisungen vorgestellt.

4.3.2 Zusammengesetzte Anweisungen

Die zusammengesetzten Anweisungen werden benötigt, um komplexere Fälle bearbeiten zu können. Das sind Fälle, bei denen eine der Stellen des Datenwettlaufes in einem solchen Statement liegt. Dann hat die zusammengesetzte Anweisung einen Einfluss auf die Korrektur und muss mitbetrachtet werden. Ihr interner Aufbau orientiert sich an der Quelltextdarstellung, enthält aber immer noch einige Elemente, die für Assemblersprachen typisch sind. Zum Beispiel Sprünge, wenn diese nicht eindeutig mit einem anderen Teil kombiniert werden können.

Das in dieser Arbeit vorgestellte Code Object Model unterstützt die folgenden zusammengesetzten Anweisungen:

- BLOCK - Anweisung
- IF - Anweisung
- IF/ELSE - Anweisung
- DO/WHILE - Schleife
- WHILE - Schleife
- FOR - Schleife
- LOCK - Block

Diese Liste überdeckt nicht alle Sprachelemente einer modernen objektorientierten Programmiersprache ist aber für die Ziele der Arbeit ausreichend.

4.4 Analyse der Testergebnisse

Während der Testphase sollten die existierenden Datenwettläufe erkannt werden. Die Fehlerstellen sind aber nicht anhand der betroffenen Daten gruppiert, sondern liegen als Tuppel von Datenwettlaufpositionen vor. Deswegen müssen die Testergebnisse weiter analysiert werden, um die Fehlerstellen zu gruppieren und die folgenden Informationen zu extrahieren:

1. Kritische Daten
2. Liste der beteiligten Codestellen. Diese enthalten:
   a) Die genaue Codeposition, in Form einer Maschineninstruktion.
   b) Die Anweisung, zu der die Instruktion gehört. Dieses entspricht dem Quelltextstatement, das geschützt werden muss.
   c) Typ des Zugriffes: lesend oder schreibend.

In dieser Arbeit wird eine Datenwettlaufgruppe als eine Sammlung aller gemeldeten Fehlerstellen definiert, an denen auf dieselben Daten parallel zugegriffen wird. Diese Analyse muss als Abstraktion vom Test-Werkzeug verstanden werden. Damit bleibt die Testumgebung transparent und somit austauschbar [A1].

Jede Gruppe enthält Platzhalter für die folgenden Daten:

- Sperre, die die kritischen Abschnitte schützt.
- Für jede beteiligte Codestelle existiert ein Korrekturobjekt: eine Menge von Anweisungen, die von der Sperre zu schützen sind.

Die genaue Codestelle, an der ein paralleler Zugriff auf gemeinsam benutzte Daten aufgetreten ist, wird von den Testergebnissen hergeleitet und im eigentlichen Maschinencode verifiziert. Von der gefundenen Instruktion werden auch die betroffenen Daten ausgelesen. Je nach der Position der Codestelle werden zwei Szenarien untersucht:

1. Die Codestelle liegt im Programmcode.
2. Die Codestelle liegt in externem Code.

Liegt das zweite Szenario vor, dann steht auch kein Quelltext für die Codestelle zur Verfügung und die betroffenen Daten lassen sich mithilfe der Instruktion nicht auslesen. Einige Beispiele für Quellen solcher Stellen sind:

1. externe Bibliotheken.
2. Frameworks.
3. Codebasis der benutzten Programmiersprache.

Diese Stellen können nicht direkt geschützt werden und werden deswegen gesondert betrachtet. Die vom Datenwettlauf betroffenen Daten werden durch einen Methodenaufruf im Programmcode erreicht.

In Abbildung 4.2 ist der Stack Trace eines Datenwettlaufs dargestellt. Der Zugriff auf gemeinsame Daten liegt in der `List.Add`-Methode des .Net-Frameworks. Anhand des Stacktraces kann auch die Stelle des Aufrufs im Programmcode identifiziert werden.

```
System.Collections.Generic.List`1.$stub$Add( )
Bank.Account.Read()
Bank.Account.Withdraw(System.Int32 amount)
TestBank.TestBank.<WithdrawAndDepositConcurrently>b__0(System.Object o)
System.Threading.ThreadHelper.ThreadStart_Context(System.Object state)
System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext…)
System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext…)
System.Threading.ThreadHelper.ThreadStart(System.Object obj)
```

Abbildung 4.2: Stacktrace von einem Datenwettlauf in externem Code.

Wenn der Datenwettlauf im externen Code stattfindet, dann existieren mehrere Möglichkeiten für das betroffene Feld. Dieses kann zum einen ein, per Referenz übergebenes Argument oder das Instanzobjekt sein. Oder es ist ein Feld, das nicht zum eigentlchen Programmcode gehört.

Zu welcher Kategorie das betroffene Feld gehört, könnte anhand einer Kontroll-und Datenflussanalyse festgestellt werden. Die Kontroll-und Datenflussanalyse in dieser Arbeit basiert aber auf dem Code Object Model. Aus diesem Grund wird der gesamte Methodenaufruf als betroffene Stelle markiert und die Suche nach dem betroffenen Feld nicht ausgeführt.

Ein Spezialfall, bei dem es sich um Wettlaufsituationen in Datenstrukturen handelt, wird in Abschnitt 4.10.5 vorgestellt.

Aufbau der Datenwettlaufgruppen

Nachdem alle Fehlerstellen identifiziert und die notwendigen Informationen gesammelt sind, werden die Datenwettlaufgruppen aufgebaut. Die Gruppierung erfolgt nach dem betroffenen Feld.

Nach dem Aufbau werden die Gruppen noch einmal durchsucht, um duplizierte Einträge auszufiltern. Die Rückgabe solcher Stellen durch die Testumgebung ist nicht ausgeschlossen und muss aus diesem Grund behandelt werden. Es gibt zwei Kriterien, die erfüllt sein müssen, um zwei Stellen als identisch zu bezeichnen:

1. An den beiden Stellen ist eine Wettlaufsituation auf den selben Daten erkannt worden.
2. Die selbe Instruktion verursacht den Datenwettlauf.

Im nächsten Schritt werden die notwendigen Schutz-Objekte erstellt. Diese gewährleisten eine sequenzielle Ausführung der kritischen Regionen.

4.5 Bestimmen der Sperren

In diesem Abschnitt wird die Vorgehensweise bei der Bestimmung der Sperren erklärt. Diese sind notwendig, um Anforderung [A2] zu erfüllen. Dabei werden alle Stellen einer Datenwettlaufgruppe von derselben Sperre geschützt.

Diese Arbeit unterscheidet zwischen zwei Sperrentypen:

- Manuelle: Sperren, die durch den Entwickler definiert sind [A2.3].
- Generierte: Sperren, die durch das Werkzeug erstellt sind.

Die folgenden zwei Abschnitte erläutern diese zwei Typen und ihre Rolle im Verfahren.

4.5.1 Wiederverwendung von Sperren

Das definierte Verfahren unterstützt eine Wiederverwendung von Sperren aus den folgenden drei Gründen:

1. Neue Sperren werden nur dann erstellt, wenn keine eindeutig passenden bereits im Code vorhanden sind.
2. Generierte Sperren werden in weiteren Iterationen des Verfahrens erkannt und wiederverwendet. Ansonsten kann keine vollständige Lösung erstellt werden, wenn neue Datenwettläufe auf ein bereits geschütztes Feld entdeckt werden.
3. Dem Entwickler soll die Möglichkeit gegeben werden, manuelle Sperren zu definieren, die danach erkannt und verwendet werden [A2.3].

Bei der Wiederverwendung von Sperren muss die Beziehung Sperre/geschütztes Feld automatisch eindeutig bestimmbar sein. Nur dann kann das Verfahren eine bereits vorhandene Sperre benutzen. In vielen Situationen entspricht eine solche Relation aber auch nicht der Vorstellung des Entwicklers. Ein Schutzobjekt kann auch eine Region von semantisch zusammengehörenden Operationen schützten und nicht nur die Verwendung von einem Feld. Beispiele dafür sind existierende Sprachelemente zum Schützen von ganzen Methoden. Zu versuchen, die Semantik des vom Entwickler geschriebenen Codes zu interpretieren, ist fehleranfällig und geht weit über den Themenbereich dieser Diplomarbeit hinaus. Deswegen wird an dieser Stelle auf eine Sperreninferenz verzichtet.

Ein solches Verfahren wird zum Beispiel in [Bon11] vorgestellt und verwendet. Dabei werden die bereits geschützten Regionen und ihre Sperren evaluiert, um ein eindeutiges Schutzobjekt zu inferieren. Es gibt aber ein Problem, dass auch oben kurz erwähnt wurde: Syntaktisch schützt die gefundene Sperre das betroffene Objekt, semantisch ist es aber möglich, dass die Sperre für etwas Anderes vorgesehen ist.

Um die Sperreninferenz umzugehen und trotzdem dem Entwickler die Möglichkeit zu geben manuelle Sperren zu definieren [A2.3], werden Sperrenannotationen eingeführt. Die Annotation definiert die notwendige Sperre/geschütztes Feld Relation und wird sowohl zu manuellen, als auch zu generierten Sperren hinzugefügt. Abbildung 4.3 zeigt eine manuell angelegte und eine generierte Sperre. Die manuelle Sperre liegt in der Klasse, in der auch der Datenwettlauf erkannt wurde und die generierte in einer separaten Sperrenklasse.

Alle Klassen- und Instanzvariablen werden durchsucht, um alle annotierten Sperren zu sammeln. Das Verfahren unterscheidet danach zwischen manuellen und generierten Sperren nicht mehr. Ist keine Annotation vorhanden, dann wird das Feld ignoriert, auch wenn dieses im Code als Sperre verwendet wird.

Wenn alle Sperren vorliegen, werden die Datenwettlaufgruppen durchsucht, um passende Paare zu finden. Für alle Gruppen, für die keine Sperre im Code entdeckt wurde, muss eine neue generiert werden.

```
1. [LockDescriptor("Bank.Account.balance")]
2. public static object MC_StaticLockObject_balance = new Object();
3.
4. [LockDescriptor("Bank.Account.balance")]
5. private object balanceLock = new object ();
```

Abbildung 4.3: Zeile 1 und 2 - generierte Sperre; Zeile 4, 5 - manuelle Sperre

4.5.2 Erstellen neuer Sperren

Wird keine annotierte Sperre für eine Datenwettlaufgruppe gefunden, dann wird eine neue generiert. Eine entsprechende Annotation wird auch hinzugefügt, um die Sperre/geschütztes Feld Relation für weitere Iterationen des Verfahrens zu definieren. Abbildung 4.3 zeigt wie eine generierte Sperre und ihre Annotation aussehen.

Dieses Vorgehen hat auch Nachteile. Wenn im Code eine passende Sperre existiert, diese aber nicht annotiert ist, dann werden bereits geschützte Felder doppelt geschützt. Dies wird zu keinem fehlerhaften Code führen, hat aber einen negativen Einfluss auf die Performanz.

4.6 Der Aufrufgraph

Der Aufrufgraph ist eine Datenstruktur, die Informationen über die Methodenaufrufe im Programmcode enthält. Diese kann nach ihrem Aufbau für eine Methode A auf die folgenden zwei Fragen antworten:

1. Welche Methoden werden an welcher Stelle von A aufgerufen?
2. In welchen Methoden und an welchen Stellen wird A aufgerufen?

Aufrufgraphen sind Datenstrukturen, die in der Codeanalyse und der Codeoptimierung verwendet werden [Muc07]. Diese sind maßgeblich für die Einhaltung der Anforderungen [A2.2], [A3] und [A4]. Der in dieser Arbeit verwendete Graph unterscheidet sich vom typischen baumförmigen Aufrufgraphen, bei dem die Main-Methode dem Startknoten entspricht. Der Unterschied ist der Tatsache geschuldet, dass nicht jedes Codestück ein selbstständiges Programm ist und eine Main-Methode enthält. Bibliotheken und Frameworks haben i.d.R. keinen eindeutig definierten Ausgangspunkt.

Es gibt zwei Möglichkeiten, dieses Problem zu lösen:

1. Alle Methoden, die von Testmethoden direkt aufgerufen werden, sind Ausgangspunkte.
2. Alle Methoden, die die Klasse als Schnittstelle nach außen sichtbar definiert, sind Ausgangspunkte.

Beim ersten Vorschlag ist es möglich Abhängigkeiten zu verpassen, wenn keine vollständige Codeabdeckung durch die Komponententests vorliegt. Ist die Abdeckung vollständig, dann definieren die zwei Vorschläge äquivalente Mengen von Ausgangsmethoden. Bei diesem Aufbau des Aufrufgraphen hat er keine Baumstruktur mit einer eindeutigen Wurzel mehr. Es gibt tatsächlich keinen Startknoten. Abbildung 4.4 zeigt einen solchen Graphen.

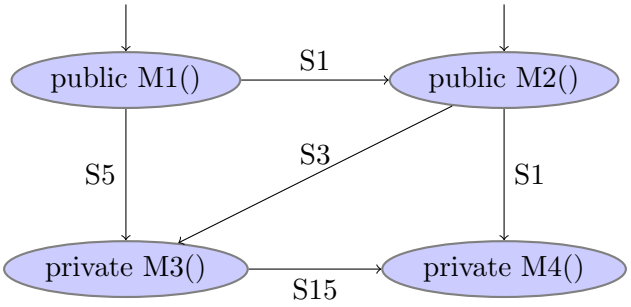

Abbildung 4.4: Beispiel eines Aufrufgraphen. Die Kantenanotationen bezeichnen die Anweisung in der der Aufruf passiert.

Der Aufbau des Graphen erfolgt folgendermaßen:

1. Für jede Methode wird ein Knoten erstellt, dazu zählen auch private Methoden.
2. Für jede Methode werden die von ihr aufgerufenen Methoden bestimmt.
3. Sind diese Methoden Teil des zu evaluierenden Codes und können diese statisch bestimmt werden, wird eine Abhängigkeit mit der genauen Position (Offset, Statement usw.) im Graphen hinzugefügt. Es werden nur eigene Methoden in Betracht genommen, weil für alle externen kein Code Object Model vorhanden ist. Dies macht deren Evaluierung nicht möglich. Auch virtuelle Methoden werden ignoriert, gemäß [E4].

4.7 Der Sperrengraph

Nach Anforderung 2.2 darf das Verfahren keine Verklemmungen während der Korrektur einbauen. Um diese Anforderung zu erfüllen, werden Informationen über die Reihenfolge der verwendet Sperren benötigt. Dies bedeutet:

Wenn eine Sperre innerhalb von einer Region akquiriert wird, die durch eine andere Sperre geschützt ist, entsteht eine feste Reihenfolge zwischen den beiden Sperren (siehe Abbildung 4.5). Falls diese Reihenfolge an einer anderen Stelle im Code verletzt ist und die zwei Sperren in umgekehrter Reihenfolge akquiriert werden, besteht Verklemmungsgefahr.

```
1.  public void SwitchTrafficLightState(TrafficLight t)
2.  {
3.      lock (RoadSystem)
4.      {
5.          if (RoadSystem.IsOnline())
6.          {
7.              lock (TrafficLightManager)
8.              {
9.                  TraddicLightManager.SwitchStateOf(t);
10.             }
11.         }
12.     }
13. }
```

Abbildung 4.5: Beispiel mit zwei geschachtelten Lock-Blöcken. Diese Schachtelung erzeugt eine feste Reihenfolge zwischen RoadSystem und TrafficLightManager.

Der Sperrengraph ist eine Abbildung der Abhängigkeiten aller Sperren, die innerhalb des evaluierten Codes zu finden sind. Dabei werden, dank des Aufrufsgraphen, auch Sper-

renreihenfolgen betrachtet, die die Grenze der Methode überschreiten, siehe Abbildung 4.6.

```
1.  public void SwitchTLState(TrafficLight t)
2.  {
3.      lock (RoadSystem)
4.      {
5.          if (RoadSystem.IsOnline())
6.          {
7.              TLManager.SwitchStateOf(t);
8.          }
9.      }
10. }
```

```
1. public void SwitchStateOf (TrafficLight t)
2. {
3.     lock (TrafficLightManager)
4.     {
5.         t.NextState();
6.     }
7. }
```

Abbildung 4.6: Beispiel mit zwei geschachtelten Lock-Blöcke.

Entsteht ein Zyklus im Graphen, so existiert eine potenzielle Verklemmungssituation (Abbildung 4.7). Dieses Vorgehen erkennt auch Situationen, die bei einer Ausführung nie zu einer Verklemmung führen können (false positives). Diese beeinträchtigen das korrekte Ergebnis dieses Verfahrens jedoch nicht.

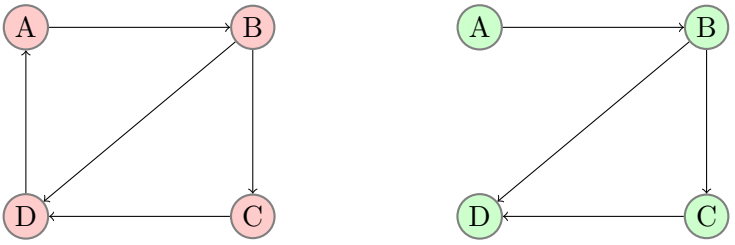

Abbildung 4.7: Auf der Abbildung sind zwei Akquirierungsreihenfolgen der Sperren A, B, C und D mithilfe von Sperrgraphen visualisiert. Der linke Graph ist nicht zyklusfrei und meldet eine mögliche Verklemmung. Der rechte Graph enthält keine Zyklen und der Code enthält mit Sicherheit keine Verklemmungen.

Der Sperrengraph erfüllt insgesamt drei Funktionen:

1. Er beschreibt alle bestimmbaren Sperrenabhängigkeiten im Code.
2. Kann für den Ausgangscode entscheiden, ob eine potenzielle Verklemmungssituation vorhanden ist.
3. Kann eine Aussage treffen, ob die definierten Korrekturen eine potenzielle Verklemmung verursachen werden [A2.2].

Falls der Sperrengraph Zyklen enthält, dann ist das ein Signal, dass potenzielle Verklemmungssituationen vorhanden sind. Sollte das der Fall sein, muss das Verfahren unterbrochen werden, bis die Reihenfolge der Sperren manuell geändert wurde.

Bevor eine errechnete Korrektur im Quelltext gespeichert wird, wird beim Sperrengraphen nachgefragt, ob das Hinzufügen der Sperren potenziell Verklemmungen verursachen könnte. Ist das der Fall, so wird die Korrektur abgebrochen und die Änderung nicht im Quelltext übertragen. So ist auch die Erfüllung der Anforderung [A2.2] sicher gestellt.

4.8 Der Datenflussgraph

Der Datenflussgraph ist eine Kripke-ähnliche [Kri71] Datenstruktur, die Informationen über den Datenfluss innerhalb eines Kontrollflusses eines Methodenkörpers enthält. Dieser wird benötigt, um [A3] und [A4] zu erfüllen. Der Datenflussgraph (siehe Abbildung 4.8) ist aus den folgenden Elementen aufgebaut:

1. Datenflussereignisse
2. Ereigniscontainer
3. Ereignisbeziehungen

Diese werden in den folgenden Abschnitten detailliert erklärt. Die Beschreibungen werden von Abbildung 4.8 grafisch unterstützt.

4.8.1 Datenflussereignisse

Datenflüsse innerhalb des Methodenkörpers werden von Lese-und Schreibereignissen aufgebaut. Der Datentransfer von Feld zu Feld und auf höherer Ebene von Methode zu Methode erfolgt im Programm durch eine Reihe von Lese- und Schreiboperationen, die in Ereignissen gekapselt werden. Unter Lesen und Schreiben wird folgendes verstanden:

- Lesen: Der Wert eines Felds bzw. einer Konstanten wird im Ausführungsspeicher geladen.
- Schreiben: Ein Wert, der im Ausführungsspeicher liegt, wird in ein Feld gespeichert.

Jedes Ereignis lässt sich nach seiner Operation eindeutig identifizieren (Operationseigenschaft des Ereignisses in Abbildung 4.8). Die verschiedenen Ereignisse werden untereinander verbunden. Das Verfolgen dieser Verbindungen kann den genauen Datenfluss eines Objektes wiederherstellen. Dabei gilt:

- Ein Leseereignis ist nur mit einem Schreibereignis verbunden: das Ereignis, in das der ausgelesene Wert geschrieben wird.
- Ein Schreibereignis kann mit 0 bis unendlich viele Leser verbunden sein. Das sind alle Leseereignisse, die einen Einfluss auf den in das Schreibereignis geschriebenen Wert haben.

Man betrachtet das Beispiel in Abbildung 4.8: `balance = balance + amount;`. Erkannt werden zwei Leseereignisse, die die Werte von `balance` und `amount` auslesen. Danach werden diese Werte addiert. Daraus folgt: Die Daten fließen von `balance` und `amount` nach `balance`. Das bedeutet die `balance` und `amount` Leseereignisse sind mit dem Schreibereignis auf `balance` verbunden.

4.8.2 Ereigniscontainer

Alle Ereignisse werden in Container nach dem Datenträger gruppiert. Datenträger können ein konkretes Feld, ein Methodenaufruf oder auch Konstanten/Strings und Return-Wrapper sein.

Anhand der Reihenfolge im Container können Vorgänger-und Nachfolgerereignisse bestimmt werden. Dies ermöglicht eine Traversierung innerhalb der Struktur. (siehe die Container in Abbildung 4.8)

Die Container bezeichnen die Knoten des Datenflussgraphen und die Beziehungen zwischen den einzelnen Ereignissen die Kanten. Die Kanten sind auf Abbildung 4.8 durch Farben dargestellt.

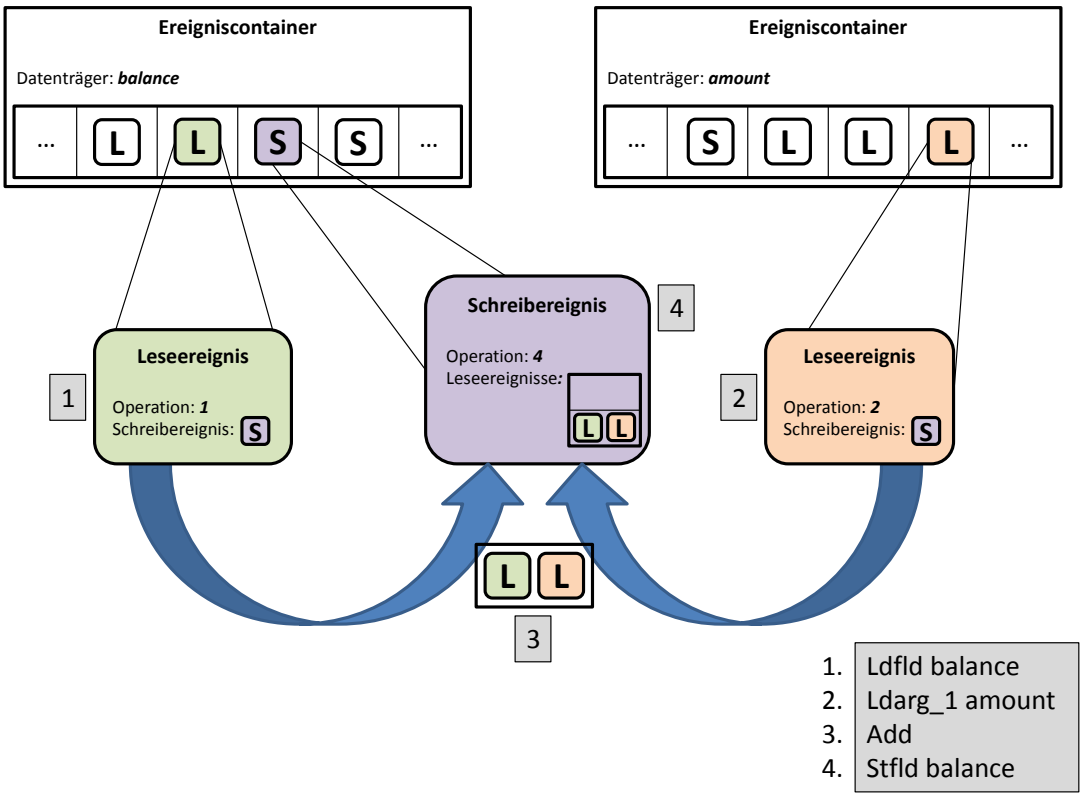

Abbildung 4.8: Ausschnitt eines Datenflussgraphen. Beschreibt den Datenfluss, der bei der Ausführung der Anweisung `balance = balance + amount;` entsteht. Unten rechts sieht man die Maschineninstruktionen, die die einzelnen Schritte des Flusses verursachen. Die Verbindungen zwischen den Ereignissen werden in dieser Abbildung durch die Verwendung derselben Farbe dargestellt. Die blauen Pfeile zeigen die Fluss- und Ausführungsrichtung.

4.8.3 Traversierung und Nutzung

Zum Traversieren sind im Graphen für jedes Ereignis die folgenden Operationen vorgesehen:

- Bestimmen des Vorgängers und des Nachfolgers im eigenen Container.
- Bestimmen der direkt abhängigen Ereignisse.

Mithilfe des Graphen kann der Datenfluss mit Startpunkt ein beliebiges Ereignis entweder weiter oder zurück zur Quelle verfolgt werden.

Man kann zum Beispiel nach Datenabhängigkeiten zwischen Feldern suchen [A4] oder auch bestimmen, ob ein Feld von sich selbst abhängig ist. Der Datenflussgraph wird später im Kapitel benutzt, um die genauen Typen von Instanzobjekten zu bestimmen. Diese werden von [A3] benötigt.

Datenflussgraphen verschiedener Methoden können verwendet werden, um den Datenaustausch und den Datenfluss bei der Parameterübergabe und beim Rückgabewert von Methodenaufrufen zu bestimmen.

Wie der Datenflussgraphen während der Kontrollflussanalyse zustande kommt, wird im nächsten Abschnitt erläutert.

4.9 Kontroll- und Datenflussanalyse

Die Kontrollflussanalyse in dieser Arbeit ist eine statische Analyse, deren Ziel es ist, alle möglichen Ausführungspfade einer Methode aufzubauen und zu verfolgen. Die Datenflussanalyse generiert für jeden Kontrollfluss eine Liste mit Lese-Schreib-und Aufrufereignissen sowie den oben definierten Datenflussgraphen. Die gesammelten Informationen beteiligen sich an der Erfüllung von mehreren Anforderungen, insbesondere [A3], [A4], [A5].

4.9.1 Motivation

Die genaue Position des betroffenen Feldes ist von besonderer Bedeutung. Eins der Ziele dieser Arbeit ist die geschützten Regionen aus Performanz- und Korrektheitsgründen möglichst klein zu halten [A6].

Wie das Beispiel auf Abbildung 4.9 zeigt, existieren in der IF/ELSE-Anweisung zwei Wettlaufsituationen auf das `state`-Feld. Die Herausforderung liegt nun in der Frage, ob die komplette Anweisung geschützt werden soll oder die zwei Stellen jeweils separat. Die zwei Vorkommnisse von `state` sind voneinander vollständig unabhängig, weil diese zu verschiedenen Kontrollflüssen gehören. Diese Frage kann nur dann beantwortet werden, wenn es bekannt ist, ob die Vorkommnisse mit Daten außerhalb der Körper in Beziehung stehen.

Die Kontroll-und Datenflussanalyse kann die Antwort der Frage ermöglichen. Diese baut die separaten Ausführungspfade und die dazugehörige Datenflussinformation auf. Das bedeutet, für jedes Vorkommnis eines Feldes können danach nur Daten analysiert werden, die in seinem Kontrollfluss liegen.

4.9.2 Aufbau und Ausführung

Die Kontrollflussanalyse wird mithilfe des Code Object Models aufgebaut und durchgeführt. Das Code Object Model verhindert das endlose Iterieren bei Schleifen und erlaubt somit das korrekte Traversieren des Methodenkörpers. Weiterhin wird sie von der simulierten Ausführung unterstützt. Die simulierte Ausführung ist eine statische Analyse, die alle Kontrollflüsse im Programm durchläuft und Informationen über den Datenfluss sammelt.

```
1.  if (turnSubsystemOn)
2.  {
3.      ...
4.      state = SystemState.Online;
5.      ...
6.  }
7.  else
8.  {
9.      ...
10.     state = SystemState.Ofline;
11.     ...
12. }
```

Abbildung 4.9: Die zwei Zugriffe auf `state` werden von Datenwettläufen betroffen. Die zwei Körper der Anweisung sind mit vieler zusätzlichen Operationen belastet.

Dabei werden die Ereignisliste und der Datenflussgraph aufgebaut. Es handelt sich bei dieser Analyse um keine echte Code-Ausführung, sondern um eine lokale Code-Interpretation. Variablen werden durch Datencontainer dargestellt und alle Lese- und Schreiboperationen erstellen Ereignisse auf die Datencontainer. Eine Datenstruktur übernimmt die Rolle des Ausführungsspeichers. Bei der Simulation werden die Ereignisse genau so manipuliert, wie die Daten bei einer echten Programmausführung. Eine Leseoperation erstellt ein Leseereignis im Container und kopiert dieses im Ausführungsspeicher. Eine Schreiboperation wird das Leseereignis aus dem Ausführungsspeicher holen und eine Abhängigkeit zwischen ihm und einem neuen Schreibereignis erstellen. Instruktionen, die nur mit dem Ausführungsspeicher arbeiten, können Ereignisse Gruppieren, Kopieren, Löschen, Verschieben u.s.w.

Abschnitt 5.6 erläutert, wie die Kontroll- und Datenflussanalyse mit der simulierten Ausführung realisiert werden kann.

Ergebnis

Nach der Abarbeitung der Kontrollflüsse aller Methoden entsteht für jeden Kontrollfluss eine Liste mit Lese-, Schreib-und Aufrufereignisse (siehe Abbildung 4.10) und einen Datenflussgraph (siehe Abbildung 4.8). Die Listen werden im nächsten Schritt analysiert, um die Abhängigkeiten zwischen den verschiedenen Vorkommnissen zu bestimmen [A4]. Die entstandenen Datenflussgraphen werden vor allem beim Austausch der Datenstrukturen benötigt [A3].

4.10 Analyse der Datenwettläufe

Nun verfügt das Verfahren über alle notwendigen Informationen und Daten, um die verschiedenen Datenwettlaufsituationen zu evaluieren. Ziel dieser Evaluation ist es nicht nur die Instruktion, die den Datenwettlauf verursacht hat, sondern ganze Regionen zu schützen, die von logisch [A4] abhängigen Codeteilen aufgebaut sind [A2]. Dabei muss unbedingt eine weitere Einschränkung definiert werden:

E5: Alle Anweisungen, die für die Korrektur einer Stelle zu schützen sind, müssen dieselbe Codetiefe haben. Das heißt, die erste und die letzte Anweisung gehören entweder zum Körper desselben zusammengesetzten Statements oder liegen direkt im Methodenkörper (siehe Abbildung 4.11).

Ereignis	Objekt
READ	Argument_0
READ	Bank.Account.list
READ	String
CALL	System.Void System.Collections.Generic.List<System.String>.Add(System.String)
READ	Argument_0
READ	Argument_0
READ	Bank.Account.balance
READ	amount
WRITE	Bank.Account.balance
READ	Argument_0
READ	Bank.Account.balance
WRITE	Return

Abbildung 4.10: Ausschnitt der Ereignisliste einer Methode. Auf der Abbildung sind die Operationen, die die Ereignisse verursacht haben, nicht dargestellt.

```
1. ...
2. if (amount > 0)
3. {
4.     balance = temp - amount;
5. }
6.
7. PersistValue(balance);
8. ...
```

```
1. ...
2. if (amount > 0)
3. {
4.     balance = temp - amount;
5. }
6.
7. PersistValue(balance);
8. ...
```

Abbildung 4.11: Die unterstrichenen Zeilen definieren die Regionen, die geschützt werden. Links sieht man eine falsch definierte Region. Rechts eine korrekt definierte.

Grund für [E5] ist die Verwendung von Synchronisationsblöcken für die Korrektur. Diese garantieren das Freigeben der Sperre in allen Kontrollflussszenarien, auch wenn im geschützten Code ein Fehler geworfen wird [A2.1]. Werden keine Blöcke verwendet, sondern direkt die Akquirierung- und Freigabenmethode, dann könnte eine Verklemmung entstehen, wenn ein Fehler vor der Freigabe der Sperre geworfen wird, was wiederum [A2.2] verletzt. Ein Beispiel für eine solche Situation sieht man in Abbildung 4.12.

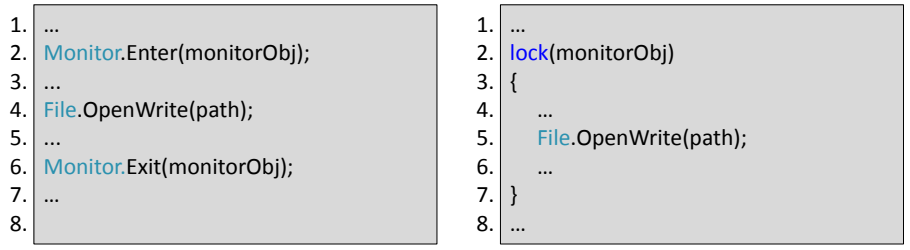

Abbildung 4.12: Links erfolgt die Synchronisierung der Region durch direkte Verwendung der .NET Methoden `Monitor.Enter()` und `Monitor.Exit()` Rechts wird dieselbe Region durch einen .NET `lock`-Block synchronisiert. Die Region enthält die .NET Methode `File.OpenWrite()`, die ein `IOException` wirft, wenn die Datei nicht vorhanden ist. Ist dies der Fall, dann findet der Aufruf von `Monitor.Exit()` nie statt und die Sperre wird nicht freigegeben. Der `lock`-Block verwendet die `Monitor.Enter()` und `Monitor.Exit()`-Methoden für die Synchronisation, jeweils mit einer Fehlerbehandlung, die die Freigabe der Sperre garantiert.

Die Verwendung eines Synchronisationsblocks erhöht die Chance für eine erfolgreiche Korrektur. Während der Analysephase werden für jede Fehlerstelle Anweisungen in einer Liste gesammelt. Diese Anweisungen stehen in einer Beziehung zu der betroffenen Stelle (zum Beispiel wegen [A5] und [A4]) und müssen deswegen mitgeschützt werden. Danach werden sie zu einer Region ergänzt. Im letzten Schritt erfolgt eine Anpassung dieser Region, die dafür sorgt, dass [E5] erfüllt ist.

Bei der Analyse wird jede Fehlerstelle separat behandelt. Wie die anderen Stellen aussehen, wird nicht betrachtet. Dies könnte ein guter Ausgangspunkt für weitere Arbeiten sein.

Die Anpassung findet auf Quelltextebene statt [A8], deswegen können keine einzelnen Instruktionen geschützt werden und die kleinste verwendete Einheit ist die Anweisung. Ausgangspunkt der Analyse ist die Anweisung, die die betroffene Instruktion enthält. Mithilfe des bereits aufgebauten Code Object Models ist diese einfach zu bestimmen.

Die Analyse kann man folgendermaßen aufteilen:

1. Schützen einfacher Anweisungen und Regionen [A2].
2. Einfluss zusammengesetzter Anweisungen für die Erfüllung von [A5].
3. Feldabhängigkeitsanalyse und Atomizitätsverletzung [A4].
4. Schützen von Fehlerstellen in externen Code [A3].
5. Zusammenfassung der Ergebnisse und letztes Expandieren, um Block-Struktur zu erreichen [E5], [A6].
6. Entfernung von Duplikaten [A7].

4.10.1 Schützen einfacher Anweisungen und Regionen

In diesem Abschnitt wird das Schützen im grundlegenden Fall, als Teil von [A2] erläutert:

- Die Datenwettlaufgruppe betrifft ein Feld.
- Alle Fehlerstellen der Gruppe liegen direkt im Methodenkörper.
- Abhängige Vorkommnisse liegen entweder auf derselben Ebene oder stecken in Methodenaufrufen dieser Ebene.
- Zusammengesetzte Anweisungen können zwischen zwei Vorkommnissen liegen, kein Vorkommnis darf jedoch Teil einer zusammengesetzten Anweisung sein.

Ein Beispiel, wie eine solche Situation aussieht, findet man in Abbildung 4.13.

```
1. public void IncrementWith(int amount)
2. {
3.     if (amount < 0)
4.     {
5.         amount = 0;
6.     }
7.
8.     value = value + amount;
9. }
```

```
1. public void DecrementWith(int amount)
2. {
3.     if (amount < 0)
4.     {
5.         amount = 0;
6.     }
7.
8.     value = value - amount;
9. }
```

Abbildung 4.13: Wegen des Datenwettlaufes auf Zeile 8 in den beiden Methoden könnte eine von den zwei Aktualisierungen von `value` verloren gehen. Als Ergebnis enthält `value` den falschen Wert.

Das Beheben des Datenwettlaufs verlangt in diesem Fall das Schützen aller erkannten Stellen vor parallelem Ausführen. Dies erfolgt durch den Lock-Block, wie auf Abbildung 4.14 gezeigt worden ist.

```
1. public void IncrementWith(int amount)
2. {
3.     if (amount < 0)
4.     {
5.         amount = 0;
6.     }
7.
8.     lock (valueLock)
9.     {
10.        value = value + amount;
11.    }
12. }
```

```
1. public void DecrementWith(int amount)
2. {
3.     if (amount < 0)
4.     {
5.         amount = 0;
6.     }
7.
8.     lock (valueLock)
9.     {
10.        value = value - amount;
11.    }
12. }
```

Abbildung 4.14: Dank des Synchronisationsblocks können die zwei Stellen nicht mehr parallel ausgeführt werden. Es treten keine Datenwettläufe mehr auf.

Darüber hinaus gibt es aber auch noch deutlich komplexere Strukturen. Daher wird im Folgenden beschrieben, welchen Einfluss die Existenz zusammengesetzter Anweisungen auf die geschützten Regionen hat.

4.10.2 Einfluss zusammengesetzter Anweisungen

In diesem Abschnitt wird der Einfluss zusammengesetzter Anweisungen auf die zu schützenden Regionen diskutiert [A5]. Laut der Definition des Code Object Models sind zusammengesetzte Anweisungen Block-Strukturen wie zum Beispiel Schleifen.

In der beschriebenen Schutzmöglichkeit wird jedes Feldvorkommnis einzeln betrachtet. Das kann zu folgenden Situationen führen:

1. Situationen, in denen mehrere Vorkommnisse in derselben zusammengesetzten Anweisung auftauchen werden einzeln behandelt.
2. Dieselben Codeteile tauchen mehrmals in der Liste auf.

Um die Komplexität an dieser Stelle zu minimieren und trotzdem schrittweise zum gewünschten Ziel zu kommen, werden an dieser Stelle keine Maßnahmen gegen 1 und 2 unternommen. Eine spätere Duplikatentfernung und eine Regionenanpassung vermeidet das mehrfache Schützen und gewährleistet eine korrekte Anpassung des Quelltextes.

4.10.2.1 Schützen einer IF/ELSE-Anweisung

Die IF/ELSE-Anweisung, die das Code Object Model definiert, hat zwei grundlegenden Formen: mit und ohne den ELSE-Teil. In Abhängigkeit davon, wo der Datenwettlauf stattfindet, beeinflusst die Anweisung den zu schützenden Code unterschiedlich:

```
1. if (increment)
2. {
3.     balance = temp + amount;
4. }
5. else
6. {
7.     balance = temp - amount;
8. }
```

```
1.  if (increment)
2.  {
3.      lock (balanceLock)
4.      {
5.          balance = temp + amount;
6.      }
7.
8.  }
9.  else
10. {
11.     lock (balanceLock)
12.     {
13.         balance = temp - amount;
14.     }
15. }
```

Abbildung 4.15: Links: ungeschützte IF/ELSE-Anweisung. Rechts: geschützte IF/ELSE-Anweisung.

- Findet der Datenwettlauf in den Körpern statt, dann wird nur die Fehlerstelle geschützt (siehe Abbildung 4.15).
- Findet der Datenwettlauf in der Bedingung statt, dann wird die ganze IF/ELSE-Anweisung geschützt (siehe Abbildung 4.16 (b)).

Der Datenwettlauf in der Bedingung führt zu einem unnötigen Schützen der gesamten Anweisung. Dies kann die Performanz deutlich verschlechtern, wenn der Körper zeitaufwendige Berechnungen enthält. In Fällen, in denen ein abhängiges Feldvorkommnis in den Körpern oder nach der IF/ELSE-Anweisung liegt, ist das Schützen notwendig, um eine Atomizitätsverletzung zu vermeiden [A4].

Wenn die Kontrollflussanalyse aber beweisen kann, dass die betroffene Bedingung keine relevanten Nachfolgervorkommnisse hat, dann kann der Datenwettlauf folgendermaßen korrigiert werden:

- Die IF/ELSE-Bedingung wird in eine lokale Variable extrahiert und vor der Anweisung in einem geschützten Block berechnet. Als Bedingung wird dann die lokale Variable verwendet (siehe Abbildung 4.16 (c)).

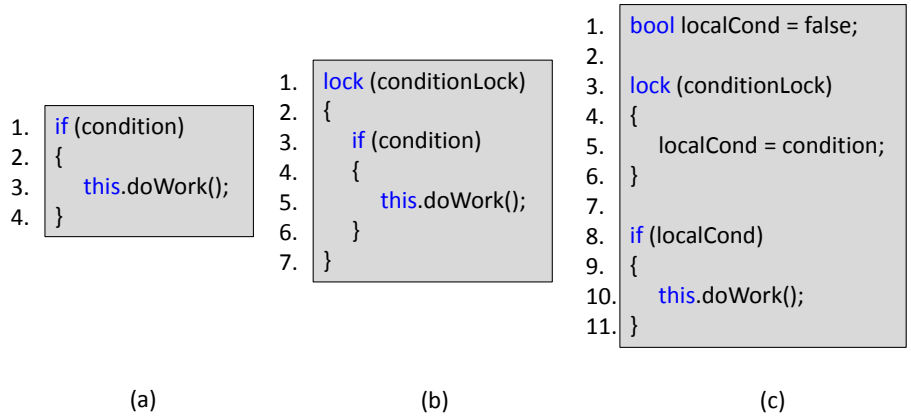

Abbildung 4.16: (a) - ursprüngliche Anweisung; (b) - komplett geschützte Anweisung; (c) - Schützten durch Bedingungsextrahierung

4.10.2.2 Schützen einer Schleife

Die Korrektur einer Schleife sieht ähnlich wie die Korrektur einer IF/ELSE-Anweisung aus. Das Code Object Model unterstützt drei verschiedene Schleifen: WHILE (siehe Abbildung 4.17 (a)), DO/WHILE (siehe Abbildung 4.17 (d)) und die FOR-Schleife (siehe Abbildung 4.18 (a)). Für alle drei gilt:

- Findet der Datenwettlauf im Schleifenkörper statt, dann ist der betroffene Codebereich ganz normal mit einem LOCK-Block zu schützen. (siehe Abbildungen 4.17 (b) (e) und 4.18 (b))
- Findet der Datenwettlauf jeweils in der Bedingung statt, dann muss die ganze Schleife geschützt werden. (siehe Abbildungen 4.17 (c) (f) und 4.18 (c))

Für die FOR-Schleife gilt zusätzlich:

- Findet der Datenwettlauf in der Initialisierung- oder in der Update-Anweisung statt, dann wird die komplette FOR-Schleife durch den LOCK-Block geschützt (siehe Abbildung 4.18 (c))).

Das Schützen einer Schleife, die einen Datenwettlauf im Körper enthält, wird komplizierter, wenn man die Performanz der Schleife betrachtet. Das Akquirieren und Freigeben von Sperren verlangt viel Zeit, auch wenn die Sperre gleich zur Verfügung steht. Aus diesem Grund können diese zwei Operationen negative Konsequenzen auf die Performanz der Schleife haben. Betrachtet werden die folgenden zwei Beispiele einer WHILE-Schleife:

Im ersten Beispiel (siehe Abbildung 4.19) wird innerhalb einer WHILE-Schleife ein von einem Datenwettlauf betroffenes Feld um eins erhöht. Die Schleife wird 1 000 000 Mal ausgeführt. Wenn die oben definierte Korrektur verwendet wird, dann wird der LOCK-Block in der WHILE-Schleife eingebaut (siehe Abbildung 4.19 (b)). Als Resultat wird die Sperre 1 000 000 Mal akquiriert und danach 1 000 000 Mal freigegeben.

Das Akquirieren und Freigeben der Sperre wird die Schleife deutlich verlangsamen. Aus diesem Grund wird für diesen Fall eine zweite Korrekturmöglichkeit vorgeschlagen:

- Findet der Datenwettlauf im Schleifenkörper statt, dann ist die komplette Schleife mit einem Lock-Block zu schützen.

Mit dieser Lösungsstrategie wird im oberen Beispiel die Sperre nur einmal am Anfang akquiriert und nach Verlassen der Schleife wieder freigegeben (siehe Abbildung 4.19 (c)).

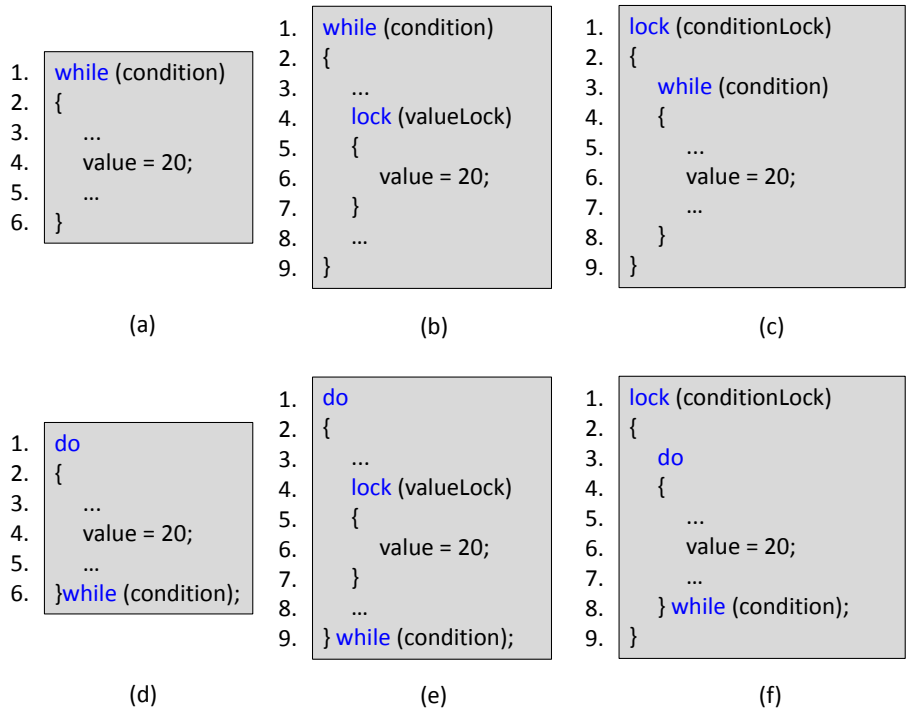

Abbildung 4.17: (a), (d) - ursprüngliche Schleifen; (b), (e) - Schützen eines Datenwettlaufes in dem Körper; (c), (f) - Schützten der kompletten Schleife wegen der Bedingung.

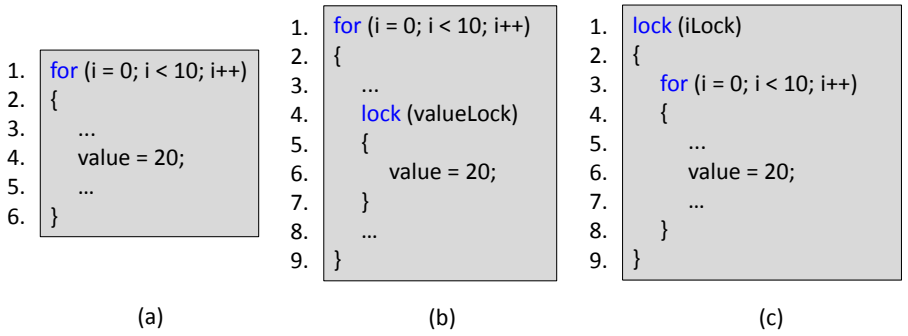

Abbildung 4.18: (a) - ursprüngliche Schleife; (b) - Schützen eines Datenwettlaufes in dem Körper; (c) - Schützten der kompletten Schleife wegen der Initialisierung/Bedingung/Update.

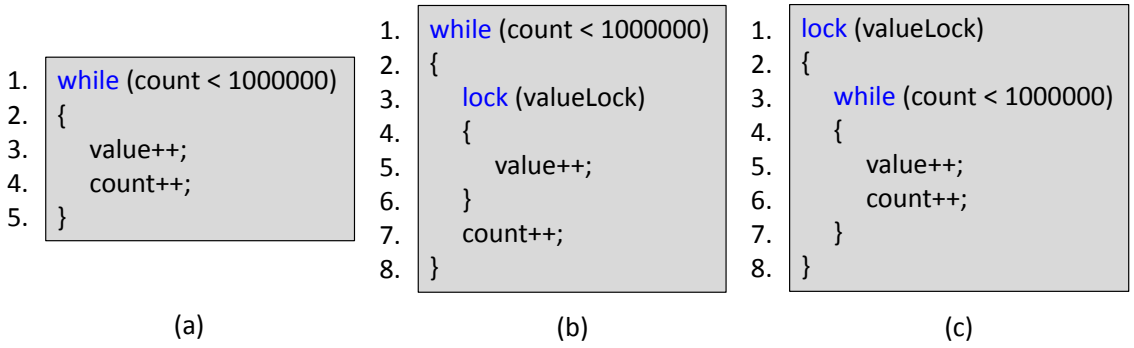

Abbildung 4.19: (a) - ursprüngliche Schleife; (b) - Schützen nur in dem Körper; (c) - Schützten der kompletten Schleife.

Betrachtet wird jetzt das zweite Beispiel (siehe Abbildung 4.20). Im Vergleich zum Ersten enthält der Methodenkörper nicht nur die Variableninkrementierung, sondern auch den Aufruf einer sehr zeitaufwendigen Methode. Wenn dieses Beispiel mit dem gerade gemachten Vorschlag korrigiert wird (siehe Abbildung 4.20 (c)), dann wird die Sperre für die ganze Ausführungszeit der Schleife unnötig akquiriert, was wiederum zu einer schlechten Performanz führt. In diesem Fall ist das Sperren nur der betroffenen Anweisung sinnvoller (siehe Abbildung 4.20 (b)).

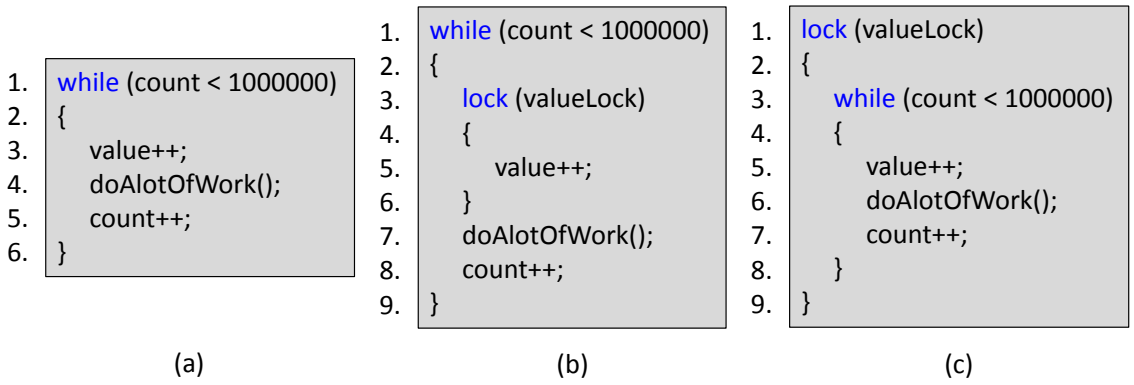

Abbildung 4.20: (a) - ursprüngliche Schleife; (b) - Schützen eines Datenwettlaufes in dem Körper; (c) - Schützten der kompletten Schleife.

Welche von den beiden Lösungsvorschlägen in einer konkreten Situation sinnvoller ist, ist eine komplizierte Frage, die in dieser Arbeit anhand zweier Extremfälle vorgestellt wurde. Diese Frage kann in einer weiterführenden Arbeit in Details untersucht werden. Das in dieser Ausarbeitung definierte Verfahren geht davon aus, dass ein Mechanismus existiert, der korrekt zwischen den beiden Fällen entscheidet.

4.10.2.3 Schützen eines LOCK-Blocks

Bei einem LOCK-Block kann ein Datenwettlauf entweder im Körper oder in der LOCK-Enter-Anweisung vorhanden sein. Die Möglichkeit, dass der Datenwettlauf in der LOCK-Exit-Anweisung steckt, wird aus dem folgenden Grund ausgeschlossen:

Welches Objekt als Sperre verwendet wird, wird am Anfang des LOCK-Enters bestimmt. Deswegen können dort Datenwettläufe eingebaut werden. Nach der Bestimmung wird eine lokale Referenz auf die Sperre angelegt. Danach erfolgt das Akquirieren und das Freigeben nur über die lokale Referenz.

Ein LOCK-Block wird folgendermaßen korrigiert:

- Liegt der Datenwettlauf im Körper, dann ist der betroffene Codebereich mit einem LOCK-Block zu schützen (siehe Abbildung 4.21 (b)).
- Findet der Datenwettlauf in der LOCK-Enter-Anweisung statt, dann wird der komplette LOCK-Block geschützt (siehe Abbildung 4.21 (c)).

Beim Schützen vom gesamten LOCK-Block ist, ähnlich wie bei der IF/ELSE-Anweisung, Folgendes möglich:

- Wegen des Datenwettlaufs im LOCK-Enter wird der gesamte LOCK-Block doppelt gesperrt. In Abhängigkeit davon, was dieser enthält und wie lange die Ausführung dauert, verschlechtert sich die Performanz.

Kommen nachfolgend keine weiteren Vorkommnisse vor, dann lässt sich der Datenwettlauf in der Lock-Enter-Anweisung folgendermaßen korrigiere:

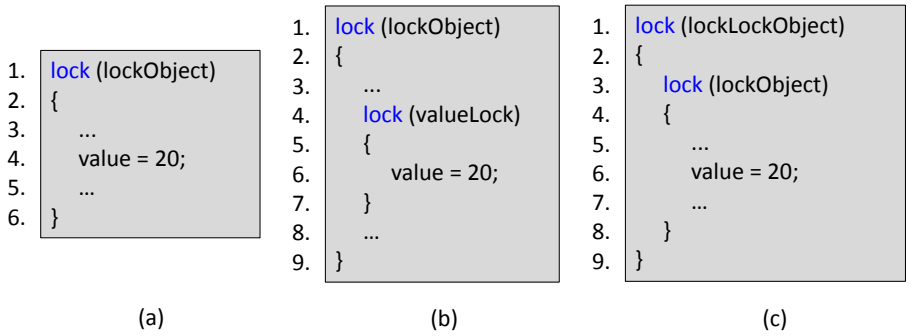

Abbildung 4.21: (a) - ursprünglicher Lock-Block; (b) - Schützen eines Datenwettlaufes im Körper; (c) - Schützen eines Datenwettlaufes im Lock-Enter.

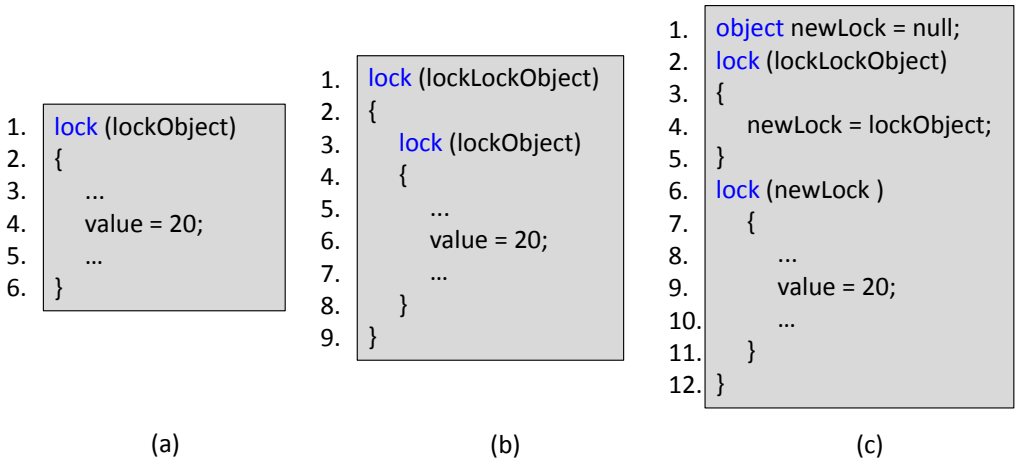

Abbildung 4.22: (a) - ursprünglicher Lock-Block; (b) - Schützen eines Datenwettlaufes im Lock-Enter; (c) - Schützen eines Datenwettlaufes im Lock-Enter durch Sperrenextrahierung.

- Der Inhalt von dem LOCK-Enter wird in eine lokale Variable vom Typ Object extrahiert. Diese wird dann vor dem LOCK-Enter in einem geschützten LOCK-Block bestimmt (4.22 (c)).

Diese Idee hat für die Korrektur einen weiteren Vorteil. Sie minimiert die Kette der abhängigen Sperren in dem Sperrgraphen.

4.10.3 Feldabhängigkeitsanalyse und Atomizitätsverletzung

Bis jetzt wurde in dieser Arbeit nur vom betroffenen Feld bzw. von Instruktion und Anweisung gesprochen. In der Praxis sehen aber die Situationen, bei denen Datenwettläufe entstehen, viel komplizierter aus. Die Methode enthält nicht nur die Lese oder Schreiboperation, die den Datenwettlauf verursacht hat, sondern IF/ELSE - Anweisungen, Schleifen, Methodenaufrufe und vor allem mehrere Zugriffe auf das betroffene Feld.

Die Tatsache, dass in einer Methode ein Datenwettlauf erkannt worden ist, bedeutet, dass eine Änderung vom betroffenen Feld durch andere parallel laufende Threads während der Ausführung der Methode möglich ist. Ist das Feld mehrmals in der Methode vorhanden, dann könnte es zur Atomizitätsverletzung auf Methodenkörperebene kommen (siehe das Beispiel auf Abbildung 4.23).

```
1. public void Method1()
2. {
3.     value = 10;
4.     //An dieser Stelle könnte der Wert
5.     //von value durch Method2
6.     //überschrieben werden
7.     Console.WriteLine(value);
8. }
```

```
1. public void Method2()
2. {
3.     value = 15;
4. }
```

Abbildung 4.23: Dieses Beispiel zeigt, wie die Atomizität der Methode Method1 durch ihre parallele Ausführung mit der Methode2 verletzt sein könnte. Die Ausgabe in Zeile 7 kann entweder 10 oder 15 sein.

Atomizitätsverletzungen, die sich auf den Zustand des betroffenen Feldes beziehen [A4], werden in dieser Arbeit durch eine Feldabhängigkeitsanalyse erkannt. Die Korrekturmöglichkeit bei diesem Fehlertyp ist auf Methodenkörper eingeschränkt. Diese Einschränkung könnte aber in zukünftigen Arbeiten aufgehoben werden.

4.10.3.1 Die vier grundlegenden Zugriffsarten

Es gibt vier Fälle für die Zugriffe auf zwei Vorkommnisse desselben Feldes in einem Methodenkörper. Eine Atomizitätsverletzung ändert den Wert des Feldes zwischen den beiden Zugriffen:

1. (Lesen, Lesen): Beim ersten Lesen wird der Wert A gelesen und beim Zweiten der Wert B. B wurde von einem anderen Thread in das Feld gespeichert.
2. (Lesen, Schreiben): Der Wert A wird ausgelesen. Danach will der Entwickler einen Wert B in dasselbe Feld speichern. Wenn der Wert des Feldes inzwischen geändert wurde, dann wird er überschrieben. Dieser Fall wirkt sich noch gravierender aus, wie auf Abbildung 4.24 zu sehen ist.
3. (Schreiben, Schreiben): In diesem Fall wird ein aktueller Wert überschrieben, ohne dass der Thread das mitbekommt.
4. (Schreiben, Lesen): Es wird ein Wert in das Feld gespeichert und später wird ein anderer Wert ausgelesen.

```
1. int temp = value * 2;
2. //temp wird hier weiter verarbeitet
3. value = temp - amount;
```

Abbildung 4.24: Auf Zeile 1 wird ein Wert vom Feld `value` ausgelesen. Dieser beteiligt sich an der Berechnung von einem neuen Wert, der auf Zeile 3 wieder in `value` geschrieben wird. Wegen eines Datenwettlaufs wird der ausgelesene Wert nicht mehr aktuell. Als Ergebnis wird in `value` ein Wert geschrieben, der auf einen nicht aktuellen Wert basiert.

Die Frage, die man sich hier stellt, ist, ob die gerade beschriebenen Wertänderungen ein Problem sein können: Die Frage hat keine eindeutige Antwort und in allen vier Situationen können Probleme auftauchen. Um automatisch auf diese Frage zu antworten, benötigt ein Werkzeug Informationen über die Semantik der Anwendung. Diese sind an dieser Stelle nicht verfügbar. In den nächsten zwei Abschnitten wird erläutert, wie dieses Problem in der vorliegenden Arbeit gelöst ist.

4.10.3.2 Entwicklererwartungen und Atomizität

Wenn ein Feld in einem Codebereich vorkommt, dann hat der Entwickler eine von den folgenden drei Erwartungen:

1. Das Feld behält seinen (zugewiesenen) Wert und die lokal durchgeführten Änderungen innerhalb des entsprechenden Codebereichs. Der Entwickler verlässt sich darauf und überprüft nie, ob sich der Wert geändert hat.
2. Der Wert des Feldes wird durch einen parallel laufenden Thread geändert. Der Entwickler erwartet dieses Verhalten, ist vorbereitet damit umzugehen und unter Umständen wird diese Änderung sogar benötigt. Dieses Szenario findet sich durchaus in parallelen Programmen.
3. Eine Änderung des Feldes ist für die Anwendung irrelevant.

In ursprünglich sequenziellen Programmen, die man parallelisiert, sieht man Szenario 1 und 3. Um zu entscheiden, ob eine Situation zu Szenario 3 gehört, ist ohne semantische Kenntnisse und Entwickleraussagen kompliziert und fehleranfällig. Aus diesem Grund wird in dieser Arbeit angenommen, dass die Atomizitätsverletzung immer ein Problem verursacht.

Im Folgenden werden ausschließlich Szenario 1 (S1) und 2 (S2) betrachtet. Liegt S1 vor, müssen alle Vorkommnisse des Feldes im entsprechenden Bereich ununterbrochen geschützt werden. Liegt S2 vor, dann könnte das gemeinsame Schützen aller Vorkommnisse zu einer Verklemmung führen. Ob alle Vorkommnisse bei S3 zusammen geschützt werden oder nicht, spielt für die Anwendung keine Rolle.

4.10.3.3 Erkennen von Atomizitätsstellen: Das Stopper-Konzept

Die Herausforderung liegt nun darin, analytisch zwischen S1 und S2 zu unterscheiden. Das geht jedoch nicht vollständig ohne Kenntnis der Semantik des Programms. Es gibt jedoch Codesegmente, deren Existenz ein Hinweis auf Szenario 2 sein könnte. Diese Segmente werden in der weiteren Beschreibung Stopper genannt. Sie basieren auf der folgenden Überlegung:

Wenn eine Änderung des Feldes erwartet und benötigt wird, dann ist der Einbau eines Wartemechanismus im Code notwendig.

Semantisch bedeutet ein Stopper Folgendes: ”Der Entwickler erwartet an dieser Stelle eine Änderung des Zustandes durch einen parallel laufenden Thread.”

Als Stopper werden in dieser Arbeit Methodenaufrufe definiert, die den aktuellen Thread temporär blockieren. Einige davon sind:

1. Thread.Sleep
2. Thread.Join
3. Monitor.Wait
4. Process.WaitForExit
5. ...

Die Liste kann mit jeder Methode ergänzt werden, die den aktuellen Thread blockiert. Ziel dieses Abschnitts ist nicht das Stopper-Konzept vollständig und tief zu untersuchen, sondern die Idee vorzustellen.

Eine weiterführende Arbeit kann die Stopper um komplexe Code-Konstrukte erweitern und zusätzlich das genaue Verhalten der Methoden analysieren, um eine bessere Aussage über das vorliegende Szenario zu treffen.

Das weiterentwickelte Konzept geht davon aus, dass mithilfe der Stopper getroffene Entscheidung über das vorliegende Szenario korrekt ist.

4.10.3.4 Erkennung der korrekten Feldabhängigkeiten

Die einfachste Möglichkeit die in den letzten Abschnitten vorgestellte Atomizitätsverletzung zu vermeiden, wäre, alle Zugriffe auf das Feld im entsprechenden Bereich zu schützen. Vielleicht sind aber nicht alle von einem Datenwettlauf betroffen. Laut [E2] werden alle vorhandenen Wettlaufsituationen erkannt. Das Beispiel auf Abbildung 4.25 zeigt eine solche Situation.

```
1. value = RecalculateValue();
2. ...
3. int temp = value * 2;
4. ...
5. thread.Start();
6. ...
7. value = temp - amount;
```

Abbildung 4.25: Die Zugriffe auf `value` auf Zeile 1 und 3 verursachen keinen Datenwettlauf, weil nur ein Thread zu der Zeit im Programm existiert. Ein Datenwettlauf ist erst ab Zeile 5 möglich. Eine Atomizitätsverletztung könnte es erst ab Zeile 5 geben.

In der auf Abbildung 4.25 vorgestellte Situation ist es nicht notwendig den Zugriff auf `value` auf Zeile 1 zu schützen, weil dieser unter keinen Umständen vom Datenwettlauf betroffen ist. Aus diesem Grund werden während der Feldabhängigkeitsanalyse nur Feldvorkommnisse betrachtet, die von einem Datenwettlauf betroffen sind, zusätzlich noch das erste unbetroffene Feldvorkommnis.

Für jedes betroffene Vorkommnis werden immer nur die nächststehenden Abhängigkeiten gesucht, also Vorgänger- und Nachfolgervorkommnisse. Um diese zu bestimmen, wird die Ergebnisliste 4.10 aus der Kontroll-und Datenflussanalyse benötigt.

Vorgehensweise bei der Abhängigkeitssuche

In diesem letzten Schritt der Feldabhängigkeitsanalyse wird die Suche nach den korrekten Abhängigkeiten erläutert.

Gegeben: Methode M, mit Feld F, bei dem die Operation O an einem Datenwettlauf beteiligt ist.

Gesucht: alle direkten Vorgänger-und Nachfolgervorkommnisse von F in M.

Für jede Abhängigkeitsliste werden die folgenden Schritte ausgeführt:

1. Finde das Ereignis für O. Wenn kein Ereignis in dieser Liste vorhanden ist, wiederhole 1 mit der nächsten Liste. Wird das Ereignis gefunden, gehe zu 2.
2. Suche nach Vorgänger- und Nachfolgervorkommnis in der Liste. Die Suche fängt in den beiden Fällen beim Startereignis an und unterscheidet sich nur durch die genommene Richtung: oben für den Vorgänger, unten für den Nachfolger. Aus diesem Grund werden die zwei Fälle zusammengefasst. Gehe zu 3.
3. Iteriere durch die Ereignisliste, bis das passende Ereignis gefunden wird, bis ein Stopper entdeckt wird, oder bis alle Elemente durchsucht worden sind:
   a) Untersuche das nächste Ereignis.
   b) Ist das Ereignis in derselben Anweisung wie das Startereignis? Gehe zu e. Anweisung ist die kleinste zu schützende Einheit. Aus diesem Grund werden Ereignisse von demselben Statement ignoriert.
   c) Ist das Ereignis ein READ- oder ein WRITE-Ereignis, vergleiche die Ereignisfelder. Sind die gleich, gehe zu 4, sonst zu e.
   d) Ist das Ereignis ein CALL-Ereignis?
      i. Hat die aufgerufene Methode ein Code Object Model und ist die Methode noch nicht besucht? Wenn ja, dann markiere die Methode als besucht, um Zyklen zu vermeiden und gehe zu ii. Sonst gehe zu e.
      ii. Suche im Methodenkörper nach einem Lese- oder nach einem Schreibereignis mit demselben Feld, wie das Startereignis. Sind CALL-Ereignisse enthalten, bearbeite diese nach demselben Prinzip. Wird ein Lese- bzw. Schreibereignes in dieser oder in einer der aufgerufenen Methoden entdeckt, markiere das ganz erste CALL Ereignis als abhängig und gehe zu 4, sonst gehe zu e.
   e) Gibt es noch nicht untersuchte Ereignisse, gehe zu a, ansonsten zu 4.
4. Werden Abhängigkeitsereignisse gefunden, speichere diese.

Das Master/Worker-Evaluiehrungsbeispiel 6.3.2.3 zeigt, wie viele Aspekte dieses Verfahrens funktionieren.

Endergebnis

Nachdem alle Abhängigkeitsereignisse abgearbeitet sind, werden von 0 bis n verschiedene Vorgänger-und Nachfolgerereignisse, je nach Lage des Startereignisses, gefunden.

Für alle diese werden die einfachen Anweisungen bestimmt und die werden, zusammen mit dem betroffenen Statement in die Liste der zu schützenden Codeteilen gespeichert.

4.10.4 Schützen von Fehlerstellen im externen Code

Datenwettläufe können an zwei Stellen auftauchen:

1. in eigenen Code
2. in externen Code

Alle Analysen und Korrekturvorschläge, die bis jetzt vorgestellt wurden, sind nur im ersten Fall anwendbar. Für den externen Code [A3] ist der Quelltext nicht vorhanden. Aus diesem Grund kann weder das Code Object Model aufgebaut werden noch die Korrektur im Quelltext eingebaut werden.

Am Anfang dieser Diplomarbeit wurde versucht, die Änderungen zuerst auf Assemblyebene und erst dann auf Quellcodeebene zu speichern. Dieses Vorgehen hat die Arbeit aus den folgenden Gründen nicht weitergebracht:

- Komplexere Änderungen wie der Einbau von LOCK-Blöcken auf Assemblyebene, ohne die Semantik des Codes zu verletzten, wurde nicht erreicht.
- Das neue Programm und der vorhandene Quelltext stimmen nach einer Änderung auf Assemblyebene nicht mehr überein. Das macht die Analyse mithilfe des Quelltextes nicht möglich.
- Außer erhöhter Performanz für eine weitere Iteration hat dieses Vorgehen keine anderen Vorteile.

Das in diesem Abschnitt vorgestellte Verfahren betrachtet nur Wettlaufsituationen, die durch Änderung im eigenen Code korrigierbar sind. Wenn der Datenwettlauf zum Beispiel durch Threads entsteht, die im externen Code erzeugt werden, dann ist dieser nicht korrigierbar (siehe Abbildung 4.26).

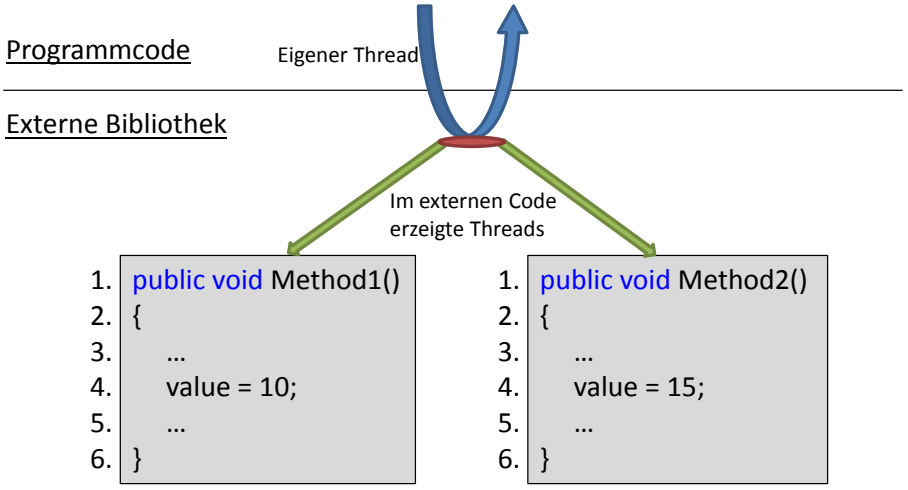

Abbildung 4.26: Ein eigener Thread ruft eine externe Methode auf. In der aufgerufenen Methode erstellt dieser Thread n neue die eine Wettlaufsituation verursachen.

Die betroffene Codestelle liegt im externen Code und im eigenen wird diese durch einen Methodenaufruf dargestellt.

Weil keine Informationen vorliegen, was genau den Datenwettlauf verursacht hat, hat man auch nicht viele Korrekturmöglichkeiten. Auch keine der bisher beschriebenen Analysen kann durchgeführt werden. Die einzige Möglichkeit im allgemeinen Fall ist:

- Schütze den Methodenaufruf vor parallelem Ausführen durch einen LOCK-Block.

Methodenaufrufe sind keine Felder und die Fehlersteller können auch nicht nach den betroffenen Daten gruppiert werden. Eine andere Strategie ist an dieser Stelle notwendig, um festzustellen, welche Methodenaufrufe in einer Datenwettlaufgruppe zusammengehören und mit derselben Sperre geschützt werden müssen. Es gibt mehrere Möglichkeiten, die zum Einsatz kommen könnten:

1. Nur Methoden, die zu derselben Klasse gehören, werden gruppiert.
2. Nur Methoden, die von der Testumgebung zusammen gemeldet werden, werden gruppiert.
3. positionsbedingte Gruppierung

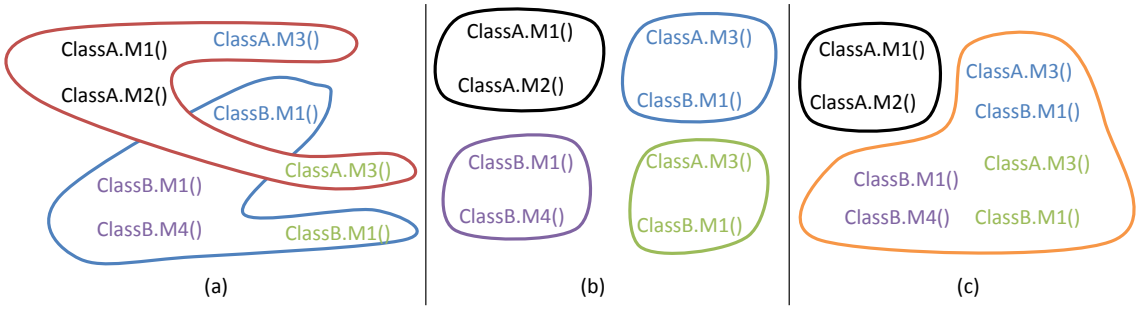

Abbildung 4.27: Drei Möglichkeiten, um Methodenaufrufe zu gruppieren. Methodenaufrufe, die Teil desselben Datenwettlaufs sind, werden mit derselben Farbe gekennzeichnet.

Beim ersten Ansatz werden Methodenaufrufe falsch gruppiert (siehe Abbildung 4.27 (a)). Der zweite Ansatz ist zu fein granular und kann zu mehrfachem Schützen derselben Stelle führen (siehe Abbildung 4.27 (b)). Beim dritten Ansatz geht es darum, gemeldete Methodenaufrufe nach gemeinsamen Teilen zu gruppieren. Ein Datenwettlauf besteht immer aus zwei Verursachern. Wenn zwei Gruppen mindestens ein übereinstimmendes Element/eine übereinstimmende Fehlerstelle haben, dann gehören sie zusammen und werden von derselben Sperre geschützt (siehe Abbildung 4.27 (c)). Die Relation ist transitiv und kann eine Kette von Gruppen definieren. Diese Einsatzmöglichkeit wird in dieser Arbeit verwendet.

4.10.5 Externe Wettlaufsituationen in Datenstrukturen

Eine der Quellen von Wettlaufsituationen in externem Code sind thread-unsichere Datenstrukturen, wie zum Beispiel Listen und Hash-Tabellen.

Einige davon unterstützen parallele Lesezugriffe, aber in dem Moment, in dem ein Schreibzugriff (Hinzufügen von Elementen) gemacht wird, können viele unvorhersehbare Seiteneffekte verursacht werden. Diese können die komplette Datenstruktur in einen fehlerhaften Zustand versetzen. Einige Beispiele dafür sind:

- Falsche Reihenfolge der Elemente.
- Falsch gemeldete Anzahl der Elemente.
- Fehlende Elemente.
- Fehlerhafte Zeigerstruktur.

Die klassischen Datenstrukturen sind aus Performanzgründen nicht auf Parallelität vorbereitet.

Alle Datenwettläufe, die durch die Verwendung ungeeigneter Datenstrukturen verursacht werden, werden wie die früher beschriebenen Datenwettläufe im externen Code erkannt (siehe Abbildung 4.28 (a)). Das bedeutet, dass alle solchen Wettlaufsituationen durch Sperren der entsprechenden Methodenaufrufe korrigiert werden können (siehe Abbildung 4.28 (b)). Speziell für Datenstrukturen bietet diese Arbeit aber auch eine weitere Möglichkeit an:

- Tausche die Datenstruktur gegen eine thread-sichere Variante mit äquivalenter Funktionalität aus (siehe Abbildung 4.28 (c)).

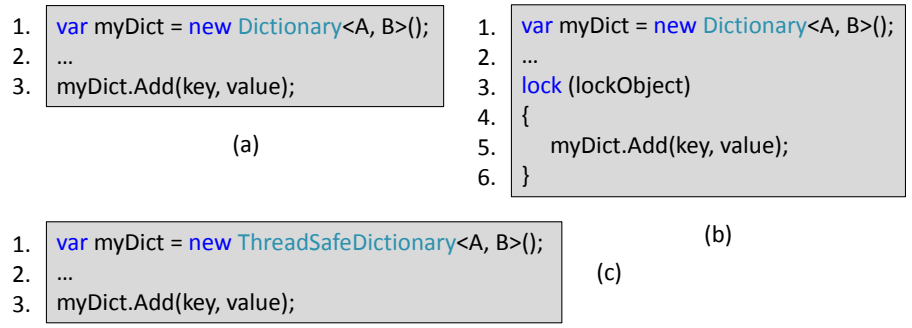

Abbildung 4.28: (a) - Ausgangssituation. Datenwettlauf auf Zeile 3 in der Add() Methode; (b) - Schützen des Methodenaufrufes mit einem Lock-Block; (c) - Austausch der Datenstruktu.

Die Verwendung thread-sicherer Datenstrukturen hat in diesem konkreten Fall diverse Vorteile gegenüber der Verwendung von Lock-Blöcken:

- Durch die Evolution des Codes können keine neuen Datenwettläufe eingebaut werden.
- Diese können Sicherheit in einer parallelen Umgebung gewährleisten, in der nicht nur Sperren verwendet werden, sondern auch andere sperrenfreie Mechanismen. Wie sperrenfreie Datenstrukturen auf der Basis von atomaren Operationen wie `Compare and Swap` implementiert werden können, wird zum Beispiel in [Ale07] gezeigt.
- Thread-sichere Datenstrukturen können auch das Problem der Skalierbarkeit besser lösen.

4.10.5.1 Definition und Wahl der Ersatzdatenstrukturen

Von der ausgewählten Ersatzdatenstruktur ist der Erfolg dieser Art von Korrektur abhängig:

1. Als ein Codeteil beteiligt sich die Datenstruktur nicht nur an den Aufrufen, die den Datenwettlauf verursacht haben.
2. Die Referenz auf die Datenstruktur kann von Feld zu Feld übergeben werden.
3. Die Referenz auf die Datenstruktur kann als Methodenparameter oder -rückgabewert verwendet werden.
4. Die Datenstrukturen sind in den objekt orientierten Programmiersprachen Teil einer Vererbungshierarchie und implementieren Schnittstellen, die den Umgang erleichtern.

5. Polymorphie.

Anhand dieser Liste kann man Folgendes schließen: Die Ersatzdatenstruktur muss mit der thread-unsicheren Datenstruktur kompatibel sein, damit die Kompilierbarkeit des Codes unverletzt bleibt. Eine Möglichkeit diese Kompatibilität zu erzwingen ist, eine Anforderung an die Ersatzdatenstruktur zu stellen:

- Die Ersatzdatenstruktur muss von der Thread-unsicheren erben und dieselben Schnittstellen implementieren.

Was spricht gegen einen Austausch mit einer inkompatiblen Datenstruktur, die aber eine äquivalente Funktionalität anbietet? Die Verwendung von nicht verwandten Ersatzdatenstrukturen ist möglich, aber mit einer komplexen Refaktorisierung und semantischen Änderungen verbunden, wie das nächste Beispiel zeigt.

```
1.  public void Evaluate(ICollection<String> collection)
2.  {
3.      ...
4.      if (collection is List<String>)
5.      {
6.          //Evaluate as List
7.      }
8.      else if (collection is Queue<String>)
9.      {
10.         // Evaluate  as Queue
11.     }
12.     else if (collection is Stack<String>)
13.     {
14.         // Evaluate as Stack
15.     }
16.     ...
17. }
```

Abbildung 4.29: Die dargestellte Methode bearbeitet Datenstrukturen, in dem verschiedenen Strategien, in Abhängigkeit vom Typ der Datenstruktur, verwendet werden.

Auf Abbildung 4.29 sieht man eine Methode, die an mehreren Stellen aufgerufen wird. An einer dieser Stellen wird eine von einem Datenwettlauf betroffene Liste als Argument übergeben. Diese Datenstruktur wird gegen eine nicht kompatible ausgetauscht.

Nach dem Austausch kompiliert der Code nicht mehr, weil die neue Datenstruktur nicht an die Methode übergeben werden kann. Es gibt mehrere offene Fragen, die eine korrekte Anpassung in diesem Fall verhindern:

- Sollen alle übergebenen Argumentenquellen angepasst werden, nur weil eine der Datenstrukturen Parallelitätsprobleme hat?
- Was passiert wenn die Signatur der Methode angepasst wird, aber die Ausführungsreihenfolge von dem Typ des Eingabeparameters abhängig ist, (wie auf Abbildung 4.29)
- Was passiert es, wenn die Methode `public` ist und von extern aufgerufen wird?

Alle diese Probleme existieren nicht, wenn die Ersatzdatenstruktur eine Unterklasse der zu ersetzenden ist.

Im nächsten Abschnitt wird eine andere Möglichkeit für Austausch gegen eine inkompatible Ersatzdatenstruktur vorgestellt.

4.10.5.2 Die Adapter-Datenstrukturen

Liegt eine passende Ersatzdatenstruktur vor, sodass die semantische Funktionalität der thread-unsicheren auf ihr abgebildet werden kann, so lässt sich diese mithilfe einer Adapter-Datenstruktur verwenden.

Technische Eigenschaften der Adapter-Datenstruktur:

1. Sie erbt von der Datenstruktur, die ersetzt wird.
2. Sie implementiert dieselben Schnittstellen, wie die Datenstruktur, die ersetzt wird.

Die Funktionalität der Adapter-Datenstruktur beschränkt sich auf das Abbilden der erwarteten Schnittstelle auf die von der inkompatiblen Datenstruktur angebotene, gemäß der Definition des Adapter-Entwurfsmusters in [GHJV04]. In Abbildung 4.30 ist dieser Prozess visualisiert.

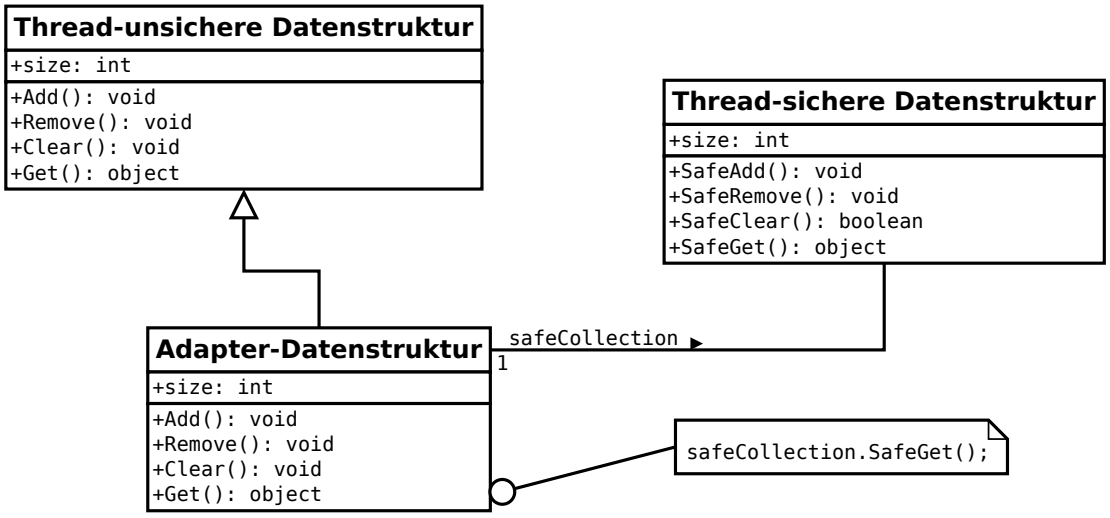

Abbildung 4.30: Adapter-Datenstruktur und ihre Beziehung zu der thread-sicheren und der thread-unsicheren Datenstrukturen.

Aus der Definition der technischen Eigenschaften der Adapter-Datenstruktur wird klar, dass diese mit der thread-unsichere Datenstruktur vollständig kompatibel ist. Für den eigenen Code ist dann die Adapter-Datenstruktur der Ersatz. Die Datenstruktur, die die thread-sichere Funktionalität anbietet, bleibt transparent.

Mithilfe der Adapter-Datenstruktur kann jede beliebige Datenstruktur als Ersatzdatenstruktur verwendet werden, soweit semantische Kompatibilität vorliegt.

Als Weiterentwicklung dieses Konzepts könnte ein Analyseverfahren die beste Ersatzmöglichkeit von mehreren auswählen, in dem Nutzungsszenarien, Zugriffe in der Anwendung und Effizienz betrachtet werden.

4.10.5.3 Erkennen von Wettlaufsituationen in Datenstrukturen

Um Datenstrukturen auszutauschen, muss zuerst erkannt werden, ob überhaupt eine Wettlaufsituation in der Datenstruktur vorliegt. Als Zweites muss auch noch der genaue Typ der Datenstruktur bestimmt werden.

Damit ein mögliches Detektionsverfahren beschrieben werden kann, müssen einige Einschränkungen eingeführt werden:

1. Das Erstellen der Datenstruktur lässt sich bis zu einer oder mehreren Stellen im eigenen Quellcode verfolgen.
2. Die Datenstruktur kommt nicht als Eingabeparameter von externem Code.
3. Die Datenstruktur kommt nicht als Rückgabewert eines externen Methodenaufrufs.

Eine Vorgehensweise zum Erkennen von Datenstrukturen kann folgendermaßen aussehen:

1. Gehört die aufgerufene Methode zu einer Datenstruktur, gehe zu Schritt 2; beende die Suche andernfalls.
2. Bestimme das Instanzobjekt.
3. Verfolge das Instanzobjekt zum Ort seiner Instanzierung.
4. Für jede gefundene Stelle wird der Typ des aufgerufenen Konstruktors bestimmt. Das ist der Typ der Datenstruktur. Ist die Datenstruktur nicht thread-sicher, füge die Stelle einer Ergebnisliste hinzu.

Die Verfolgung in Schritt 3 erfolgt über den Datenflussgraphen, der in Abschnitt 4.8 vorgestellt wurde.

Verfolgungsverfahren

Dieses Verfahren beschreibt, wie eine Datenstruktur zu ihren möglichen Erstellungsquellen zurück verfolgt wird. Die Verfolgung startet beim Instanzobjekt, das die vom Datenwettlauf betroffene Methode aufruft:

1. Bestimme alle Kontexte, die den Methodenaufruf enthalten.
2. Verfolge für jeden Kontrollfluss die Datenstruktur bis zu der ersten Erstellungsquelle.
3. Existieren Kontrollflüsse, in denen keine Quelle gefunden wurde, bestimme anhand des Aufrufgraphen alle Methoden, die die gerade untersuchte Methode aufrufen. Wurde für alle relevanten Kontrollflüsse eine Quelle gefunden, breche ab.
4. Führe Schritt 2 für jede gefundene Aufrufmethode aus.

Schritt 2 muss detaillierter erklärt werden. In den unteren Überlegungen ist zu beachten, dass es sich um die Verfolgung eines Referenztyps handelt:

- Wird ein Schreibzugriff auf dem gerade verfolgten Objekt entdeckt, dann wird der Schreiber weiterverfolgt. Dies entspricht einer Referenzenübergabe.
- Wird ein Methodenaufruf entdeckt, dann wird die Methode von unten nach oben durchsucht - iterativ mit weiteren Untermethoden.
- Wird ein Konstruktoraufruf entdeckt, breche ab und speichere den Konstruktor in der Ergebnisliste.
- Beim Durchsuchen einer Methoden werden die verschiedenen Kontrollflüsse betrachtet.
- Ist die gerade untersuchte Variable ein Argument der Methode, so wird die Quellvariable bestimmt und weiterverfolgt. Wer die Aufrufer sind, wird anhand des Aufrufgraphen bestimmt.
- Aufgerufene Methoden werden nur dann untersucht, wenn:
  - Die gerade untersuchte Variable ein Klassenfeld ist.
  - Wenn die Methode in das verfolgten Feld schreibt.

Diese Analyse liefert eine Liste von Konstruktoraufrufen zurück. Anhand des Typs der Konstruktoren werden die Typen aller Datenstrukturen bestimmt, die an der Fehlerstelle die betroffene Methode aufrufen.

4.10.5.4 Austausch der Datenstrukturen

Nachdem alle Erstellungsquellen gefunden worden sind, können die Datenstrukturen gegen thread-sichere Implementierungen ausgetauscht werden. Die Ersatzdatenstrukturen erfüllen die Bedingungen, die früher in diesem Kapitel definiert wurden. Die Korrektur erfolgt folgendermaßen:

- Ersetze die alten Datenstrukturen, in dem die alten Konstruktoraufrufe gegen die neuen ausgetauscht werden.

Es reicht an dieser Stelle, dass man die Namen der Konstruktoren austauscht. Die Ersatzdatenstrukturen sind kompatibel und deswegen ist keine zusätzliche Änderung des Quellcodes notwendig. Ausnahmen dieser Regel existieren zum Beispiel in den folgenden Fällen:

- Referenzieren der notwendigen externen Bibliotheken
- Importieren der neuen Datenstruktur im Quelltext

Diese sind aber spezifisch für die verwendete Technologie und Ersatzdatenstruktur und beeinflussen das allgemeine Konzept nicht.

4.11 Bestimmen geschützter Regionen

Verschiedene Teile des bereits beschriebenen Konzepts bestimmen, zusätzlich zu der direkt betroffenen Anweisung, weitere von ihr abhängige einfache und zusammengesetzte Anweisungen, die auch zu schützen sind [A4, A5]. Alle Anweisungen liegen unsortiert in einer Liste. Doppelte oder überlappende Einträge können auch vorhanden sein.

Ein weiteres Problem sind die Beziehungen, die zwischen den Anweisungsblöcken entstehen:

- Schachtelung
- Vollständige oder teilweise Überlappung

Ein weiterer Punkt, der in diesem Abschnitt besprochen wird, ist das Schützen von bereits geschützten Regionen [A7] und was deren Existenz verursachen kann.

4.11.1 Die Initialregionen

Wie in der Einleitung zu diesem Abschnitt beschrieben wurde, enthält die Ergebnisliste Anweisungen in unsortierter Reihenfolge, die keinen Block bilden.

Mit einer solchen Liste kann die Mutationsphase den Quelltext nicht korrigieren. Damit Lock-Blöcke eingebaut werden können, müssen die geschützten Anweisungen einen Block bilden. Das heißt, die erste und die letzte Anweisung haben dieselbe Codetiefe [E5].

In diesem Abschnitt des Kapitels wird beschrieben, wie aus der Ergebnisliste ein Block entsteht, der sich durch einen LOCK-Block schützen lässt.

Bei der Bestimmung der Regionen werden folgende Regeln gefolgt:

1. Die definierte Region muss alle Anweisungen der Liste enthalten.
2. Liegt der Anfang oder das Ende einer zusammengesetzten Anweisung in der Liste, dann wird diese vollständig zur Region inkludiert.

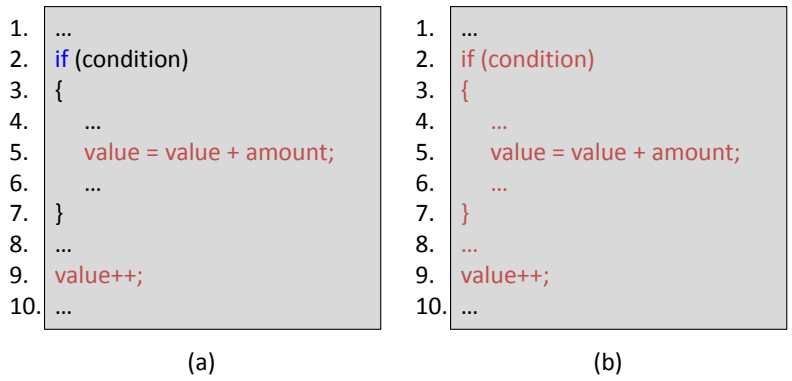

Abbildung 4.31: (a) - Ausgangssituation: Die roten Zeilen sind die Anweisungen, die zu schützen sind. (b) - Abschlusssituation: Die roten Zeilen definieren die Region.

Punkt 2 garantiert auch, dass der Beginn und das Ende der Region auf einer Ebene liegen, entweder im Körper einer zusammengesetzten Anweisung oder direkt im Methodenkörper.

Eine Region wird folgendermaßen bestimmt:

1. Von allen Anweisungen in der Liste, bestimme die Operation startOp mit dem kleinsten und die Operation endOp mit dem größten Methoden-Offset.
2. Bestimme alle einfachen Anweisungen im Intervall [startOp, endOp].
3. Mithilfe vom Code Object Model bestimme aus den einfachen Anweisungen die zusammengesetzten:
   a) Ist eine einfache Anweisung der Beginn einer zusammengesetzten, inkludiere die zusammengesetzte in der Region und überspringe alle enthaltenen einfachen Anweisungen.
   b) Ist eine einfache Anweisung das Ende einer zusammengesetzten, lösche alle bisherigen Anweisungen aus der Region und inkludiere die Zusammengesetzte.
   c) Ist die Anweisung kein Beginn oder Ende einer zusammengesetzten Anweisung, dann wird diese in der einfachen Form in der Region übernommen.
4. Gebe die definierte Region zurück.

Abbildung 4.31 zeigt die Ausgangs- und die Abschlusssituation dieses Verfahrens.

4.11.2 Bereits geschützte Regionen

Wie früher in diesem Kapitel erklärt wurde, macht dieser Ansatz keine Versuche, die Sperren vom Code abzuleiten, sondern verwendet ausschließlich neue Sperren. Dies gilt bis auf die beiden folgenden Ausnahmen:

1. Die Sperre wurde von einer früheren Iteration des Werkzeuges erzeugt.
2. Der Entwickler hat die Sperre entsprechend annotiert, um die geschütztes Feld/Sperre-Relation zu definieren.

In diesem Abschnitt werden nur Sperren betrachtet, für die eine der oberen zwei Punkte gilt. Regionen, die nicht in ihrer Methode oder von anderen Sperren geschützt sind, werden ignoriert.

Wettlaufsituationen können in bereits geschützten Regionen gemeldet werden, wenn:

1. der Entwickler Codeänderungen implementiert hat und das geschützte Feld an einer parallel ausgeführten Stelle ungeschützt verwendet hat.
2. der Entwickler die Sperre/Feld-Beziehung manuell im Code festgelegt hat, aber der LOCK-Block an einer Stelle vergessen wurde.
3. Die Testumgebung hat neue, früher nicht erkannte Wettlaufsituationen entdeckt.

Fall 3 wird wegen [E2] an dieser Stelle ausgeschlossen. Das limitiert aber die Fähigkeiten des Verfahrens nicht - es ist auch in diesem Fall in der Lage, sinnvoll zu reagieren.

Eine Region gilt genau dann als bereits geschützt, wenn:

1. Diese vollständig in einem Lock-Block derselben Methode liegt. Der LOCK-Block darf auch größer sein als die definierte Region.
2. Die Sperre/Feld-Beziehung zwischen der Sperre des LOCK-Blocks und dem betroffenen Feld der Region gilt.

Eine bereits geschützte Region wird kein zweites Mal geschützt. Die Erkennung solcher Regionen erfolgt nach dem folgenden Muster:

1. Iteriere mithilfe des Code Object Models durch alle übergeordneten Anweisungen der ersten Anweisung der Region und speichere die gefundenen LOCK-Blöcke in einer Liste.
2. Iteriere durch die gefundenen LOCK-Blöcke von Innen nach Außen und teste, ob die Sperre mit dem Sperrobjekt der Region übereinstimmt. Ist das der Fall, markiere die Region als bereits geschützt.

Eine Überprüfung, ob alle Anweisungen der Region in dem LOCK-Block enthalten sind, ist nicht nötig. Laut der Definition der Region liegen alle Anweisungen entweder in einer übergeordneten Anweisung oder direkt im Methodenkörper. Wird eine Region als bereit geschützt erkannt, dann wird diese nicht weiterbetrachtet.

4.11.3 Beziehungen zwischen Regionen

Wegen der Verwendung von LOCK-Blöcken müssen die Positionen der einzelnen Regionen auch in Betracht genommen werden. Es muss sichergestellt werden, dass keine geschachtelten oder überlappenden Regionen existieren. Diese Beziehungen können das korrekte Hinzufügen der LOCK-Blöcke verhindern.

Die Beziehungen werden in zwei Gruppen unterteilt:

- zwischen Regionen desselben Objekts
- zwischen Regionen verschiedener Objekte (siehe Abbildung 4.32).

An dieser Stelle können Duplikatregionen entfernt werden, sofern diese zu derselben Datenwettlaufgruppe gehören.

Regionenanpassung bei erkannten Beziehungen

Beziehungen zwischen Regionen desselben Objekts sind wegen der durchgeführten Analyse begrenzt möglich und nur in Verbindung mit dem Stopper-Konzept zu finden (siehe das Beispiel auf Abbildung 4.33). Schachtelung ist die einzige Beziehung in diesem Fall und diese kann durch Entfernen der kleineren Region beseitigt werden.

Die Behandlung der Beziehungen wird komplexer, wenn Regionen aller betroffenen Felder betrachtet werden:

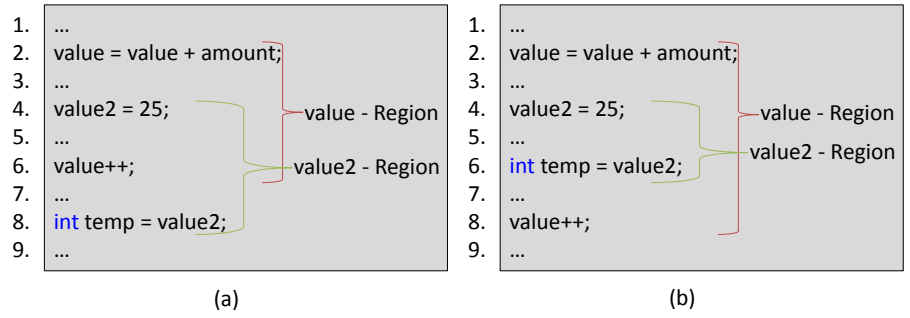

Abbildung 4.32: (a) - zwei überlappende Regionen. (b) - zwei geschaltete Regionen.

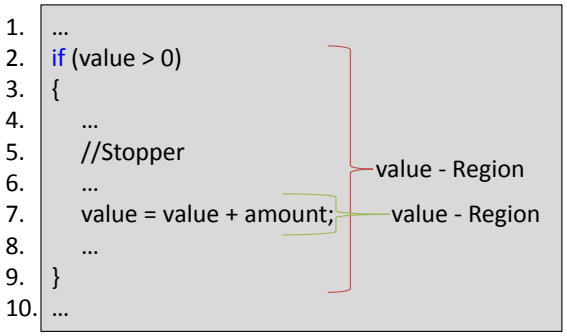

Abbildung 4.33: Der Stopper auf Zeile 5 verhindert den Aufbau der Abhängigkeit zwischen den Vorkommnissen von `value` auf Zeile 7 und auf Zeile 2, was zum Erstellen einer gemeinsamen Region führen würde. Die Variable `value` liegt auf Zeile 2 in einer Bedingung. Aus diesem Grund wird die gesamte IF/ELSE-Anweisung geschützt.

- Schachtelung: Eine Region des Feldes A liegt vollständig in einer Region des Feldes B. Diese Situation ist harmlos, wenn zwei Bedingungen erfüllt sind. Sind diese verletzt, dann muss die kleinere Region um die in ihr fehlenden Teile der größeren ergänzt werden:
  - Es existiert keine Region des Feldes B, die vollständig in einer Region des Feldes A liegt.
  - Das Sperren der beiden Regionen in dieser geschachtelten Form verursacht keinen Zyklus im Sperrengraphen.
- Überlappung: Eine Region des Feldes A und eine Region des Feldes B haben gemeinsame Anweisungen am Anfang oder am Ende. In diesem Fall müssen die beiden Regionen solange erweitert werden, bis eine vollständige Überlappung erreicht wird.

Nach der Ausführung dieser Schritte entstehen nur Regionen, die sich von LOCK-Blöcken problemlos schützen lassen. Diese Schritte müssen wiederholt ausgeführt werden, bis keine Änderungen in den Regionen stattfinden.

4.12 Ordnung von Sperren

Die Analyse, die im letzten Schritt durchgeführt wurde, garantiert Folgendes:

1. Es existieren keine überlappenden Regionen.
2. Wenn Regionen des Feldes A in Regionen des Feldes B geschachtelt sind, dann sind keine Regionen des Feldes B in Regionen des Feldes A geschachtelt.

Diese Analyse wurde nur auf Methodenkörperebene durchgeführt, weil bei einer Wettlaufsituation in Methode A nur Bereiche von A angepasst werden [E3].

Das bedeutet, dass ein LOCK-Block die Grenzen seines Methodenkörpers nicht verlassen darf, bedeutet aber auch, dass die Entstehung von Zyklen im Sperrgraphen und dadurch potenzielle Verklemmungssituationen nicht auszuschließen ist (siehe Abbildung 4.34) [A2.2]. Diese Zyklen entstehen wegen Methodenaufrufen und Beziehungen zwischen neuen und bereits existierenden Lock-Blöcken. Die Verklemmung kann in diesem Fall vermieden werden, wenn die Lock-Blöcke in die Aufrufmethoden ausgelagert werden. Das kann aber weitere Probleme und Spezialfälle verursachen, die in einer weiterführenden Arbeit untersucht werden können.

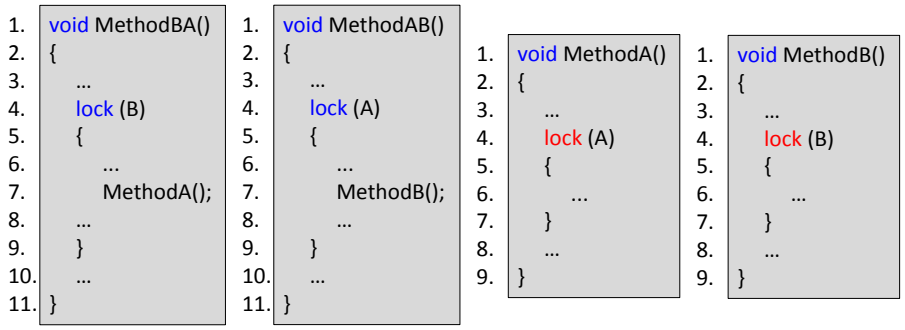

Abbildung 4.34: Rote Lock-Blöcke sind neu definiert, blauen bereits im ursprünglichen Code enthalten.

Während der Regionenanalyse wurden die Regionen so weit wie möglich lokal angepasst, sodass keine Probleme durch die neuen Lock-Blöcke entstehen.

In diesem Abschnitt werden nicht nur die eigenen LOCK-Blöcke betrachtet, sondern alle, die im Programm existieren. Dabei wird auch methodenübergreifend analysiert. Die Ergebnisse sind aber nur informativ. Bei einem gemeldeten Zyklus wird nicht versucht, diesen zu beheben. Dafür existieren zwei Gründe:

- die Beteiligung von bereits vorhandenem Code und LOCK-Blöcken
- die Tatsache, dass jede Verschiebung eines LOCK-Blocks neue Sperrenabhängigkeiten und dadurch neue Zyklen erzeugen könnte. So kann die Analyse in eine Endlosschleife geraten.

Eine weiterführende Arbeit kann alle möglichen Situationen untersuchen und Vorschläge zur Refaktorisierung des Codes machen. Im allgemeinen Fall kann die Reihenfolge der Lock-Blöcke getauscht werden, in dem der innere LOCK-Block expandiert wird. Diese ist eine einfache Refaktorisierung, die viele Konsequenzen haben kann. Nimmt man die methodenübergreifende Reihenfolge mit in Betracht, so steigt die Komplexität weiter.

Vor dem Test auf Zyklusfreiheit muss überprüft werden, dass für gleiche Regionen, die aber verschiedener Variablen gehören, unbedingt eine feste Reihenfolge definiert ist. Nur dann kann auch die Reihenfolge der LOCK-Blöcke festgelegt werden. Gibt es keine solchen Regionen, dann kann dieser Schritt übersprungen werden und direkt der Test auf Zyklusfreiheit durchgeführt werden.

4.12.1 Bestimmung der Reihenfolge der LOCK-Blöcke

Eine feste Reihenfolge zu bestimmen, ist wie bereits erwähnt für Situationen notwendig, in denen dieselbe Region durch mehrere LOCK-Blöcke geschützt werden muss (siehe Abbildung 4.35). Für diese Stellen ist die Reihenfolge der Schutzblöcke von besonderer Bedeutung. Diese sind auch die Stellen, an denen noch Änderungen vorgenommen werden können, um die Verklemmungsgefahr [A2.2] zu minimieren. Das folgende Verfahren bestimmt die Reihenfolge der LOCK-Blöcke.

```
1.  lock (A)
2.     {
3.     lock (B)
4.        {
5.        lock (C)
6.        {
7.           //Region
8.        }
9.     }
10. }
```

```
1.  lock (A)
2.     {
3.     lock (C)
4.        {
5.        lock (B)
6.        {
7.           //Region
8.        }
9.     }
10. }
```

```
1.  lock (B)
2.     {
3.     lock (A)
4.        {
5.        lock (C)
6.        {
7.           //Region
8.        }
9.     }
10. }
```

```
1.  lock (B)
2.     {
3.     lock (C)
4.        {
5.        lock (A)
6.        {
7.           //Region
8.        }
9.     }
10. }
```

Abbildung 4.35: 4 von den 6 verschiedenen Möglichkeiten eine Region mit 3 Sperren zu schützen.

Es existiert eine Menge von Regionen vom oben beschriebenen Typ. Für die Sperren, die diese Regionen schützen, muss eine feste Reihenfolge definiert werden. Aus diesem Grund werden diese in einer Menge gespeichert.

Vorgehensweise

1. Erweitere den existierenden Sperrgraphen, um die fest vorliegenden Reihenfolgen, die durch die geschachtelten und die einzelnen LOCK-Blöcke festgelegt werden. Dabei werden auch Methodenaufrufe betrachtet. Ist der Graph zyklusfrei, gehe zu 2. Ansonsten könnte die geplante Mutation eine Verklemmung verursachen.

2. Erstelle eine leere Liste, in der die Reihenfolge der Sperren definiert wird.
3. Für jedes Element der Sperrenmenge werden die folgenden Schritte ausgeführt:
   a) Enthält der Sperrengraph Informationen über die Sperre, gehe zu b. Sonst zu e.
   b) Bestimme die Schnittmengen der Elemente der sortierten Liste mit den Vorgänger- und mit den Nachfolgerelementen aus dem Sperrgraphen.
   c) Ist die Vorgängerschnittmenge nicht leer, füge die Sperre direkt nach dem letzten Element der Liste, das auch in der Schnittmenge enthalten ist. Ist die Vorgängerschnittmenge leer, gehe zu d, sonst zu f.
   d) Ist die Nachfolgermenge nicht leer, füge die Sperre direkt vor dem ersten Element der Liste, das auch in der Schnittmenge enthalten ist. Ist die Nachfolgermenge leer, gehe zu e, sonst zu f.
   e) Füge die Sperre am Ende der Liste ein. Gehe zu f.
   f) Gibt es noch unsortierte Sperren, dann gehe zu a, mit der nächsten Sperre, sonst zu 4.
4. Die Reihenfolge der Sperren in der Liste entspricht der Reihenfolge, in der die LOCK-Blöcke sortiert werden.

Jetzt existiert für jede Sperre und entsprechend für jeden LOCK-Block eine feste Reihenfolge. Im nächsten Abschnitt wird getestet, ob die Sperren einen Zyklus im Sperrgraphen verursachen.

4.12.2 Test auf Zyklusfreiheit

Der Test auf Zyklusfreiheit (siehe Abschnitt 4.7) stellt fest, ob die geplanten Änderungen eine Verklemmungssituation verursachen könnten [A2.2]. Ist das der Fall, wird das Verfahren unterbrochen.

Der Sperrengraph wird neu, mit den geplanten Änderungen aufgebaut.

Ist der resultierende Graph zyklusfrei, dann wird mit der Mutationsphase weitergemacht.

4.13 Die Mutationsphase

Die Mutationsphase ist der letzte Schritt des Verfahrens. Diese bekommt eine Liste von Datenwettlaufgruppen. Jede Gruppe enthält eine Menge von Codestellen, die mit der assoziierten Sperre zu schützen sind. Jede Stelle definiert entweder eine Standardänderung mit Start- und Endposition des zu schützenden Bereichs oder eine spezielle Änderung wie Extrahieren einer Bedingung. Ziel ist es, die vorbereiteten Änderungen im Quelltext fehlerfrei einzubauen [A8]. Dabei wird Folgendes beachtet:

- Jede Änderung definiert ihre Einbaustelle im Quelltext. Diese bezieht sich aber auf die Originalversion der Datei.
- In einer Datei müssen Änderungen an mehr als einer Stelle eingebaut werden.
- Mehrere Änderungen für dieselbe Stelle existieren. Beispiel: mehrere LOCK-Blöcke, die dieselbe Region schützen.
- Eine Änderung kann an mehr als einer Stelle eine Mutation verursachen.
- Wird eine neue Sperre erstellt, muss die Region, die von ihr geschützt ist, auf sie zugreifen können.

Die Mutation selbst erfolgt nach dem folgenden Muster:

1. Sortiere alle Änderungen aufsteigend nach ihrer Einbaustelle. Bei gleichen Stellen wird die Sperrenordnung genommen. Diese Reihenfolge erlaubt das Speichern aller Änderungen mithilfe eines Verschiebungsindexes.
2. Iteriere durch die einzelnen Fehlerstellen und baue die beschriebenen Änderungen im Quelltext ein.
3. Speichere die Änderungen. Dabei müssen Quelltextdateien formatiert werden.

Informationen über die Implementierung der Code-Mutation findet man in Abschnitt 5.8.

Nachdem alle Änderungen gespeichert sind, wird die korrigierte Anwendung mit der Test-Umgebung auf Korrektheit überprüft.

4.14 Zusammenfassung

Dieses Kapitel hat ein Verfahren zur Korrektur von Wettlaufsituationen durch LOCK-Blöcke und Austausch von Datenstrukturen vorgestellt.

Detailliert wurden das Code Object Model, eine aufwendige Kontroll- und Datenflussanalyse, das Stopper-Konzept zum Erkennen von Atomizitätstellen sowie Korrekturmuster zum Korrigieren von zusammengesetzten Anweisungen vorgestellt. Eine Regionenanpassung und eine Sperrenordnung sorgen für das Erzeugen korrekter und verklemmungsfreier Korrekturvorschläge. Somit wurden alle Anforderungen erfüllt.

Im nächsten Kapitel werden einige interessante Aspekte der Implementierung vorgestellt. Danach wird das Konzept anhand der Implementierung mit Beispielanwendungen evaluiert.

5. Implementierung

Dieses Kapitel erläutert die Implementierung des in Kapitel 4 vorgestellten Entwurfs zum automatischen Vorbereiten von Lösungen für Wettlaufsituationen sowie den Einsatz externer Werkzeuge und Technologien (siehe Abschnitt 2.2). Das in C# implementierte Werkzeug analysiert .NET Anwendungen auf Quelltext- und IL-Ebene und generiert Korrekturen für die Datenwettläufe, die im C#-Quelltext eingebaut werden.

Struktur des Kapitels

Im ersten Abschnitt wird ein Überblick über die Architektur und den Ablauf des Werkzeugs geschaffen. Danach wird in Abschnitt 5.2 eine Übersicht der implementierten und nicht implementierten Features gezeigt. In Abschnitt 5.3 wird die grafische Oberfläche vorgestellt. Abschnitt 5.4 beschäftigt sich mit der Ausführung der Komponententests und der Zusammenarbeit zwischen Programmcode und der eingesetzten Testumgebung CHESS (siehe Abschnitt 2.2.5). Anschließend wird in Abschnitt 5.5 der Aufbau des Code Object Models erläutert. Zum Schluss werden in Abschnitte 5.6, 5.7 und 5.8 einige Aspekte der Implementierung der simulierten Codeausführung, der Generierung und des Speicherns von Quelltextkorrekturen vorgestellt.

5.1 Architektur und Ablauf im Überblick

Dieser Abschnitt gibt einen Überblick über die Architektur des Werkzeugs. Es werden die Hauptkomponenten der Implementierung anhand Abbildung 5.1 sowie die Ausführungsreihenfolge anhand eines Sequenzdiagramms (siehe Abbildung 5.2) vorgestellt.

5.1.1 Die Komponenten

Der implementierte Code ist nach Funktionalität in verschiedenen Komponenten organisiert. Abbildung 5.1 zeigt eine vereinfachte grafische Darstellung der Beziehungen zwischen den einzelnen Codeteilen. Aus Übersichtsgründen wurden einige Beziehungen weggelassen, diese werden aber in der nachfolgenden Komponentenbeschreibung nachgeholt. Die Komponenten werden in 6 Gruppen aufgeteilt:

User Interface

Die `User Interface`-Gruppe enthält zwei Komponenten, die die Kommunikation zwischen Benutzer und Werkzeug übernehmen:

Frontend: Diese Komponente enthält die gesamte Funktionalität, die dem Benutzer von der grafischen Oberfläche zur Verfügung gestellt wird:

- Konfiguration und Eingabe der Parameter.
- Erzeugen und Starten des `DataRaceSolvers`.
- Anzeigen von Ausgabeinformationen durch das Output System.

Mithilfe der `CIL`-Komponente kann das `Frontend` auch den Inhalt von .NET Assemblies anzeigen. Die `Frontend`-Komponente ist für die anderen Komponenten nur über das `Output System` aufrufbar.

Output System: Das `Output System` ist die Kommunikationsschnittstelle des Werkzeugs, mit deren Hilfe dem Benutzer Ausgabe-und Fehlerinformationen mitgeteilt werden. Die Ausgabe erfolgt über grafische Elemente, die am Anfang bei der Initialisierung der Komponente vom `Frontend` übergeben werden. Das `Output System` verfügt über statische Methoden, die Text in verschiedenen Ansichten, sowie grafische Meldungen, anzeigen. Die Komponente ist aus dem gesamten Code direkt aufrufbar.

External Tools

Die zwei Komponenten dieser Gruppe kümmern sich um die Integration von den notwendigen externen Werkzeugen:

Build Environemt: Diese Komponente ist für das Kompilieren von .NET-Projekten zuständig. Sie verwendet die Funktionalität, die Visual Studio über `devenv.com` [Deva] bereitstellt.

Test Environemt: Das `Test Environment` erfüllt die folgenden Aufgaben:

- Ausführen der vom `DataRaceSolver` übergebenen Komponententests mit Hilfe des CHESS-Werkzeugs (siehe Abschnitt 2.2.5)
- Evaluieren der Ergebnisse mit dem `Concurrency Explorers` und Entfernen unnötiger Ergebnisse
- Rückgabe der relevanten Ergebnisse (TestResults)

Das Zusammenspiel dieser Komponente mit CHESS wird in Abschnitt 5.4 detailliert erläutert.

Native Code

Die `Native Code`-Gruppe ist die Schnittstelle zum Code des übersetzten Programms. Diese besteht aus einer Komponente:

CIL: Die `CIL`-Komponente ermöglicht das Laden von .NET Assemblies. Wenn aufgerufen, liefert diese ein `LoadedModule`-Objekt zurück, das den Inhalt der Assembly sowie den der PDB-Datei enthält. Diese Komponente erfüllt alle ihren Aufgaben mithilfe von CCI (siehe Abschnitt 2.2.3).

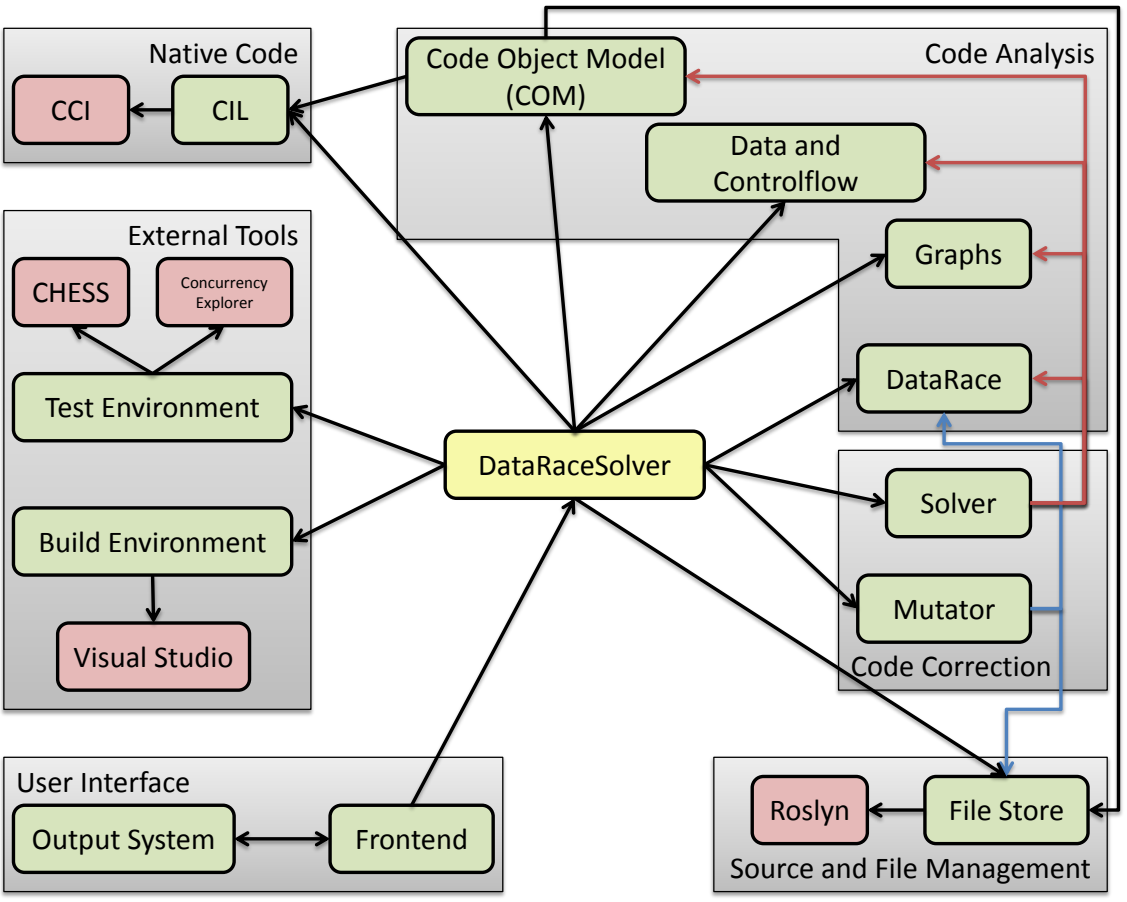

Abbildung 5.1: Vereinfachte Abbildung der Architektur des implementierten Werkzeugs. Grün und Gelb - interne Komponenten, Rot - externe Bibliotheken und Werkzeugen.

Code Analysis

Die Gruppe `Code Analyse` enthält vier Komponenten, die das Code Object Model aufbauen, es analysieren und alle für die Korrektur notwendigen Informationen vorbereiten:

Code Object Model (COM): Die Komponente `Code Object Model` nutzt die Daten, die die Komponenten `CIL` und `File Store` zur Verfügung stellen, um das in Abschnitt 4.3 vorgestellte Modell aufzubauen. Im Zentrum des Code Object Models liegen die Statement-Objekte, die die Methoden aufbauen. Die Komponente enthält Erweiterungsmethoden, die die Lücke zwischen `COM` und `CIL` überbrücken. Diese Methoden ermöglichen zum Beispiel den direkten Zugriff auf einer COM-Anweisung über ihre CIL-Instruktionen.

Der genaue Aufbau wird in Abschnitt 5.5 erläutert.

Data- and Controlflow: Auf Basis des Code Object Models führt diese Komponente den Code simuliert aus und sammelt Daten- und Kontrollflussinformationen. Die Implementierung folgt die Beschreibung von Abschnitt 4.9.

Graphs: Diese Komponente ist für den Aufbau von Aufruf- (siehe Abschnitt 4.6) und Sperrengraphen (siehe Abschnitt 4.7) auf Basis des Code Object Models zuständig.

DataRace: Die Aufgabe der `DataRace`-Komponente ist, die erkannten Fehlerstellen im Code Object Model zu finden und in `RacingLocation`-Objekten zu beschreiben. Danach werden diese Objekte in Datenwettlaufgruppen eingeteilt.

Code Correction

Die zwei Komponenten der `Code Correction`-Gruppe sind für das Erzeugen und Speichern der Code-Änderungen zuständig:

Solver: Die `Solver`-Komponente (siehe Abschnitt 4.10) analysiert alle Fehlerstellen und bildet für jede Stelle eine Liste von `Statement`-Objekten, die geschützt werden müssen, um den Datenwettlauf zu beheben. Zusätzlich werden die notwendigen neuen Sperren bestimmt und zurückgegeben, sodass diese vom `Mutator` auch im Quelltext eingebaut werden. Wie Abbildung 5.1 zeigt, muss diese Komponente für ihre Arbeit auf alle `Code-Analysis`-Komponenten zurückgreifen.

Mutator: Der `Mutator` (siehe Abschnitt 4.13) baut die vom `Solver` vorbereiteten Codeänderungen im Quelltext ein. Die Änderungen sind Teil des `DataRaceGroup`-Objektes. Der Mutator verwendet die Dienste, die die `File Store`-Komponente zur Verfügung stellt.

Source and File Management

Diese Gruppe ist die Schnittstelle zum Dateisystem und ermöglicht das Öffnen, Speichern, Editieren und Wiederherstellen von Dateien. Diese enthält nur eine Komponente:

File Store: Die `File Store`-Komponente ermöglicht das Laden und Speichern von Dateien sowie ihr Editieren. Diese bekommt den Pfad zu einer Datei und liefert ein initialisiertes VirtualFile-Objekt zurück. Geladene Dateien werden auch in einem Cache gehalten. definierten Eigenschaften. Für Quelltextdateien werden die `Roslyn` APIs (siehe Abschnitt 2.2.4) verwendet, um zusätzlich zu der Textdarstellung einen Syntaxbaum bereitzustellen und den Quelltext vor dem Speichern zu formatieren.

5.1.2 Ablauf und die DataRaceSolver-Komponente

Im letzten Abschnitt wurden die einzelnen Komponenten mit ihrer Funktionalität und Abhängigkeiten vorgestellt. In diesem wird das Zusammenspiel zwischen den Komponenten sowie der Ablauf des Werkzeugs erläutert.

Das Sequenzdiagramm auf Abbildung 5.2 zeigt den Ablauf des Werkzeugs. Einige Beziehungen sind nur durch den Datenaustausch zwischen den Komponenten abgedeckt. Einige Komponenten, wie zum Beispiel das Output System, wurden aus dem Diagramm entfernt, um dieses übersichtlich zu halten. Diese Komponenten werden auch nicht zum Verständnis des internen Aufbaus des Werkzeugs benötigt.

Die wichtigsten Objekte, die während des gesamten Ablaufs manipuliert und zwischen den Komponenten ausgetauscht werden, sind auf dem Sequenzdiagramm (siehe Abbildung 5.2) als Rückgabewerte und Aufrufparameter dargestellt.

Wie auf Abbildung 5.2 zu erkennen ist, hat die Implementierung eine zentrale Steuerungseinheit:

DataRaceSolver: Die `DataRaceSolver`-Komponente ist der Kern des Werkzeugs. Diese erzeugt alle anderen Komponenten, steuert sie und kümmert sich um den notwendigen Datenfluss. Eine Instanz der Klasse wird von der `Frontend`-Komponente erzeugt und führt danach ihre Aktivitäten unabhängig davon. `DataRaceSolver` übernimmt die Rolle des im klassischen MVC-Architekturmuster vorgestellten Controllers [Gam11]. Er implementiert keine eigene analytische Funktionalität und basiert seine Steuerungslogik auf den Ergebnissen, die anderen Komponenten liefern. Eine Code-Analyse und -Korrektur kann von einigen Sekunden bis mehreren Minuten dauern. Diese Zeit wird vor allem mit Ausführen der Komponententests verbracht. Aus diesem Grund wird die Komponente in einem eigenen Thread ausgeführt, sodass die Benutzerschnittstelle während der Ausführung nicht blockiert bleibt.

Im nächsten Abschnitt wird eine Übersicht über die implementierten und nicht implementierten Features des Entwurfs gemacht.

5.2 Implementierte Features

Die Implementierung des Werkzeugs folgt strikt dem in Kapitel 4 vorgestellten Entwurf. In der Tabelle in Abbildung 5.3 sind alle implementierten Features aufgezählt mit einer Referenz auf den entsprechenden Abschnitt in Kapitel 4 und Bemerkungen, die einige Eigenschaften der Implementierung beschreiben.

Die Korrektur von Datenwettläufen im externen Code durch Austausch von Datenstrukturen (erläutert in Abschnitt 4.10.5) wurde aus Zeitgründen nicht implementiert. Der Aufbau des dafür notwendigen Datenflussgraphen (siehe Abschnitt 4.8) ist im Code vorhanden, seine Traversierung, die für das Verfolgen von Objekten zu ihren Erstellungsquellen aber nicht. Diese wird in der Arbeit nur für das Erkennen der Datenstruktur verwendet. Die Korrektur von Datenwettläufen im externen Code wird in dieser Implementierung durch Lock-Blöcke im Programmcode korrigiert (siehe Abschnitt 4.10.4), was den allgemeinen Fall abdeckt. Der Austausch von Datenstrukturen kann in einer weiterführenden Arbeit implementiert und evaluiert werden.

In den folgenden Abschnitten werden interessante Teile der Implementierung und das Werkzeug detaillierter vorgestellt.

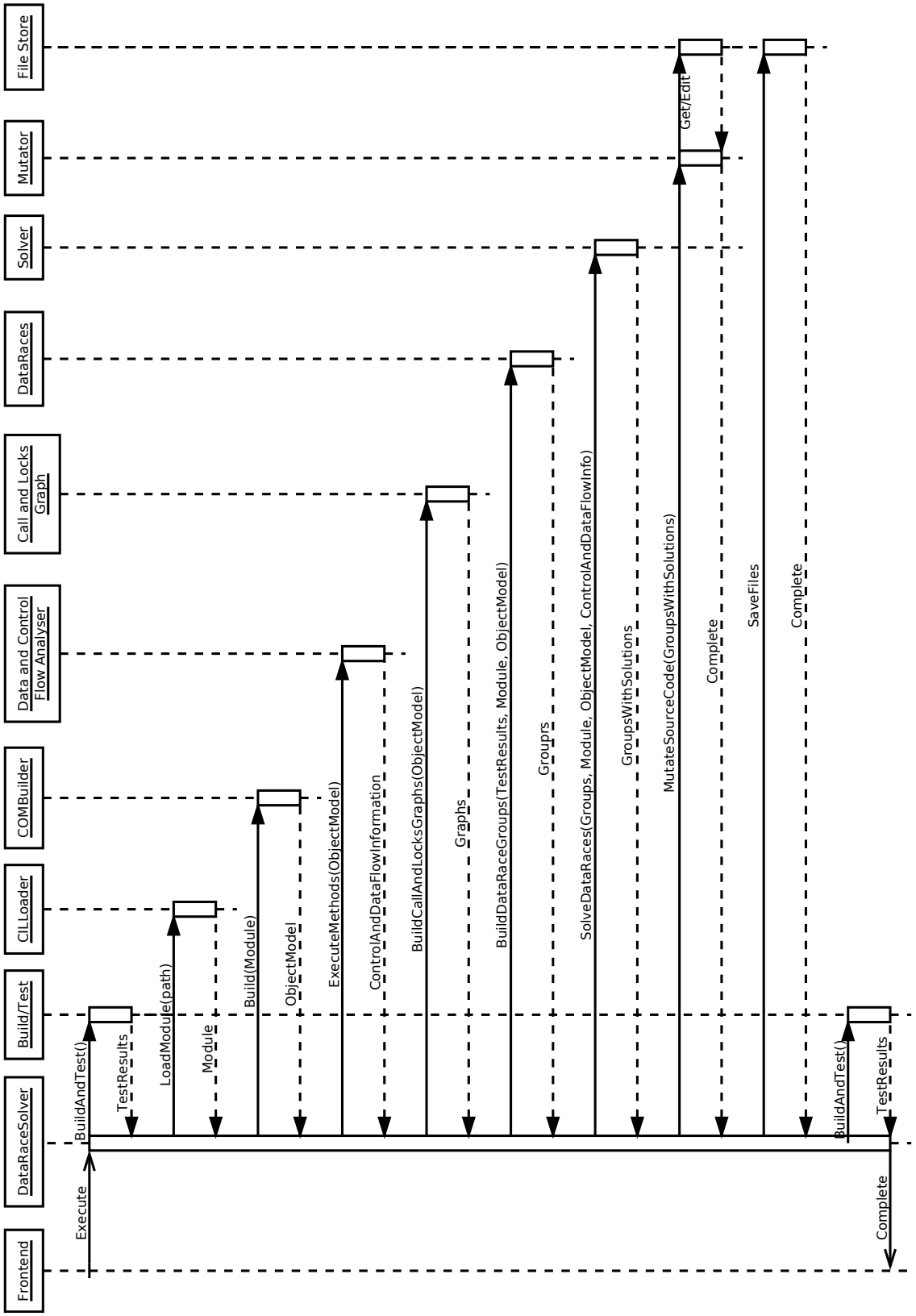

Abbildung 5.2: Vereinfachtes Sequenzdiagramm des Ablaufs des Werkzeugs.

Feature	Abschnitt	Zuständige Komponente	Bemerkungen
.Net Projekt Kompilieren	-	Build Environmen	Verwendet Visual Studio
Ausführung der parallelen Tests	4.2	Test Environment	Verwendet CHESS
Laden von .NET Assemblies	4.3	CIL	Mittels CCI
Code Object Model	4.3	Code Object Model	
Analyse der Testergebnisse	4.4	DataRace	
Bestimmen der Sperren	4.5	Solver	Erkennen und Wiederverwenden bereits vorhandener Sperren mittels Annotationen unterstützt. Bei Notwendigkeit werden neue Sperren generieren.
Aufrufgraph	4.6	Graphs	
Sperrgraph und Erkennen von potenziellen Verklemmungsituationen	4.7	Graphs	
Kontroll- und Datenflussanalyse • Simulierte Ausführung • Ergebnisliste • Datenflussgraph (Aufbau)	4.8/4.9	Data and Controlflow	
Analyse der Datenwettläufe • Feldabhängigkeiten und Stopper-konzept • Datenwettlaufkorrektur durch Lock-Blöcke (im Programmcode und im externen Code)	4.10	Data and Controlflow Solver	Die Korrektur durch Austausch von Datenstrukturen wird in dieser Implementierung nicht unterstützt.
Regionnenanpassung	4.11	Solver	
Ordnung von Sperren	4.12	Solver	
Quellcodemutation	4.13	Mutator	Die Mutation unterstützt den Austausch von Datenstrukturen nicht.

Abbildung 5.3: Liste aller implementierten Features.

5.3 Die grafische Oberfläche

In diesem Abschnitt wird die grafische Oberfläche des Werkzeugs vorgestellt. Diese wurde mit `Windows Forms` [Micd, Micb] implementiert. Auf Abbildung 5.4 sieht man das Hauptfenster der Anwendung. Dieses bietet Konfigurations- und Ausgabemöglichkeiten anhand folgender Elemente:

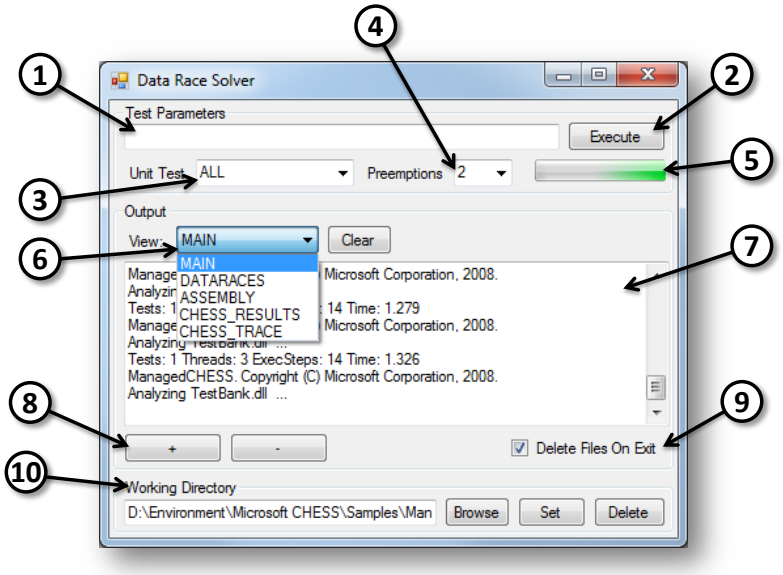

Abbildung 5.4: Hauptfenster des implementierten Werkzeugs.

1. Eingabefeld für die manuelle Testkonfiguration. Dieses Feld unterstützt auch Drag and Drop für CHESS Konfigurationsdateien.
2. Startknopf. Während des Testens und der Korrektur der Assembly sind alle Kontrollelemente deaktiviert.
3. Liste mit verfügbaren Komponententests. Die Tests werden von der CHESS-XLM-Datei ausgelesen. Der Benutzer hat die Möglichkeit einen bestimmten oder alle verfügbaren Komponententests zu verwenden.
4. Einstellung, die die maximale Anzahl von Thread-Wechslungen beim Testen steuert. Die Einstellung wird CHESS übergeben. Der Standardwert 2 bedeutet, dass nur Schedules die maximal 2 Thread-Wechslungen enthalten, untersucht werden.
5. Statusanzeige.
6. Liste mit Ansichten für die Textausgabe. Der Benutzer kann unter den Folgenden auswählen:

   MAIN: Textausgabe, die vom Werkzeug generiert wird, um Fortschrittsstatus zu melden.

   DATARACES: Reportansicht, die alle von CHESS erkannten Datenwettläufe anzeigt. Diese sind die Datenwettläufe, die korrigiert werden.

   ASSEMBLY: Zeigt den Inhalt der getesteten Assembly an. Diese Ansicht kann auch als reiner Assembly Viewer verwendet werden und unterstützt Drag and Drop von Assembly-Dateien.

   CHESS_RESULTS: Enthält den Inhalt aller generierten CHESS result-Dateien.

CHESS_TRACE: Enthält den Inhalt aller generierten CHESS trace-Dateien.

7. Textfeld für die Textausgabe.
8. Knöpfe, die die Schriftgröße der Textausgabe kontrollieren.
9. Einstellung, die das Löschen aller von CHESS erzeugten Dateien nach der Ausführung steuert.
10. Dieser Bereich erlaubt das Setzten und Anpassen des Arbeitsverzeichnisses. Beim Parsen einer CHESS-XML-Datei wird dieses Verzeichnis automatisch bestimmt.

Abbildung 5.5 stellt ein Fenster dar, das die alte und die neue Version des Quelltextes vor dem Speichern zum Vergleich anzeigt. Neu erstellte Dateien werden auch angezeigt.

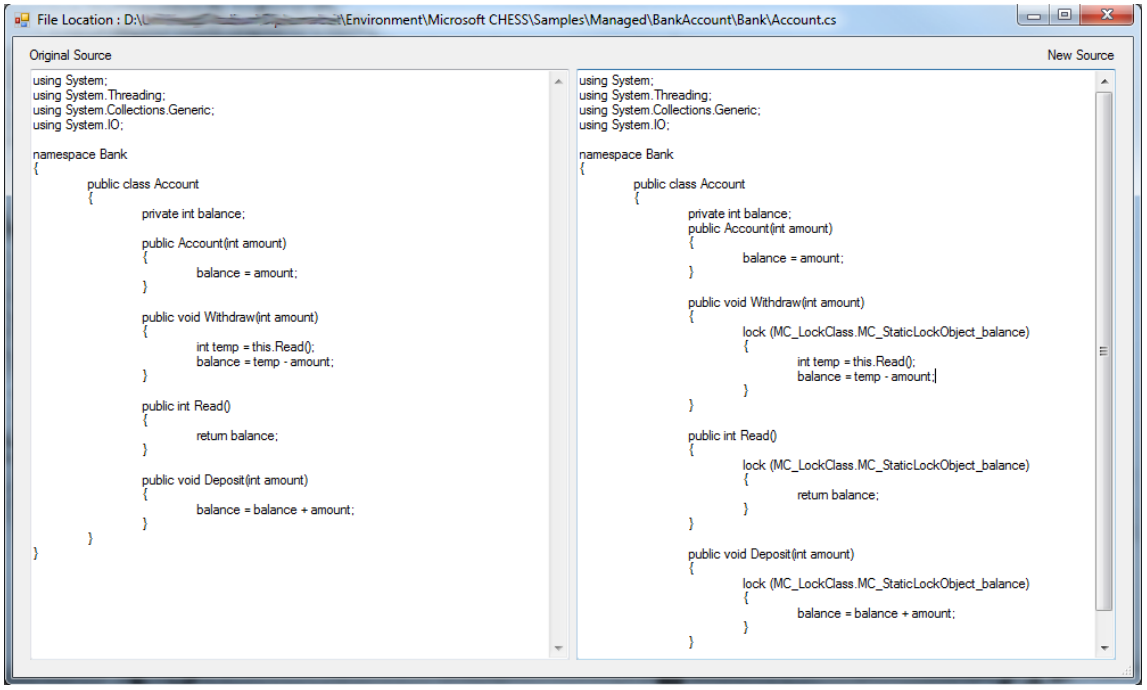

Abbildung 5.5: Vergleich zwischen alter und neuer Version einer Klasse.

5.4 Testen und Zusammenarbeit mit CHESS

In diesem Abschnitt wird erläutert, wie CHESS beim Ausführen der Komponententests vom Werkzeug gesteuert wird, um alle Datenwettläufe zu erkennen. Die Parameter, die an CHESS übergeben werden, kommen von der grafischen Oberfläche und werden entweder manuell oder durch Parsen einer XML-Konfigurationsdatei (siehe Abbildung 5.6) eingegeben. Die Konfigurationsdatei enthält alle notwendigen Informationen, die das Werkzeug benötigt, um ein C#-Projekt zu Übersetzten, mit CHESS zu testen und dann zu korrigieren.

Um alle Datenwettläufe zu erkennen, wird CHESS mehrmals als separater Prozess ausgeführt. Für jeden Komponententest werden die folgenden Schritte ausgeführt:

- Der Test wird mit CHESS ausgeführt. Bei dieser Ausführung wird eine Berichtdatei erstellt, die eine Beschreibung aller gefundenen Probleme enthält (siehe Abbildung 5.7).
- Wie auf Abbildung 5.7 zu sehen ist, enthält die Berichtdatei nur eine kurze Beschreibung sowie den Schedule, mit dem CHESS das Problem reproduzieren kann. Um

```
<?xml version="1.0" encoding="utf-8"?>
<testlist name="BankAccount" xmlns="http://research.microsoft.com/chess">
   <!-- This element describes how to build the Bank example -->
   <build>
      <startdir>@</startdir>
      <executable>msbuild</executable>
      <arg>Bank.sln</arg>
      <arg>/p:Configuration=Debug</arg>
   </build>
   <!-- We use just a single test in this example -->
   <test name="Test">
      <startdir>@\TestBank\bin\Debug</startdir>
      <executable>mchess</executable>
      <carg>TestBank.dll</carg>
      <carg>/ia:Bank</carg>
      <carg>/ia:mscorlib</carg>
   </test>
</testlist>
```

Abbildung 5.6: Diese XML-Konfigurationsdatei enthält Übersetzungsparameter und die-Beschreibung eines Komponententests. Aus dieser Datei werden das Arbeitsverzeichnis, die Wege zur Test- und zur getesteten Assembly ausgelesen. Ihr XML-Layout wurde für CHESS Board definiert und wird in dieser Arbeit wiederverwendet.

einen Datenwettlauf zu korrigieren, werden die Positionen der beiden Instruktionen benötigen, die ihn verursacht haben. Um diese für alle erkannten Datenwettläufe zu bekommen, wird CHESS noch einmal für jeden Datenwettlauf mit dem entsprechenden Schedule in Trace-Modul gestartet. Jede Ausführung liefert einen Trace-Bericht.

- Mit Concurrency Explorer werden aus jedem Trace-Bericht die beiden Stack Traces bestimmt, die das Problem verursacht haben. Die Anpassung, die in CHESS und in Concurrency Explorer gemacht wurde, erlaubt das Auslesen zusätzlicher Informationen wie Methodenparameter und Instruktionsoffset 2.2.5.

```
<result>
   <label>R1</label>
   <description>Found data race at Bank->D:\BankAccount\TestBank\bin\Debug\Bank.dll->
      Bank.Account->System.Int32 Bank.Account.Read()->16:Account.cs(49)
   </description>
   <action name="Repro" race="1" />
   <schedule format="hex„>01000000 01000000 09000000 01000000 FF010000 01000000
      01000000 FF010000 12000000 01000000 02000000 05000000 01000000 02000000
      09000000 01000000 03000000 05000000 01000000 03000000 09000000 02000000
      02000000 06000000 02000000 58DB0000 02000000 02000000 58DB0000 02000000
      03000000 01000000 00000000 01000000 FF010000 12000000 01000000 01000000
      01000000 00000000 00000000 01000000 00000000 06000000 02000000 08000000
   </schedule>
</result>
```

Abbildung 5.7: Ausschnitt einer CHESS-Berichtdatei, die einen erkannten Datenwettlauf beschreibt. Aus diesem Bericht ist aber nur eine der beteiligten Stellen identifizierbar.

Alle gefundenen Datenwettläufe werden in einer Liste zurückgegeben.

5.5 Aufbau des Code Object Models

In diesem Abschnitt wird der Aufbau des Code Object Models und vor allem der in Abschnitt 4.3 definierten Anweisungen erläutert. Abbildung 5.8 stellt die implementierte

Klassenhierarchie der Anweisungen dar. Das `Statement`-Objekt stellt eine einfache Anweisung dar, zum Beispiel eine Variablenzuweisung oder einen Methodenaufruf. Auf der Basis dieses Objektes sind die zusammengesetzten Anweisungen aufgebaut. Sie erben alle von der abstrakten `ComplexStatement`-Klasse. In den nachfolgenden Abschnitten werden alle von der Implementierung unterstützten zusammengesetzten Anweisungen vorgestellt:

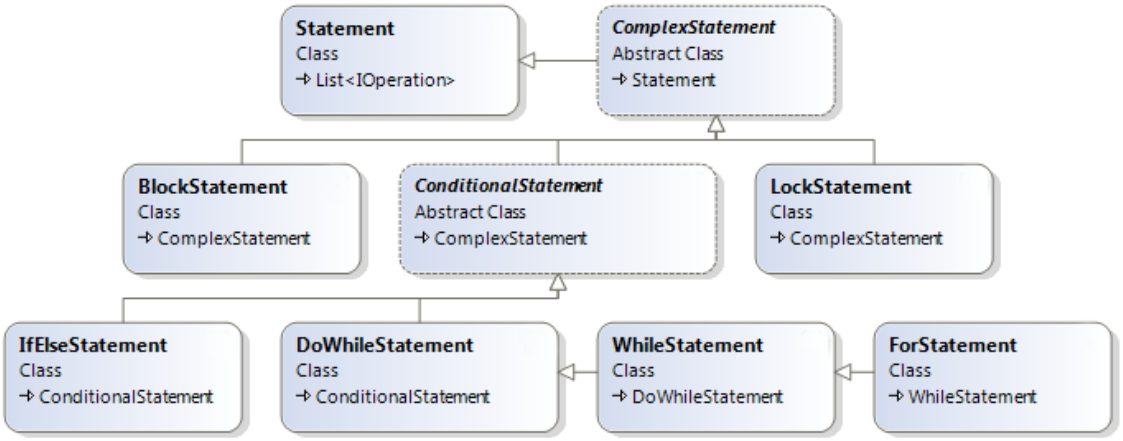

Abbildung 5.8: Implementierte Hierarchie aller Anweisungen.

BLOCK - Anweisung

Die Block-Anweisung beschreibt eine Menge von einfachen und zusammengesetzten Statements, die im Quelltext zusammengehören und zum Beispiel durch geschweifte Klammern vom Rest isoliert sind. Dieser Statementtyp hat vor allem eine Rolle im Aufbau der anderen Statement-Objekte. Kann beim Aufbauen nicht entschieden werden, was der Typ einer zusammengesetzten Anweisung sein sollte, so wird Block genommen. Eine spätere Typanpassung ist in diesem Fall notwendig.

IF - Anweisung

Die IF-Anweisung ist die einfachste Form einer Bedingunganweisung. Im Code Object Model enthält eine IF-Anweisung die folgenden Bestandteile:

- Sprungbedingung mit einem Sprungbefehl (falls ein Sprung notwendig ist).
- IF-Körper, der ausgeführt wird, wenn die Bedingung gültig ist.

IF/ELSE - Anweisung

Die IF/ELSE-Anweisung ist eine Erweiterung der IF-Anweisung und enthält die folgenden Teile:

- Sprungbedingung mit einem Sprungbefehl (falls ein Sprung notwendig ist).
- IF-Körper, der ausgeführt wird, wenn die Bedingung gültig ist.
- ELSE-Sprungbefehl: Wird ausgeführt, um den Körper zu überspringen, im Fall, dass der IF-Körper ausgeführt wird.
- ELSE-Körper, der ausgeführt wird, wenn die Sprungbedingung verletzt ist.

Die IF- und die IF/ELSE-Anweisung sind beide mit der IfElseStatement-Klasse implementiert. Liegt eine IF-Anweisung vor, dann sind Else-Felder nicht belegt.

DO/WHILE - Schleife

Im Code Object Model haben alle Schleifen einen ähnlichen Aufbau, der auf der Darstellung der DO/WHILE-Schleife basiert:

- Schleifenkörper, der vor der Überprüfung der Schleifenbedingung einmal ausgeführt wird.
- Schleifenbedingung, die äquivalent zu der Sprungbedingung der IF-Anweisung ist. Wenn die Bedingung erfüllt ist, wird die Ausführung am ersten Befehl des Schleifenkörpers fortgesetzt. Dieser Sprung bildet auch die Schleife.

WHILE - Schleife

Die WHILE-Schleife ist eine leicht erweiterte Form der DO/WHILE-Schleife. Dabei wird zuerst die Bedingung überprüft und erst dann der Schleifenkörper ausgeführt. Aus diesem Grund enthält diese Schleife die folgenden Elemente:

- Sprung zu der Schleifenbedingung. Dieser Befehl überspringt die erste Ausführung des Schleifenkörpers.
- Schleifenkörper, der ausgeführt wird, wenn die Schleifenbedingung erfüllt ist.
- Schleifenbedingung, die äquivalent zu der Schleifenbedingung der DO/WHILE-Schleife aufgebaut ist.

FOR - Schleife

Als eine Erweiterung der WHILE-Schleife, erbt diese Anweisung ihren Aufbau von der WHILE-Schleife und fügt zusätzliche Elemente und Funktionalität hinzu:

- FOR-Initialisierung. Dieses Statement wird nur einmal ausgeführt, um einen Wert der Zählvariable zu initialisieren.
- Sprung zu der Schleifenbedingung. Dieser Befehl überspringt die erste Ausführung des Schleifenkörpers.
- Schleifenkörper, der ausgeführt wird, wenn die Schleifenbedingung wahr ist.
- FOR-Update, aktualisiert die Zählvariable der Schleife, um diese für die nächste Iteration vorzubereiten.
- Schleifenbedingung, die äquivalent zu der Schleifenbedingung der DO/WHILE-Schleife aufgebaut ist und sich verhält.

Eine Unterscheidung zwischen einer WHILE- und einer FOR-Schleife ist anhand der Informationen, die die CIL-Instruktionen und die PDB-Datei enthalten, nicht möglich. Auf CIL-Ebene können zwei Code-Bereiche, die jeweils einer der beiden Schleifen enthalten, identisch aussehen. An dieser Stelle wird auf den Quelltext zugegriffen, um den genauen Schleifentyp zu erkennen.

LOCK - Block

Der Lock-Block ist ein Synchronisationskonstrukt. Er stellt eine Möglichkeit dar, sicher Sperren zu akquirieren und automatisch auch in dem Fall eines Fehlers freizugeben. Zahlreiche Informationen über die verwendeten Sperren, wie zum Beispiel deren Reihenfolge, lassen sich von dieser Anweisung extrahieren. Im Code Object Model sieht ein Lock-Block folgendermaßen aus:

- LockEnter-Statement, das auch den Aufruf der Akquirierungsmethode enthält.
- Körper, der eine Liste von geschützten Anweisungen enthält.
- LockExit-Statement. Dieses Statement enthält die Methode, die die Sperre freigibt. Eine Fehlerbehandlung, für den Fall, dass eine Exception im Körper geworfen wird, ist auch enthalten.

Aufbauschritte

Der Aufbau der Code Object Model-Anweisungen (siehe Abbildung 5.8) besteht aus insgesamt 3 Schritten. Abbildung 5.9 zeigt den Informationsfluss und die Bestandteile des implementierten Code Object Models.

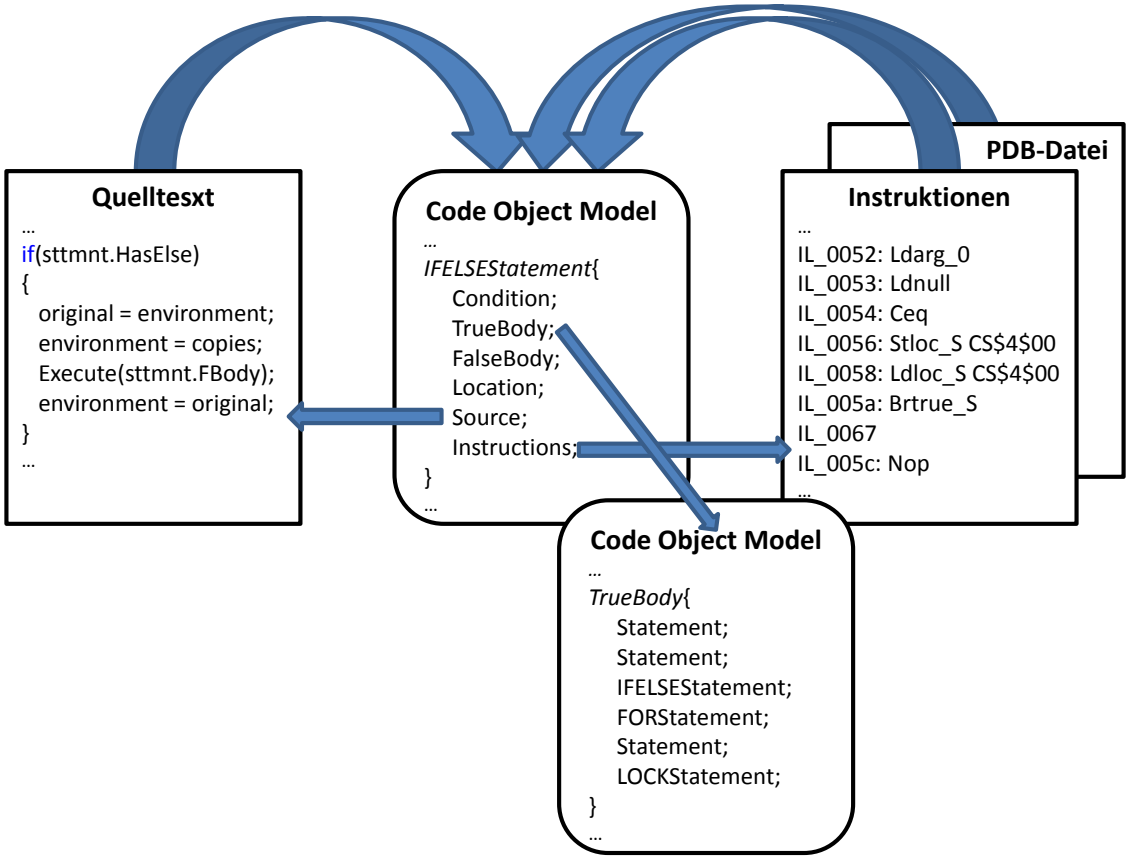

Abbildung 5.9: Code Object Model: Informationsfluss und Bestandteile.

1. Aufbau der `Statement`-Objekte: Aus der pdb-Datei, kann die Quelltextposition der ersten IL-Instruktion einer Anweisung ausgelesen werden. Diese Information wird in Visual Studio beim Debuggen verwendet und mithilfe dieser Information werden auch die `Statement`-Objekte aufgebaut. Eine IL-Instruktion mit gültiger Position signalisiert das Ende des gerade aufgebauten `Statement`-Objekts und der Anfang eines Neuen. Hat die IL-Instruktion keine Quelltextposition, so gehört sie zum bereits angefangenen `Statement`-Objekt. Die versteckten Instruktionen sind eine Ausnahme. Das sind Instruktionen, die vom Compiler erzeugt werden und vom Debugger übersprungen werden [Bas, Sta]. Diese werden anhand ihrer Zeilennummer (16707566) erkannt. Beispiele für solche Instruktionen sind branch-Instruktionen nach Sprungbedingungen oder am Anfang von while-Schleifen. Mehrere solcher Instruktionen werden vom Compiler gruppiert, sodass nur die Erste die spezielle Zeilennummer hat. Um die `Statement`-Objekte korrekt aufzubauen, wird zum Beispiel eine versteckte `branch`-Instruktion als letzte Instruktion eines `Statement`-Objektes interpretiert.

2. Gruppieren zusammengehörender Statement-Objekte: Um den Aufbau der `ComplexStatement`-Objekte zu erleichtern, wurde dieser Zwischenschritt definiert. Logisch zusammengehöre `Statement`-Objekte werden verschmolzen. Beispiel ist eine IF-Bedingung und der danach folgende Sprung. Kann die Bedingung zur Übersetzungszeit evaluiert werden, so fehlt der Sprung. Das Verschmelzen der zwei `Statement`-Objekte erlaubt dem dritten Schritt, die zwei Fälle gleich zu bearbeiten.

3. Aufbau der `ComplexStatement`-Objekte: In diesem Schritt werden alle vorgestellten Anweisungen aufgebaut. Hierfür werden die `Statement`-Objekte eines nach dem anderen analysiert. Die Analyse versucht anhand der Instruktionen, die das Objekt aufbauen und des Quelltextes, den Typ der Anweisung zu bestimmen: `If`, `IfElse`, `While`, `DoWhile`, `JumpToWhile`, `JumpToFor`, `BlockStart`, `BlockEnd`, `Hidden`, `LockEnter`, `LockExit`, `Sourceless`, `Return`, `CILReturn`, `Unknown`. Jeder Typ hat eine eigene Bedeutung, die für verschiedene zusammengesetzte Anweisungen variieren kann. `IfElse` startet zum Beispiel den Aufbau eines neuen `IfElseStatement`-Objekts. Die verschiedenen `ComplextStatement`-Unterklassen implementieren Aufbauregeln, die bestimmen, in welcher Reihenfolge die `Statement`-Objekte akzeptiert werden und wann die Anweisung aufgebaut ist. Geschachtelte Anweisungen werden mithilfe eines Kellers aufgebaut. Wird eine neue zusammengesetzte Anweisung angefangen, so wird die Alte unfertige gespeichert. Ist die Neue fertig, so wird diese im Körper der Alten hinzugefügt. Sind alle `Statement`-Objekte in allen Methoden abgearbeitet, dann ist der Code Object aufgebaut.

Der Aufbau ist sowohl vom Quelltext als auch vom IL-Code abhängig und basiert auf Kenntnisse, die durch Analyse von übersetztem Quelltext gesammelt sind. Diese Implementierung arbeitet mit .NET 2.0/3.0 IL-Code, der vom Visual Studio-Compiler generiert ist.

5.6 Kontroll- und Datenflussanalyse mit simulierter Code-Ausführung

In diesem Abschnitt wird die Implementierung der Kontoll- und Datenflussanalyse sowie der simulierten Codeausführung erläutert.

Die Kontrollflussanalyse wird separat für jeden Methodenkörper ausgeführt. Zuerst wird eine Ausführungsumgebung erstellt, die alle in einem Moment vorhandenen Ausführungspfade als separate Kontexte enthält.

Jeder Kontext enthält die folgenden Informationen:

1. Liste mit Lese/Schreibe und Methodenaufruf-Ereignisse: In dieser Liste wird die genaue Position und Reihenfolge gespeichert.

2. Datenflussgraph: die in Abschnitt 4.8.1 vorgestellte Datenstruktur. Er bildet den Datenfluss im betrachteten Kontext ab.

3. Lokaler Ausführungsspeicher: Kann in Abhängigkeit von der analysierten Technologie eine Registerliste, Ausführungskeller oder eine andere Datenstruktur sein. Er erfüllt genau die Aufgaben der oben erwähnten Strukturen und spielt eine Rolle beim Aufbau der Ereignisliste und vor allem des Datenflussgraphen. Die Datenstruktur muss immer aktuell gehalten werden. Aus diesem Grund werden alle Instruktionen des Methodenkörpers in den entsprechenden Ausführungspfaden simuliert ausgeführt.

Die Ereignisliste und der Datenflussgraph werden simultan aufgebaut, weil die beiden eine simulierte Ausführung vom Methodenkörper benötigen.

Simulierte Ausführung des Methodenkörpers

Für diese Arbeit wurde eine Ausführung für CIL implementiert. Es werden alle Kelleroperationen mit Dummy-Daten ausgeführt, um Datenabhängigkeiten zu bestimmen und das Verfolgen von Objekten zu ermöglichen. Für die Ausführung der Instruktionen wurde dem ECMA-335 Standard gefolgt [Int10]. Der .NET-Ausführungskeller wird mit einem `Stack`-Objekt ersetzt und Variablen werden auf Datencontainer abgebildet. Beim Laden und Speichern von Daten entstehen Beziehungen zwischen den Datencontainern die danach verfolgt werden können, um zum Beispiel den Ursprung eines Objektes zu bestimmen.

Die simulierte Ausführung wird folgendermaßen durchgeführt:

1. Für den Methodenkörper wird eine Ausführungsumgebung mit einem Kontext angelegt.
2. Mithilfe des Code Object Models wird der ganze Körper iterativ abgearbeitet, in dem die Ausführung auf oberste Ebene anfängt, bis die gefundene Anweisung keine geschachtelten zusammengesetzten Anweisungen mehr enthält. Dabei werden die verschiedenen Anweisungstypen folgendermaßen ausgeführt:
   a) IF/ELSE - Anweisung: Zuerst wird die Bedingung ausgeführt. Danach wird die Ausführungsumgebung geklont. Mit den beiden Klonen werden den IF-Teil und den ELSE-Teil (falls vorhanden) ausgeführt. Danach werden die zwei Umgebungen verschmolzen. Abbildungen 5.10 (a) und 5.10 (b) zeigen diese Ausführung.
   b) DO/WHILE - Schleife: Der Körper und die Bedingung werden ausgeführt. Danach wird die Ausführungsumgebung geklont und der Körper und die Bedingung werden mit der originalen Umgebung noch einmal ausgeführt. Danach werden die zwei Teile verschmolzen und es entsteht nur eine Ausführungsumgebung. Die Ausführung ist auf Abbildung 5.10 (c) grafisch dargestellt.
   c) WHILE - Schleife: Die Bedingung wird ausgeführt. Danach wird die Ausführungsumgebung geklont und der Körper und dann noch einmal die Bedingung werden mit der originalen Umgebung ausgeführt. Die zwei Umgebungen werden am Ende verschmolzen.
   d) FOR - Schleife: Die Initialisierung und die Bedingung werden ausgeführt. Nach dem Klonen der Umgebungen wird mit der originalen Umgebung der Körper, die Aktualisierungsanweisung und am Ende die Bedingung ausgeführt. Die zwei Umgebungen werden am Ende verschmolzen.
   e) LOCK und andere Blöcke: Diese werden ohne Klonen der Umgebung ausgeführt.
   f) Einfache Anweisungen: Wenn eine Anweisung erreicht wird und diese keine geschachtelten Anweisungen enthält, dann wird diese auf Instruktionsebene für die aktuelle Umgebung ausgeführt.
3. Ausführung auf nativer Ebene: Auf nativer Ebene ist der Code eindeutig. Es gibt zum Beispiel eine Möglichkeit einen Wert aus einem Feld zu laden oder in ein Feld zu speichern. Auch bei Parameterübergabe werden die Felder zuerst geladen und dann übergeben. Ein Leseereignis wird nicht verpasst und muss auf keinen Fall gesondert betrachtet werden. Merkmale dieser Ausführung sind:
   a) Diese Ausführung kann keine neue Kontrollflüsse erzeugen. Bedingte Sprünge werden ignoriert, weil diese von dem Code Object Model in Betracht genommen wurden.

b) Bedingungen werden nur strikt ausgeführt und die enthaltenen bedingten Sprünge werden nicht ausgeführt, um die Zustandsexplosion der Kontrollflüsse zu minimieren. Eine Duplizierung der Umgebung und Erstellung neuer Kontexte bringt in diesem Fall auch nichts, weil Teile der Bedingung nicht separat geschützt werden können.

c) Nicht bedingte Sprünge werden genommen und der Kontext wird, bis die Zielinstruktion kommt, blockiert. Eine Ausnahme ist die Behandlung von Fehlern (Exceptions). Der Kontext verfügt über die notwendigen Informationen, um entsprechende finally-Blöcke auch nach einem blockierenden Sprung auszuführen. So werden potenzielle Abhängigkeiten nicht verpasst.

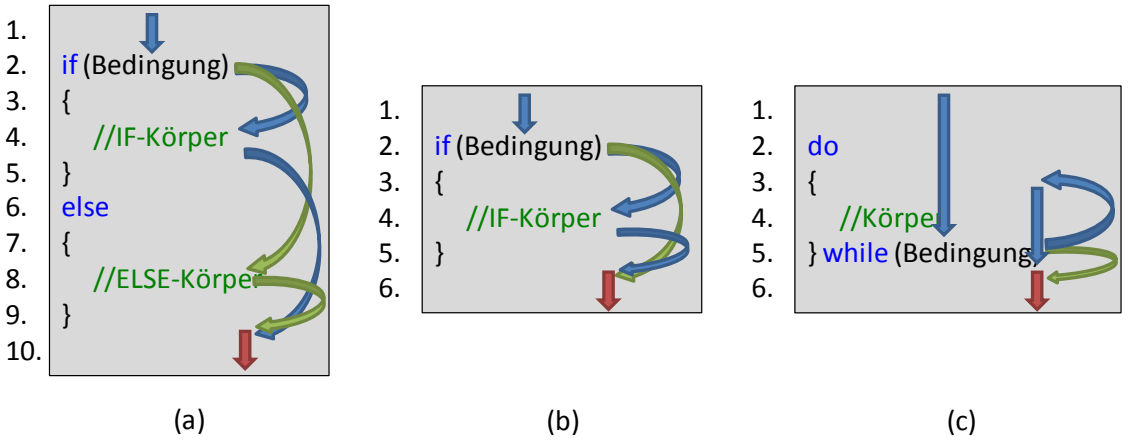

Abbildung 5.10: Simulierte Ausführung von IF, IF/ELSE und DO/WHILE - Anweisungen.

Die Ereignisliste und der Datenflussgraph werden bei der simulierten Ausführung nach dem folgenden Prinzip aufgebaut:

1. Wird ein Wert im lokalen Ausführungsspeicher geladen, erstelle ein neues Leseereignis, speichere es im entsprechenden Container des Datenflussgraphen und auch als einzelnes Element einer Liste im Ausführungsspeicher. Erstelle einen Leseeintrag in der Ereignisliste.
2. Wird eine Instruktion ausgeführt, dann werden die Spezifikationen der Sprache gefolgt und diese mit Ereignissen als Daten aus dem Ausführungsspeicher simuliert. Beispiel: Bei der Addition werden alle Ereignisse, die auf zwei gewünschten Positionen gespeichert sind, in einer gemeinsamen Liste hinzugefügt. Dann wird diese Liste als Ergebnis der Instruktion zurück in dem Ausführungsspeicher kopiert (siehe Abbildung 5.11). Alle Instruktionen, die den Ausführungsspeicher beeinflussen, müssen simuliert werden.
3. Wird ein Wert in ein Feld geschrieben, erstelle ein neues Schreibereignis. Speichere es im entsprechenden Container des Datenflussgraphen und verbinde es mit allen Leseereignissen, deren Wert in ihm gespeichert wird. Erstelle einen neuen Schreibeintrag in der Ereignisliste.
4. Wird eine Methode aufgerufen, dann sind die notwendigen Parameter bereits als Listen von Leseereignissen in dem Ausführungsspeicher geladen. Erstelle ein neues Schreibereignis für den Aufruf und verbinde die notwendige Anzahl von Leseereignissen mit dem Schreibereignis, sodass bei einer Analyse die Parameterübergabe korrekt verfolgt werden kann. Hat die Methode einen Rückgabewert, erstelle ein Leseereignis für ihn und speichere dieses in dem Ausführungsspeicher. Erstelle ein Aufrufereignis für die Ereignisliste.

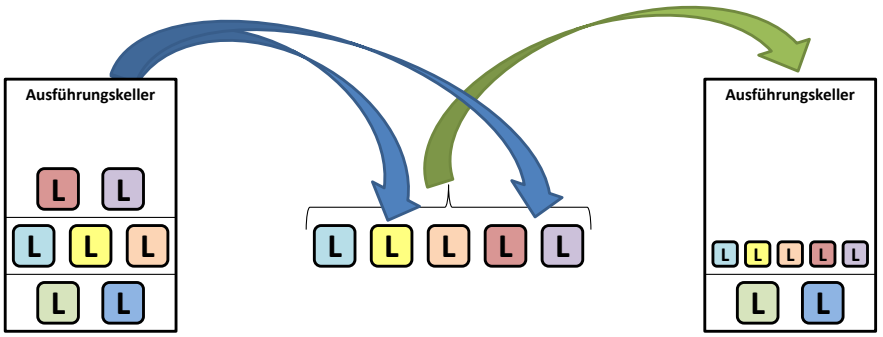

Abbildung 5.11: Simulierte Ausführung einer Addition auf Instrukionsebene.

5.7 Generieren der Korrekturen

Das Korrigieren der Datenwettläufe wurde detailliert in Kapitel 4 erläutert. Die in den Abschnitten 4.10, 4.11 und 4.12 vorgestellten Konzepte werden folgendermaßen implementiert:

1. Bestimme die Sperren, die die Datenwettlaufsgruppen schützen (siehe Abschnitt 4.5):
   a) Durchsuche die getestete Assembly und finde alle annotierten Sperren (siehe Abschnitt 4.5.1).
   b) Wird für eine Gruppe keine passende annotierte Sperre gefunden, erstelle eine neue (siehe Abschnitt 4.5.2).
2. Einsatz primärer Schutztechniken. Für jede Fehlerstelle führe Folgendes aus (siehe Abschnitte 4.10.1, 4.10.2):
   a) Erstelle eine Liste für alle Anweisungen, die geschützt werden müssen.
   b) Füge die betroffene Anweisung in der Liste hinzu.
   c) Bestimme Vorgänger- und Nachfolgerdatenabhängigkeiten für das betroffene Feld. Werden solche gefunden, füge sie in der Liste hinzu (siehe Abschnitt 4.10.3).
   d) Ist eine der Anweisungen aus der Liste eine FOR-Update-Anweisung, füge die FOR-Schleife in der Liste hinzu (siehe Abschnitt 4.10.2.2).
   e) Bilde aus den gesammelten Anweisungen einen Block. Liegt der Anfang oder das Ende einer zusammengesetzten Anweisung in der Liste, übernehme die ganze Anweisung. Anfang einer zusammengesetzten Anweisung kann unter anderem eine IF/ELSE-Bedingung, ein LOCKEnter oder eine FOR-Inizialisierung sein. Ende einer zusammengesetzten Anweisung kann unter anderem eine Schleifen-Bedingung sein (siehe Abschnitt 4.11.1).
3. Entferne bereits geschützte Fehlerstellen (siehe Abschnitt 4.11.2).
4. Einsatz sekundärer Schutztechniken. Für jede Fehlerstelle führe Folgendes aus:
   a) Ist die erste Anweisung des Blocks aus dem letzten Schritt eine IF/ELSE-Bedingung oder eine LOCKEnter-Anweisung, gehe zu b, sonst führe diese Schritte für die nächste Anweisung aus (siehe Abschnitte 4.10.2.1, 4.10.2.3).
   b) Wurden für die Fehlerstelle Datenabhängigkeiten erkannt? Wenn ja, führe die Schritte für die nächste Anweisung aus. Wenn nein, dann markiere die Stelle für eine Extrahierung.

5. Führe eine Regionenanpassung durch (siehe Abschnitt 4.11.3).
6. Erstelle die Sperrenordnung (siehe Abschnitt 4.12).

Nach jedem Schritt werden Duplikate und Regionen, die vollständig in einer Region derselben Datenwettlaufsgruppe liegen, entfernt.

5.8 Code-Mutation

Das Speichern der Code-Änderungen wurde in Abschnitt 4.13 vorgestellt. In dieser Implementierung wird für die Synchronisierung der kritischen Bereiche die C# `lock`-Anweisung verwendet. Zusätzlich wird die Projektdatei der korrigierten Assembly (.csproj) angepasst, sodass die Sperrendatei im Projekt hinzugefügt wird. Das ist nur dann notwendig, wenn neue Sperren angelegt werden und die Datei nicht vorhanden ist. Die angepassten Quelltextdateien werden vor dem Speichern mit Roslyn formatiert. Die Code-Mutation ist in der `Mutator`-Komponente implementiert und basiert auf der Funktionalität der `FileStore`-Komponente und vor allem der `VirtualFile`-Klasse.

Die virtuelle Datei

Die `VirtualFile`-Klasse ist Teil der `FileStore`-Komponente. Diese ist eine virtuelle Darstellung einer Datei, die nicht unbedingt eine Quelltext-Datei sein soll.

Die virtuelle Darstellung einer Datei ist ein String, das auch Zeilenumbrüche, Tabulatoren und Sonderzeichen enthält. Folgende Operationen werden unterstützt:

1. Textsuche mit Startposition und Richtung. Ergebnis ist der Startindex.
2. Hinzufügen vor und nach einem Index
3. Ersetzen und Anfügen
4. Übernahme von Änderungen ohne diese im Inhalt einzubauen.
5. Speichern aller übernommenen Änderungen auf einmal.

Das wichtigste Merkmal der virtuellen Datei ist die Flexibilität, die sie beim Ändern des Inhalts anbietet. Wenn eine Änderung eingetragen wird, dann wird diese nicht direkt auf den Inhalt angewandt. Dieser bleibt solange unverändert bis das gewünscht ist. Diese Funktionalität erlaubt es, mehrere Änderungen anhand des ursprünglichen Index einzubauen. Jede Textposition in der Datei verfügt über Listen von Änderungen, die vor und nach ihm einzubauen sind. Beim Speichern der Änderungen wird die Datei von oben nach unten abgearbeitet und für jede Position zuerst die Vor- und danach die Nachänderungen eingebaut. Nach einer Änderung verschiebt sich ein interner Positionszähler um die Länge des eingefügten Strings. Als Ergebnis wird die nächste Änderung an der Stelle gemacht, für die sie vorgesehen ist.

Nach dem Speichern werden die Dateien auf die Festplatte geschrieben. Quellcode-Dateien werden dabei neu formatiert, um sinnvolle Einrückungen und Zeilenumbrüche einzubauen.

Aufbau der Sperrendatei

Die Sperrendatei stellt einen Container für Sperren dar, in dem diese definiert und initialisiert werden. Der gesamte Code kann auf die Klasse zugreifen, um eine Referenz auf die notwendige Sperre zu bekommen. Dafür sind die Sperren als annotierte statische Felder implementiert.

Nach den Sperren wird eine Stelle in der Quelldatei markiert. An dieser Stelle können bei Notwendigkeit weitere Sperren hinzugefügt werden.

5.9 Zusammenfassung

Dieses Kapitel hat die Architektur sowie einige weitere Aspekte einer Implementierung des in Kapitel 4 definierten Entwurfs vorgestellt. Detailliert wurden die grafische Oberfläche und der Einsatz von CHESS sowie wesentliche Teile der Implementierung erläutert. Diese sind unter anderem der Aufbau des Code Object Models, die Kontroll- und Datenflussanalyse, die simulierte Ausführung, die Generierung der Korrekturvorschläge und die Code-Mutation.

6. Evaluation

In diesem Kapitel werden die in Kapitel 4 definierten Konzepte zur automatischen Korrektur von Wettlaufsituationen auf der Basis der in Kapitel 5 vorgestellten Implementierung anhand einiger Beispiele auf Korrektheit überprüft. In Abschnitt 6.1 wird zuerst die Evaluierungsstrategie vorgestellt. Danach werden in Abschnitt 6.2 die Voraussetzungen für die Beispiele erläutert. Abschnitt 6.3 beschreibt die Code-Beispiele und die Ergebnisse. Abschließend werden in Abschnitt 6.4 die gesammelten Beobachtungen zusammengefasst.

6.1 Evaluierungsstrategie

Ziel der Evaluierung ist es, festzustellen, ob alle erkannten Datenwettläufe erfolgreich korrigiert werden. Die erste Überprüfung erfolgt durch das implementierte Werkzeug. Nachdem die Code-Änderungen im Quelltext eingebaut wurden, wird der neue Code kompiliert und mit denselben Komponententests erneut auf Parallelitätsfehler überprüft. Treten Fehler auf, wird der alte Zustand des Codes wiederhergestellt. Es werden insgesamt sieben Beispiele in verschiedenen Situationen getestet, um die Korrektheit folgender Komponenten zu verifiziert:

- Analyse der Testergebnisse und Aufbau der Datenwettlaufgruppen
- Sperrenanalyse und Erstellen neuer Sperren
- Datenabhängigkeiten
- Schutzstrategien
- Regionenanpassung
- Ordnung der Sperren
- Mutation des Quelltextes

Für alle sieben Beispiele werden auch die erkannten Datenwettläufe aufgelistet, um den Leser mit den Ausgangsdaten vertraut zu machen. Existiert eine bessere Lösung, dann wird diese mit dem erzeugten Quelltext verglichen.

Die für die Korrektur benötigte Zeit wird nicht gemessen. Diese ist für alle Beispiele unter einer Minute auf einer Testmaschine mit einem Intel Core 2 Duo mit 2.4GHz und 3GB Arbeitsspeicher. Die Laufzeit des Werkzeugs stellt für die praktische Arbeit keine Einschränkung dar, da in Nightly-Build-Umgebung integrierbar ist und die Anzahl zu korrigierenden Fehler im Allgemeinen klein sein sollte.

Im nächsten Abschnitt werden die Voraussetzungen der verwendeten Beispiele erläutert.

6.2 Voraussetzungen und Einschränkungen

Das vorgestellte Verfahren und das darauf basierte Werkzeug zur automatischen Korrektur von Wettlaufbedingungen unterstützen nur eine Teilmenge der C# Sprach-Features. In diesem Abschnitt werden die Voraussetzungen an den für die Evaluierung verwendeten Beispiele beschrieben. Eine Weiterentwicklung des Projektes kann das Werkzeug um die fehlenden Sprachfeatures ergänzen. Unterstützt werden .NET 2.0/3.0-Projekte, die in C# implementiert sind und nur folgende Syntaxelemente enthalten:

- IF-Anweisung
- IF/ELSE-Anweisung
- DO/WHILE-Schleife
- WHILE-Schleife
- FOR-Schleife
- LOCK-Block

Das Werkzeug kann auch beliebige Block-Anweisungen schützen (zum Beispiel `using()`) aber der Code-Erkenner, der das Code Object Model aufbaut, ist momentan nicht in der Lage, diese zu erkennen. An Stellen, an denen es nötig ist, werden die Code-Beispiele angepasst, um die Kompatibilität zu gewährleisten.

Das in Kapitel 4 vorgestellte Verfahren setzt voraus, dass der Code, der korrigiert wird, keine Verklemmungen enthält [E1]. Beispiele, die solche enthalten, werden vor der Korrektur entsprechend angepasst und die Verklemmungen werden beseitigt. Zusätzlich wird sichergestellt, dass die Komponententests korrekt sind und dass alle anderen Probleme, die sie melden, entfernt wurden.

Eine weitere Voraussetzung ist die Erkennung aller Datenwettläufe [E2]. Diese Voraussetzung wird nicht so streng kontrolliert und wird an einige Stellen verwendet, um Werkzeug-Features zu verifizieren.

6.3 Getestete Code-Beispiele und Ergebnisse

Die getesteten Code-Beispiele sind so ausgewählt, dass sie das Verhalten des Werkzeugs in verschiedenen Situationen testen. Diese sind nicht groß, aber trotzdem aussagekräftig und zeigen die Vor- und die Nachteile des Verfahrens. Dank ihrer Übersichtlichkeit sind diese Beispiele schnell zu verstehen und zu erklären und die enthaltenen Fehler können auch schnell vorgestellt werden. Das macht sie geeignet, um die Korrektheit des Verfahrens zu verifizieren und seine Features direkt mit dem korrigierten Code zu zeigen. Die Tabelle auf Abbildung 6.1 bietet eine Übersicht über die korrigierten Anwendungen. Dabei ist zu beachten, dass in der Tabelle die erfolgreichsten Korrekturen der sieben Anwendungen zusammengefasst sind. Probleme, die bei den anderen Korrekturversunchen aufgetreten sind, sind in den entsprechenden Abschnitten erläutert. Die Probleme können mit Zahlenwerten nicht sinnvoll ausgedrückt werden.

6.3.1 CHESS-Beispiele

Microsoft CHESS kommt mit vielen Code-Beispielen, die dafür verwendet werden, den Benutzer mit dem Testwerkzeug vertraut zu machen und die angebotene Funktionalität zu zeigen. Sie enthalten zahlreiche Parallelitätsfehler wie Datenwettläufe und Verklemmungen. Aus den 6 Beispielen werden 2 mit Datenwettläufen ausgewählt und vorgestellt.

Programmname	Testumfang (ohne Tests)	Testumfang (korrigiert, ohne Tests)	Anzahl Testfälle	Erkannte Datenwettläufe	Korrigierte Datenwettläufe	Beteiligte Methoden	Betroffene Felder	Erkannte Datenwettlauf-gruppen
Bank Account	36	65	6	6	6	3	1	1
Dekker	38	73	3	8	8	2	2	2
Parallel Printing	63	85	1	8	8	1	1	1
Limited Queue	44	87	3	15 interne 20 externe	35	2	2 + 2 externe Aufrufe	3
Master/Worker	38	64	1	4	4	2	1	1
RGB	60	127	10	40	40	4	4	4
Statement Diversity	40	86	2	10	10	1	4	4
Summe	/	/	26	111	111	15	15 + 2 externe Aufrufe	16

Abbildung 6.1: Übersicht über die korrigierten Beispielanwendungen.

6.3.1.1 Bank Account

Bank Account implementiert ein Bankkonto. Es werden öffentliche Methoden zum Einzahlen (`Deposit()`), Abheben (`Withdraw()`) und Anzeigen des Kontozustandes (`Read()`) zur Verfügung gestellt. Die drei Methoden greifen auf die Instanzvariable `balance` zu, was bei paralleler Ausführung dieser Methoden zu Datenwettläufen führt. `Deposit()` greift sowohl lesend als auch schreibend, `Read()` greift lesend und `Withdraw()` greift schreibend und mit dem Aufruf von `Read()` auch indirekt lesend auf die Variable zu. D. h. Threadsicher sind in dieser Implementierung nur parallele Aufrufe von `Read()`. Das Beispiel ist als eine .NET-Bibliothek implementiert und kann direkt nicht ausgeführt werden. Das Beispiel ist auf Abbildung 6.2 (a) zu sehen.

(a)

```
1.  public class Account {
2.      private int balance;
3.      public Account(int amount) {
4.          balance = amount;
5.      }
6.      public void Withdraw(int amount) {
7.          int temp = Read();
8.          balance = temp - amount;
9.      }
10.     public int Read() {
11.         int temp;
12.         temp = balance;
13.         return temp;
14.     }
15.     public void Deposit(int amount) {
16.         balance = balance + amount;
17.     }
18. }
```

(b)

```
1.  public void Withdraw(int amount) {
2.      lock (LockClass.StaticLockObject_balance) {
3.          int temp = Read();
4.          balance = temp - amount;
5.      }
6.  }
7.  public int Read() {
8.      int temp;
9.      lock (LockClass.StaticLockObject_balance) {
10.         temp = balance;
11.     }
12.     return temp;
13. }
14. public void Deposit(int amount) {
15.     lock (LockClass.StaticLockObject_balance) {
16.         balance = balance + amount;
17.     }
18. }
```

(c)

```
1.  [AttributeUsage(AttributeTargets.Field)]
2.  public class LockDescriptor : Attribute {
3.      public LockDescriptor(string protectedField) {
4.          this.protectedField = protectedField;
5.      }
6.      string protectedField;
7.  }
8.  public class MC_LockClass {
9.      [LockDescriptor("Bank.Account.balance")]
10.     public static object StaticLockObject_balance = new Object();
11. }
```

Abbildung 6.2: (a) - Quelltext von Bank Account vor der Korrektur. (b) - Quelltext von Bank Account nach der Korrektur. (c) - Automatisch generierten Klassen, die für die Korrektur notwendig sind.

Komponententests

6 Komponententests wurden implementiert, um die parallele Ausführung der Methoden zu überprüfen. Jeder Test erstellt eine Instanz der Klasse `Account` und führt zwei Methoden in separaten Threads parallel aus.

Korrekturergebnisse

Insgesamt viermal wurde das Beispiel korrigiert, um das Verhalten des Werkzeugs in verschiedenen Situationen zu evaluieren. In den nachfolgenden Abschnitten werden die Ergebnisse vorgestellt.

Erste Korrektur: Für die erste Korrektur wird der Code vollständig ungeschützt (siehe Abbildung 6.2 (a)) korrigiert. CHESS hat bei der Ausführung der 6 Komponententests insgesamt 8 Datenwettläufe auf die Variable `balance` erkannt. Die Duplikate wurden aus der nachfolgenden Liste entfernt. Der Code-Zeilen entsprechen die Zeilen auf Abbildung 6.2 (a)):

1. Datenwettlauf zwischen:
   - Schreibzugriff auf Zeile 8 in `Withdraw()`. ILOffset: 12
   - Lesezugriff auf Zeile 12 in `Read()`. ILOffset: 2
2. Datenwettlauf zwischen:
   - Schreibzugriff auf Zeile 8 in `Withdraw()`. ILOffset: 12
   - Schreibzugriff auf Zeile 8 in `Withdraw()`. ILOffset: 12
3. Datenwettlauf zwischen:
   - Schreibzugriff auf Zeile 16 in `Deposit()`. ILOffset: 10
   - Lesezugriff auf Zeile 12 in `Read()`. ILOffset: 2
4. Datenwettlauf zwischen:
   - Schreibzugriff auf Zeile 16 in `Deposit()`. ILOffset: 10
   - Schreibzugriff auf Zeile 8 in `Withdraw()`. ILOffset: 12
5. Datenwettlauf zwischen:
   - Schreibzugriff auf Zeile 16 in `Deposit()`. ILOffset: 10
   - Lesezugriff auf Zeile 16 in `Deposit()`. ILOffset: 3
6. Datenwettlauf zwischen:
   - Schreibzugriff auf Zeile 16 in `Deposit()`. ILOffset: 10
   - Schreibzugriff auf Zeile 16 in `Deposit()`. ILOffset: 10

Bei der Analyse hat das Werkzeug alle Fehlerstellen zu derselben Datenwettlaufgruppe zugeordnet, weil es festgestellt hat, dass alle die Instanzvariable `balance` betreffen. Diese Information liefert CHESS nicht. Für die Korrektur wurde eine neue Sperre erstellt und entsprechend annotiert. Diese ist auf Abbildung 6.2 (c) zu sehen.

Abbildung 6.2 (b) zeigt den automatisch korrigierten Quelltext. Alle Fehlerstellen sind durch `lock`-Blöcke geschützt. Interessant ist die Korrektur der `Withdraw`-Methode: Das Werkzeug hat die Datenabhängigkeit zwischen dem `balance`-Schreibzugriff auf Zeile 4 und dem indirekten Lesezugriff auf Zeile 3 erkannt und hat die zwei Anweisungen zusammen geschützt, um eine Atomizitätsverletzung zu vermeiden. Werden die beiden nicht zusammen geschützt, dann ist eine solche tatsächlich möglich.

Nachdem die Änderungen gespeichert wurden, wurde das Beispiel automatisch neu kompiliert und mit CHESS erneut getestet. CHESS hat keine Fehler im Code erkannt.

Zweite Korrektur: Für die zweite Korrektur wurden die Ergebnisse der Ersten genommen und der `lock`-Block in `Withdraw()` wurde entfernt. CHESS hat alle Datenwettläufe von der ersten Korrektur erkannt, in denen die Methode beteiligt war. Weil die verwendete Sperre vom Werkzeug annotiert wurde, konnte die automatische Korrektur den entfernten `lock`-Block rekonstruieren und den Quelltext zurück auf den Zustand von Abbildung 6.2 (b) bringen. Die geschützten Regionen in `Read()` und `Deposit` wurden als bereits geschützt erkannt und haben keine zusätzlichen `lock`-Blöcke bekommen.

Dritte Korrektur: Für die dritte Korrektur wurde der Quelltext von Abbildung 6.2 (a) genommen und eine manuelle Sperre als Instanzvariable der Klasse eingebaut. Die Ergebnisse sind äquivalent zu den Ergebnissen der ersten Korrektur bis auf die verwendete Sperre. Das Werkzeug hat die Sperre erkannt und diese bei der Korrektur verwendet.

Vierte Korrektur: Für die vierte Korrektur wurden die `Deposit()`- und die `Read()`-Methode mit einer nicht annotierten Sperre geschützt. `Withdraw()` ist in dieser Korrektur nicht geschützt und verursacht Datenwettläufe mit den anderen Methoden. Diese werden alle von CHESS erkannt. Weil die Sperre nicht annotiert ist, erzeugt das Verfahren eine neue Sperre, mit der die drei Fehlerstellen geschützt werden. Die `lock`-Blöcke in `Deposit()` und in `Read()` werden in den bereits vorhandenen `lock`-Blöcken eingebaut. `Withdraw()` wird wie in Abbildung 6.2 (b) geschützt. Es entsteht die Situation, die in Abschnitt 4.12 beschrieben ist. Eine Methodenübergreifende Verklemmung wurde eingebaut. Diese entsteht, wenn `Withdraw()` `Read()` aufruft und einen anderen Thread direkt die `Read()`-Methode ausführt. Das Werkzeug erkennt die Verklemmung und führt die Quelltextmutation nicht aus.

Wird die vorhandene Sperre am Anfang annotiert, dann wird diese für die Korrektur der `Withdraw()`-Methode genommen. Die zwei Regionen in `Deposit()` und in `Read()` wird das Werkzeug als bereits geschützt markieren und es wird keine Verklemmung entstehen.

6.3.1.2 Dekker

Das Dekker-Beispiel implementiert einen wechselseitigen Ausschluss zweier Regionen in einer einfacheren, Dekker-Algorithmus-ähnlichen Form. [Dij02]. Die Implementierung enthält kein aktives Warten [Bus]. Wenn die kritische Region nicht ausgeführt werden darf, dann wird sie übersprungen. Das Beispiel (siehe Abbildung 6.3) besteht aus zwei Methoden (`thread1()` und `thread2()`), die sich gegenseitig ausschließen. Dabei wird auf zwei gemeinsam verwendete statische Variablen zugegriffen `t1_is_entering` und `t2_is_entering`, die die Ausführung der Regionen kontrollieren. `t1_is_entering` ist für `thread1()` verantwortlich und `t2_is_entering` für `thread2()`. Wird eine der beiden Methoden ausgeführt, so wird am Anfang die entsprechende Variable auf `true` gesetzt. Dann wird überprüft, ob die andere Methode im Moment ausgeführt wird, in dem ihre Variable überprüft wird. Vor Verlassen der Methode wird die eigene Variable auf `false` gesetzt, was die Ausführung der anderen Methode freischaltet. Bei parallelem Ausführen der beiden Methoden kommt es zu Datenwettläufen auf die beiden Variablen. Als Seiteneffekt kann es vorkommen, dass eine der Regionen oder sogar beide nicht ausgeführt werden. Zum Beispiel hat der Thread die Region verlassen, aber die Variable noch nicht aktualisiert. Der andere Thread überprüft die Variable und überspringt seine geschützte Region. Dieses CHESS-Beispiel wurde um das Zurücksetzen der Variablen am Ende der Methoden ergänzt.

Komponententests

Für die Korrektur wurden drei Komponententests geschrieben. Sie initialisieren die zwei Variablen mit `false` und führen danach die zwei Methoden in alle drei möglichen Kombinationen in separaten Threads parallel aus.

Korrekturergebnisse

Insgesamt wurde das Beispiel zweimal korrigiert. Die Ergebnisse werden in den nächsten Abschnitten vorgestellt. Anschließend folgt eine Diskussion über die Qualität des erzeugten Quelltextes.

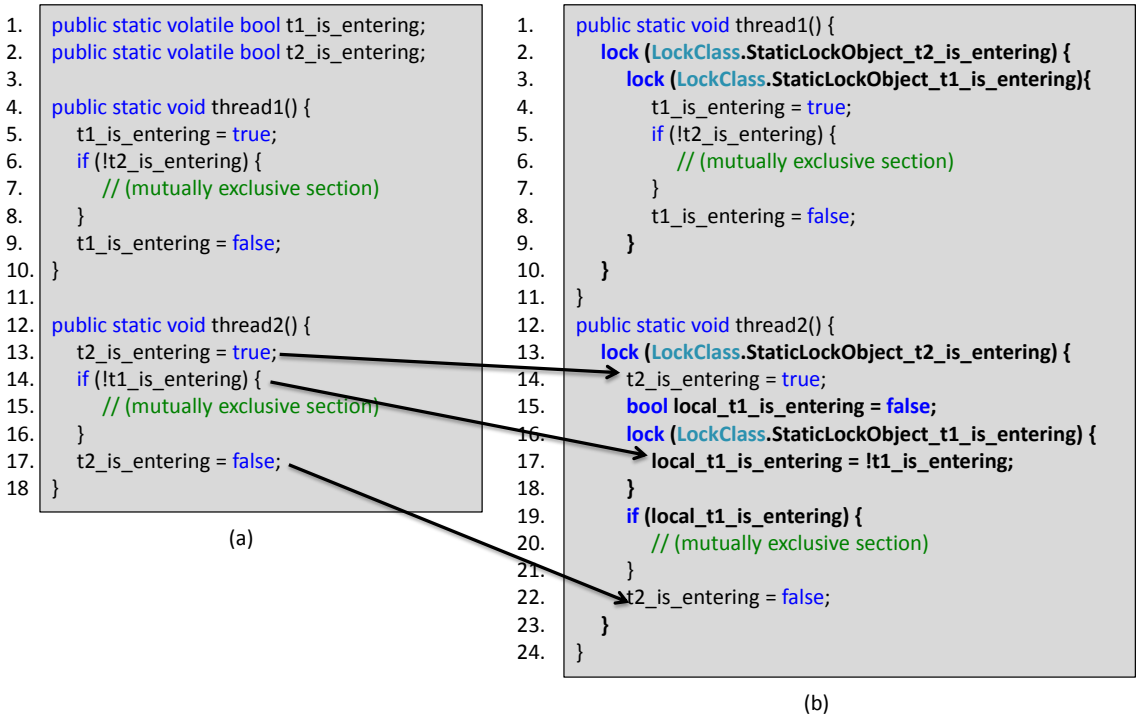

Abbildung 6.3: (a) - Quelltext von Dekker vor der Korrektur. (b) - Quelltext von Dekker nach der Korrektur.

Erste Korrektur: Für die erste Korrektur wurde der Code vollständig ungeschützt (siehe Abbildung 6.3 (a)) verwendet. CHESS hat bei der Ausführung der Komponententests insgesamt 2 Datenwettläufe erkannt:

1. Datenwettlauf auf `t1_is_entering` zwischen:
   - Schreibzugriff auf Zeile 5 in `thread1()`. ILOffset: 4
   - Lesezugriff auf Zeile 14 in `thread2()`. ILOffset: 11
2. Datenwettlauf auf `t2_is_entering` zwischen:
   - Schreibzugriff auf Zeile 13 in `thread2()`. ILOffset: 4
   - Lesezugriff auf Zeile 6 in `thread1()`. ILOffset: 11

Die zwei Datenwettläufe bilden zwei Datenwettlaufgruppen, für die zwei neue Sperren erzeugt und annotiert wurden.

Die Tatsache, dass Zeile 9 und Zeile 17 vom CHESS als Fehlerstellen nicht erkannt werden, erlaubt die Ergebnisse der durchgeführten Datenflussanalyse zu sehen. Zeile 5 und Zeile 9 sind voneinander abhängig und ein Datenwettlauf auf Zeile 5 ist ein Hinweis auf eine mögliche Atomizitätsverletzung, auch wenn kein Datenwettlauf auf Zeile 9 gemeldet wurde. Das Stopper-Konzept 4.10.3.3, das in dieser Arbeit als Idee vorgestellt wurde, kann an dieser Stelle verwendet werden, um die Abhängigkeit zu brechen. Kommen nach Zeile 9 weitere Vorkommnisse der Variablen, die nicht als Fehlerstellen erkannt werden, dann werden sie nicht als Abhängigkeiten interpretiert (siehe Abschnitt 4.10.3). Die Situation in Methode `thread2()` wird äquivalent bearbeitet. Als Ergebnis dieser Analyse wird das Werkzeug die Zeilen 5 bis 9 und 13 bis 17 in zwei `lock`-Blöcken zusammen schützen.

Bei der Analyse der Fehlerstellen in den beiden `if`-Bedingungen wird festgestellt, dass für die betroffenen Variablen keine Abhängigkeiten in den Methoden existieren. Aus diesem

Grund werden die Bedingungen vor der `if`-Anweisung in `lock`-Blöcken extrahiert (siehe Abbildung 6.3 (b), Methode `thread2()`).

Im nächsten Schritt wird eine gegenseitige Schachtelung der Regionen festgestellt. Wird der Code so gespeichert, entsteht bei parallelem Ausführen der beiden Methoden eine Verklemmung. Um diese zu vermeiden, wird das Extrahieren der `if`-Bedingung in `thread1()` eingestellt und stattdessen die Region expandiert und von Zeile 5 bis 9 geschützt. Für die zwei Regionen, die sich vollständig überdecken, wird eine Sperrenordnung eingeführt. Diese kommt aus den geschachtelten Regionen in Methode `thread2()`. Das Ergebnis der Korrektur ist auf Abbildung 6.3 (b) zu sehen.

Nachdem die Änderungen gespeichert wurden, wurde das Beispiel automatisch neu kompiliert und mit CHESS erneut getestet. CHESS hat keine Fehler im Code erkannt.

Zweite Korrektur: Bei der zweiten Korrektur war das Ziel alle Datenwettläufe zu erkennen. CHESS konnte die Fehlerstellen auf Zeile 9 und Zeile 17 nicht erkennen. Eine Anpassung der Komponententests hat nichts gebracht. Ein weiterer Versuch, alle Datenwettläufe zu erkennen, bestand darin, die zwei Variablen nicht als `volatile` [vol, Int06] zu definieren. In diesem Fall konnte CHESS alle Datenwettläufe erkennen. `volatile` verhindert das Auftreten von Datenwettäufen nicht, schaltet nur diverse Compiler-Optimierungen, die die Reihenfolge der CIL-Instruktionen ändern. Nach einer Untersuchung der Assembly wurde keine vertauschte Reihenfolge der Instruktionen festgestellt. D. h. `volatile` beeinflusst die Analyse, die CHESS durchführt, was als Fehler in der Testumgebung definiert werden kann. Nach Entfernen des Schlüsselwortes wurden insgesamt 10 Datenwettläufe erkannt, 2 davon sind Duplikate. Unten sind einige der neuen aufgelistet:

1. Datenwettlauf auf `t2_is_entering` zwischen:
   - Lesezugriff auf Zeile 6 in `thread1()`. ILOffset: 7
   - Schreibzugriff auf Zeile 13 in `thread2()`. ILOffset: 2
2. Datenwettlauf auf `t1_is_entering` zwischen:
   - Schreibzugriff auf Zeile 9 in `thread1()`. ILOffset: 19
   - Lesezugriff auf Zeile 14 in `thread2()`. ILOffset: 29
3. Datenwettlauf auf `t2_is_entering` zwischen:
   - Lesezugriff auf Zeile 6 in `thread1()`. ILOffset: 7
   - Schreibzugriff auf Zeile 17 in `thread2()`. ILOffset: 19

Der erstellte Quelltext ist identisch zu dem Quelltext aus der ersten Korrektur (siehe Abbildung 6.3 (b)).

Diskussion Der automatisch erzeugte Quelltext (siehe Abbildung 6.3 (b)) ist korrekt und alle Datenewettläufe, die im Programm vorhanden waren, wurden entfernt. Betrachtet man den Code, sieht man schnell eine bessere Lösung: das Sperren der beiden Methodenkörper durch einen `lock`-Block. Das vorgestellte Verfahren betrachtet aber die verschiedenen Felder voneinander getrennt und kombinieren separate Datenwettlaufgruppen nicht. Eine weiterführende Arbeit kann an dieser Stelle das Verfahren erweitern, die verschiedenen Fällen analysieren und eine optimiertere Lösung liefern.

6.3.2 Andere Beispiele

Zusätzlich zu den zwei CHESS-Beispielen werden fünf weitere korrigiert. Diese sind so ausgewählt, dass sie noch nicht evaluierte Eigenschaften des Verfahrens überprüfen.

6.3.2.1 Multithreaded Printing

Multithreaded Printing ist ein Beispiel, das von der Diplomarbeit von Filip Dimitrov [Dim11] übernommen wurde. In diesem Beispiel wird aus mehreren Threads auf die Windows-Konsole zugegriffen. Das wird durch die Klasse `MultithreadedPrinter` und die Methode `Print()` ermöglicht (siehe Abbildung 6.4 (a)). `Print()` ist die einzige Methode, die parallel ausgeführt wird. Die Idee der Methode ist, mehrere Zahlen über mehrere Aufrufen der Konsole innerhalb einer `for`-Schleife ununterbrochen, als eine Seite, auszugeben. Der Ausgabetext wird in dieser Methode zuerst in eine Instanzvariable (`consoleOutput`) gespeichert und dann wird diese ausgegeben. Wird die Methode parallel ausgeführt, so sind Datenwettläufe auf die Variable möglich. Als Ergebnis kann kein Thread eine vollständige Seite ganz ausgeben. Wegen der Parallelität werden die Seiten nicht nur stückchenweise und voneinander unterbrochen ausgegeben, sondern es kann wegen der Datenwettläufe auch vorkommen, dass Zeilen vollständig verloren gehen oder mehrfach durch mehrere Threads ausgegeben werden.

```
1.  private string consoleOutput = "";
2.  public void Print() {
3.      Random r = new Random();
4.      int threadId = Thread.CurrentThread.ManagedThreadId;
5.      consoleOutput = threadId + ": started printing: " + System.DateTime.Now;
6.      Console.WriteLine(consoleOutput);
7.      consoleOutput = "";
8.      for (int i = 0; i < 25; i++) {
9.          consoleOutput += string.Format("({0}:{1}), ", threadId, i);
10.         Console.WriteLine(consoleOutput);
11.     }
12.     consoleOutput = threadId + ": finished printing: " + System.DateTime.Now;
13.     Console.WriteLine(consoleOutput);
14. }
```

(a)

```
1.  private string consoleOutput = "";
2.  public void Print() {
3.      Random r = new Random();
4.      int threadId = Thread.CurrentThread.ManagedThreadId;
5.      lock (LockClass.StaticLockObject_consoleOutput) {
6.          consoleOutput = threadId + ": started printing: " + System.DateTime.Now;
7.          Console.WriteLine(consoleOutput);
8.          consoleOutput = "";
9.          for (int i = 0; i < 25; i++) {
10.             consoleOutput += string.Format("({0}:{1}), ", threadId, i);
11.             Console.WriteLine(consoleOutput);
12.         }
13.         consoleOutput = threadId + ": finished printing: " + System.DateTime.Now;
14.         Console.WriteLine(consoleOutput);
15.     }
16. }
```

(b)

Abbildung 6.4: (a) - Quelltext von Multithreaded Printing vor der Korrektur. (b) - Quelltext von Multithreaded Printing nach der Korrektur.

Komponententests

1 Komponententest reicht an dieser Stelle aus, um die `Print()`-Methode auf Parallelitätsfehler zu überprüfen. Der Test erzeugt mehrere Threads, die die Methode parallel ausführen.

Korrekturergebnisse

Dieses Beispiel wurde mehrmals korrigiert. Die Ergebnisse der einzelnen Korrekturen werden in den nachfolgenden Abschnitten vorgestellt.

Erste Korrektur: Für die erste Korrektur wurde der Code vollständig ungeschützt (siehe Abbildung 6.4 (a)) verwendet. CHESS hat bei der Ausführung des Komponententests insgesamt 8 Datenwettläufe auf die Variable `consoleOutput` erkannt:

1. Datenwettlauf zwischen:
   - Schreibzugriff auf Zeile 12 in `Print()`. ILOffset: 170
   - Schreibzugriff auf Zeile 5 in `Print()`. ILOffset: 45
2. Datenwettlauf zwischen:
   - Schreibzugriff auf Zeile 12 in `Print()`. ILOffset: 170
   - Lesezugriff auf Zeile 6 in `Print()`. ILOffset: 51
3. Datenwettlauf zwischen:
   - Schreibzugriff auf Zeile 12 in `Print()`. ILOffset: 170
   - Schreibzugriff auf Zeile 7 in `Print()`. ILOffset: 68
4. Datenwettlauf zwischen:
   - Schreibzugriff auf Zeile 12 in `Print()`. ILOffset: 170
   - Lesezugriff auf Zeile 9 in `Print()`. ILOffset: 80
5. Datenwettlauf zwischen:
   - Schreibzugriff auf Zeile 12 in `Print()`. ILOffset: 170
   - Schreibzugriff auf Zeile 9 in `Print()`. ILOffset:112
6. Datenwettlauf zwischen:
   - Schreibzugriff auf Zeile 12 in `Print()`. ILOffset: 170
   - Lesezugriff auf Zeile 10 in `Print()`. ILOffset: 118
7. Datenwettlauf zwischen:
   - Schreibzugriff auf Zeile 12 in `Print()`. ILOffset: 170
   - Schreibzugriff auf Zeile 12 in `Print()`. ILOffset: 118
8. Datenwettlauf zwischen:
   - Schreibzugriff auf Zeile 12 in `Print()`. ILOffset: 170
   - Lesezugriff auf Zeile 27 in `Print()`. ILOffset: 176

Bei der Analyse hat das Werkzeug alle Fehlerstellen zu derselben Datenwettlaufgruppe zugeordnet, weil es festgestellt hat, dass alle die Instanzvariable `consoleOutput` betreffen. Für die Korrektur wurde eine neue Sperre erstellt und entsprechend annotiert.

Die Datenabhängigkeitsanalyse stellt fest, dass die Vorkommnisse von `consoleOutput` von Zeile 5 bis Zeile 13 voneinander abhängig sind und zusammen geschützt werden müssen. Als Ergebnis werden die einzelnen Regionen vergrößert bis festgestellt wurde, dass diese identisch sind und alle Duplikate entfernt werden. Zeile 3 und Zeile 4 sind weder Fehlerstellen noch von den anderen abhängig. Aus diesem Grund werden sie nicht zum `lock`-Block hinzugefügt. Interessant sind die Aufrufe von Console.WriteLine() und die Parameterübergabe. Diese wird nicht nur vom CHESS erkannt, sondern auch vom Werkzeug als Datenwettlauf auf die Variable korrekt bearbeitet. Abbildung 6.4 (b) zeigt den automatisch korrigierten Quelltext.

Nachdem die Änderungen gespeichert wurden, wurde das Beispiel automatisch neu kompiliert und mit CHESS erneut getestet. CHESS hat keine Fehler im Code erkannt. Eine manuelle Ausführung des Codes zeigt eine korrekte Ausgabe.

Weitere Korrekturen: Wird ein Teil der Zeilen, zum Beispiel die komplette `for`-Schleife, mit einem `lock`-Block geschützt, so wird er im generierten Code übernommen. Es werden wieder dieselben Zeilen geschützt wie auf Abbildung 6.4 (b). Auch wenn die verwendete Sperre annotiert ist, wird die `for`-Schleife doppelt geschützt. Das Entfernen eines bereits vorhandenen `lock`-Blocks ist im Verfahren nicht vorgesehen.

6.3.2.2 Safe Limited Queue

Das SafeLimitedQueue-Beispiel (siehe Abbildung 6.5 (a)) implementiert eine Schlange, die thread-sicher sein soll. Zusätzlich wird diese Schlange mit einer Obergrenze (`limit`) für die maximale Anzahl der gehaltenen Elemente initialisiert. Wird die Grenze erreicht, akzeptiert die Schlange keine Elemente mehr. Die zwei Methoden `Enqueue()` und `Dequeue()` implementieren das Hinzufügen und das Entnehmen von Elementen. Die Variable `current` überwacht die Anzahl der Elemente, die zu einem bestimmten Zeitpunkt in der Schlange gespeichert sind. Zusätzlich wird jede Operation, die auf die Schlange durchgeführt wird mit dem Erhöhen eines Zählers (`numberOfOps`) protokolliert. Die Schlange ist klassenintern durch den thread-unsicheren .NET-Typ `Queue` der `System.dll`-Assembly implementiert [Que]. Bei einer parallelen Ausführung der Methoden `Enqueue()` und `Dequeue()` entstehen Datenwettläufe, die die zwei Instanzvariablen `current` und `numberOfOps` betreffen. Zusätzlich entstehen auch Datenwettläufe in der `Queue`-Klasse, die nur parallele Leseoperationen, aber keine Schreibzugriffe unterstützt.

```
1.  public class SafeLimitedQueue {
2.      private Queue<Object> queue;
3.      private int numberOfOps = 0;
4.      private int current = 0;
5.      private int limit;
6.      public SafeLimitedQueue(int limit) {
7.          this.limit = limit;
8.          queue = new Queue<Object>();
9.      }
10.     public void Enqueue(Object item) {
11.         if (current < limit) {
12.             numberOfOps++;
13.             current = current + 1;;
14.             queue.Enqueue(item);
15.         }
16.     }
17.     public Object Dequeue() {
18.         if (current > 0) {
19.             numberOfOps++;
20.             current = current -1;
21.             return queue.Dequeue();
22.         }
23.         return null;
24.     }
25. }
```

(a)

```
1.  public void Enqueue(Object item) {
2.      lock (LockClass.StaticLockObject_current) {
3.          if (current < limit) {
4.              lock (LockClass.StaticLockObject_numberOfOps) {
5.                  numberOfOps++;
6.              }
7.              current = current + 1;
8.              lock (LockClass.StaticLockObject_1) {
9.                  queue.Enqueue(item);
10.             }
11.         }
12.     }
13. }
14. public Object Dequeue() {
15.     lock (LockClass.StaticLockObject_current) {
16.         if (current > 0) {
17.             lock (LockClass.StaticLockObject_numberOfOps) {
18.                 numberOfOps++;
19.             }
20.             current = current - 1;
21.             lock (LockClass.StaticLockObject_1) {
22.                 return queue.Dequeue();
23.             }
24.         }
25.     }
26.     return null;
27. }
```

(b)

Abbildung 6.5: (a) - Quelltext von Safe Limited Queue vor der Korrektur. (b) - Quelltext von Safe Limited Queue nach der Korrektur.

Komponententests

Drei Komponententests wurden für dieses Beispiel geschrieben. Diese erzeugen eine Instanz der Klasse `SafeLimitedQueue` und rufen in verschiedenen Threads die Methoden `Enqueue()` und `Dequeue()` in alle möglichen Kombinationen parallel auf. Für dieses Beispiel wurde die `System.dll`-Assembly zu der Liste der getesteten Aseemblies in CHESS hinzugefügt. Wird das nicht gemacht, werden die externen Datenwettläufe nicht erkannt.

Korrekturergebnisse

Dieser Abschnitt erläutert die verschiedenen Fälle, bei denen das Beispiel getestet und korrigiert wurde.

Erste Korrektur: Die erste Implementierung dieses Beispiels verwendet die `Count`-Eigenschaft des .NET-Typs `Queue` an der Stelle der Instanzvariable `current`. Beim Testen wurde aber festgestellt, dass CHESS den Datenwettlauf auf `queue.Count` in `Dequeue` (Zeile 18) nicht erkennen konnte. Der Datenwettlauf, der in der Eigenschaft beim Aufrufen von `Enqueue` passiert, wurde korrekt erkannt. Ohne diese Information konnte das Werkzeug den Quelltext nicht erfolgreich korrigieren. Alle Fehlerstellen bis auf die nicht erkannte wurden korrekt angepasst. Nach dem Speichern der Ergebnisse hat CHESS weitere Datenwettläufe gefunden. Aus diesem Grund hat das Werkzeug die alte Codeversion wiederhergestellt, statt eine fehlerhafte zu speichern.

Zweite Korrektur: Für die zweite Korrektur wurde die `queue.Count`-Eigenschaft mit der Variable `current` ersetzt (siehe Abbildung 6.5). In der Testphase hat CHESS alle Fehlerstellen erkannt: 15 interne und 20 externe. Diese, die beim parallelen Ausführen der beiden Methoden gefunden wurden, sind hier aufgelistet:

1. Datenwettlauf auf `current` zwischen:
   - Schreibzugriff auf Zeile 20 in `Dequeue()`. ILOffset: 41
   - Lesezugriff auf Zeile 11 in `Enqueue()`. ILOffset: 22
2. Datenwettlauf auf `current` zwischen:
   - Schreibzugriff auf Zeile 13 in `Dequeue()`. ILOffset: 41
   - Lesezugriff auf Zeile 20 in `Enqueue()`. ILOffset: 39
3. Datenwettlauf auf `current` zwischen:
   - Schreibzugriff auf Zeile 13 in `Dequeue()`. ILOffset: 41
   - Schreibzugriff auf Zeile 20 in `Enqueue()`. ILOffset: 46
4. Datenwettlauf auf `numberOfOps` zwischen:
   - Schreibzugriff auf Zeile 19 in `Dequeue()`. ILOffset: 27
   - Lesezugriff auf Zeile 12 in `Enqueue()`. ILOffset: 25
5. Datenwettlauf auf `numberOfOps` zwischen:
   - Schreibzugriff auf Zeile 19 in `Dequeue()`. ILOffset: 27
   - Schreibzugriff auf Zeile 12 in `Enqueue()`. ILOffset: 32
6. Mehrere Datenwettlaufe in externen Methoden zwischen:
   - `queue.Dequeue()` auf Zeile 21. ILOffset: 47
   - `queue.Enqueue()` auf Zeile 14. ILOffset: 63

Die Aufrufe der externen Methoden verursachen mehrere Datenwettäufe, die an verschiedenen Stellen im externen Code passieren.

Das Verfahren erkennt, dass Datenwettläufe in den externen Methoden gefunden wurden und versucht diese durch `lock`-Blöcke im eigenen Programmcode zu beheben. An dieser

Stelle ist im Verfahren auch eine Korrektur durch Austausch von thread-unsicheren gegen thread-sichere Datenstrukturen vorgesehen (siehe Abschnitt 4.10.5). Diese wurde aber im Werkzeug nicht implementiert. Aus diesem Grund wird die allgemeine Korrekturtechnik für externe Datenwettläufe verwendet. Einer der Nachteile dieses Schützens ist, dass die erzeugte Sperre nicht vom Werkzeug wiederverwendbar ist. Es wurde keine eindeutige Möglichkeit gefunden, die Sperre/geschütztes Objekt-Relation zu definieren. Während der Analyse hat das Verfahren Datenabhängigkeiten für die Variable `current` festgestellt: Die Eine zwischen Zeilen 11 und 13, die Andere zwischen 18 und 20. Wegen dieser Abhängigkeiten werden die Bedingungen auch nicht extrahiert, um eine Atomizitätsverletzung zu vermeiden. Auf Abbildung 6.5 (b) sieht man den automatisch erzeugten Quelltext. Insgesamt drei Sperren wurden angelegt, eine für jede Datenwettlaufgruppe.

Nachdem die Änderungen gespeichert wurden, wurde das Beispiel automatisch neu kompiliert und mit CHESS erneut getestet. CHESS hat keine Fehler im Code erkannt. Eine manuelle Ausführung des Codes zeigt eine korrekte Ausgabe.

Diskussion: Genau so wie beim Dekker-Beispiel, ist auch hier das vollständige Schützen der Methoden eine bessere Lösung. Diese benötigt nur eine Sperre anstatt drei und erhöht sowohl die Codeübersichtlichkeit als auch die Performanz.

6.3.2.3 Master/Worker

In diesem Master/Worker-Beispiel startet der Master-Thread in Zeile 10 (siehe Abbildung 6.6 (a)) einen Worker-Thread, der die `DoSomeWork()`-Methode ausführt. Ab diesem Moment haben die beiden Threads eine gemeinsame Variable `workDone`, die sie um 10 erhöhen, um Arbeit zu simulieren. Zwischen den Zugriffen auf `workDone` auf Zeile 11 und die Zugriffe auf die Variablen in der `DoSomeWork()`-Methode entstehen Datenwettläufe, die zu einem falschen Wert der Variable führen können. Die Zugriffe auf `workDone` bis Zeile 11 und ab Zeile 12 sind von Datenwettläufen nicht betroffen, weil der Master-Thread der einzige laufende Thread zur Zeit der Ausführung dieser Zeilen ist.

```
1.  public class MasterWorker {
2.      private int workDone = 0;
3.      public void StartMaster() {
4.          Thread s = null;
5.          Console.WriteLine("Work done: " + workDone);
6.          workDone += 10;
7.          s = new Thread(l => {
8.              (l as MasterWorker).DoSomeWork();
9.          } );
10.         s.Start(this);
11.         workDone += 10;
12.         s.Join();
13.         Console.WriteLine("Work done: " + workDone);
14.     }
15.     private void DoSomeWork() {
16.         workDone += 10;
17.         for (int i = 0; i < 10; i++) {
18.             workDone += 10;
19.         }
20.     }
21. }
```

(a)

```
1.  private int workDone = 0;
2.  public void StartMaster() {
3.      Thread s = null;
4.      Console.WriteLine("Work done: " + workDone);
5.      lock (LockClass.StaticLockObject_workDone){
6.          workDone += 10;
7.          s = new Thread(l => {
8.              (l as MasterWorker).DoSomeWork();
9.          } );
10.         s.Start(this);
11.         workDone += 10;
12.     }
13.     s.Join();
14.     Console.WriteLine("Work done: " + workDone);
15. }
16. private void DoSomeWork() {
17.     lock (LockClass.StaticLockObject_workDone) {
18.         workDone += 10;
19.         for (int i = 0; i < 10; i++) {
20.             workDone += 10;
21.         }
22.     }
23. }
```

(b)

Abbildung 6.6: (a) - Quelltext von Master/Worker vor der Korrektur. (b) - Quelltext von Master/Worker nach der Korrektur.

Komponententests

In dieser Anwendung läuft nur ein Teil der `StartMaster()`-Methode und die `DoSomeWork()`-Methode parallel. Eine parallele Ausführung der `StartMaster()`-Methode mit sich selbst ist nicht vorgesehen. Das Beispiel wird vom CHESS getestet, in dem in einem Komponententest die `StartMaster()`-Methode aufgerufen wird.

Korrekturergebnisse

Die Anwendung wird ganz ohne Synchronisation getestet und korrigiert. CHESS erkennt alle Datenwettläufe auf die `workDone`-Variable, insgesamt 4:

1. Datenwettlauf zwischen:
   - Schreibzugriff auf Zeile 11 in `StartMaster()`. ILOffset: 100
   - Lesezugriff auf Zeile 16 in `DoSomeWork()`. ILOffset: 3
2. Datenwettlauf zwischen:
   - Schreibzugriff auf Zeile 13 in `StartMaster()`. ILOffset: 100
   - Schreibzugriff auf Zeile 16 in `DoSomeWork()`. ILOffset: 11
3. Datenwettlauf zwischen:
   - Schreibzugriff auf Zeile 13 in `StartMaster()`. ILOffset: 100
   - Lesezugriff auf Zeile 18 in `DoSomeWork()`. ILOffset: 23
4. Datenwettlauf zwischen:
   - Schreibzugriff auf Zeile 13 in `StartMaster()`. ILOffset: 100
   - Schreibzugriff auf Zeile 12 in `DoSomeWork()`. ILOffset: 31

Das Verfahren gruppiert diese Fehlerstellen in einer Datenwettlaufgruppe und erstellt eine neue Sperre.

Zeile 11 ist die Einzige, die in der `StartMaster()`-Methode von Datenwettläufen betroffen ist. Die Existenz der Datenwettläufe ist ein Hinweis auf eine mögliche Atomizitätsverletzung. Die Datenabhängigkeitsanalyse stellt fest, dass der Wert von `workDone` zwischen Zeile 6 und 11 von außen durch einen anderen Thread geändert werden kann. Um das zu verhindern, wird Zeile 6 zusammen mit Zeile 11 geschützt. Für den Zugriff auf Zeile 5 besteht keine Gefahr und dieser wird nicht übernommen. In der anderen Richtung wird auf Zeile 12 die `Thread.Join()`-Methode entdeckt. Diese ist in Abschnitt 4.10.3.3 als Stopper definiert, ein Signal, dass der Entwickler an dieser Stelle die Erwartung auf eine Zustandsänderung hat. Als Ergebnis wird die Datenabhängigkeitsanalyse unterbrochen und Zeile 13 mit dem Lesezugriff auf `workDone` wird nicht in der geschützten Region übernommen. Die `DoSomeWork()`-Methode wird von einem `lock`-Block vollständig geschützt. Das Ergebnis dieser Korrektur sieht man in Abbildung 6.6 (b).

Nachdem die Änderungen gespeichert wurden, wurde das Beispiel automatisch neu kompiliert und mit CHESS getestet. CHESS hat keine Fehler im Code erkannt.

6.3.2.4 RGB

Das RGB-Beispiel (siehe Abbildung 6.7) (a) implementiert eine Klasse zum Speichern und Bearbeiten von RGB-Farben. Die Klasse enthält Methoden zum Setzen (`SetRGB()`), Auslesen (`GetRGB()`) und Invertieren (`Invert()`) der Farbe sowie zum Auslesen ihres Namens (`GetName()`). Die Information wird in vier Instanzvariablen gehalten: `red`, `green`, `blue`,

`name`. `GetRGB()` liefert die Integer-Darstellung der Farbe, die aus den drei Grundfarben beim Aufrufen der Methode berechnet wird. `Invert()` invertiert die Farbe anhand der Grundfarben. Bei paralleler Ausführung der Methoden kommt es zu Datenwettläufen und Atomizitätsverletzungen, die beispielsweise zur Berechnung und Rückgabe einer falschen Farbe führen könnten.

```
1.  private int red;
2.  private int green;
3.  private int blue;
4.  private String name;
5.  public void SetRGB(int red, int green, int blue, String name)
6.  {
7.      if (Check(red, green, blue)) {
8.          this.red = red;
9.          this.green = green;
10.         this.blue = blue;
11.         this.name = name;
12.     }
13. }
14. public int GetRGB() {
15.     return ((red << 16) | (green << 8) | blue);
16. }
17. public String GetName() {
18.     return name;
19. }
20. public void Invert() {
21.     red = 255 - red;
22.     green = 255 - green;
23.     blue = 255 - blue;
24.     name = "Inverse of " + name;
25. }
```

(a)

```
1.  public void SetRGB(int red, int green, int blue, String name) {
2.      if (Check(red, green, blue)) {
3.          lock (LockClass.StaticLockObject_red) {
4.              this.red = red; }
5.          lock (LockClass.StaticLockObject_green) {
6.              this.green = green; }
7.          lock (LockClass.StaticLockObject_blue) {
8.              this.blue = blue; }
9.          lock (LockClass.StaticLockObject_name) {
10.             this.name = name; }
11.     }
12. }
13. public int GetRGB() {
14.     lock (LockClass.StaticLockObject_red) {
15.         lock (LockClass.StaticLockObject_green) {
16.             lock (LockClass.StaticLockObject_blue) {
17.                 return ((red << 16) | (green << 8) | blue);
18.             }
19.         }
20.     }
21. }
22. public String GetName() {
23.     lock (LockClass.StaticLockObject_name) {
24.         return name; }
25. }
26. public void Invert() {
27.     lock (LockClass.StaticLockObject_red) {
28.         red = 255 - red; }
29.     lock (LockClass.StaticLockObject_green) {
30.         green = 255 - green; }
31.     lock (LockClass.StaticLockObject_blue) {
32.         blue = 255 - blue; }
33.     lock (LockClass.StaticLockObject_name) {
34.         name = "Inverse of " + name; }
35. }
```

(b)

Abbildung 6.7: (a) - Quelltext von RGB vor der Korrektur. (b) - Quelltext von RGB nach der Korrektur.

Komponententests

In diesem Code können alle vier Methoden parallel ausgeführt werden. Aus diesem Grund wurden insgesamt 10 Komponententests geschrieben, die jeweils zwei Methoden parallel starten. Diese decken alle Kombinationen der 4 Methoden ab.

Korrekturergebnisse

Die Anwendung wird ganz ohne Synchronisation getestet und korrigiert. CHESS erkennt insgesamt 40 Datenwettläufe auf die vier Instanzvariablen; einige davon sind:

1. Datenwettlauf auf `name` zwischen:
   - Lesezugriff auf Zeile 18 in `GetName()`. ILOffset: 2
   - Schreibzugriff auf Zeile 24 in `Invert()`. ILOffset: 72
2. Datenwettlauf auf `red` zwischen:
   - Schreibzugriff auf Zeile 21 in `Invert()`. ILOffset: 14
   - Lesezugriff auf Zeile 15 in `GetRGB()`. ILOffset: 2

3. Datenwettlauf auf `blue` zwischen:
    - Lesezugriff auf Zeile 15 in `GetRGB()`. ILOffset: 20
    - Schreibzugriff auf Zeile 10 in `SetRGB()`. ILOffset: 34
4. Datenwettlauf auf `green` zwischen:
    - Schreibzugriff auf Zeile 9 in `SetRGB()`. ILOffset: 27
    - Lesezugriff auf Zeile 15 in `GetRGB()`. ILOffset: 11

Das Verfahren gruppiert alle Fehlerstellen in 4 Datenwettlaufgruppen und erstellt vier neue Sperren.

Dieses Beispiel unterscheidet sich von den anderen in der Art seiner Atomizitätsverletzung. Der Zustand des Objektes ist von allen vier Variablen abhängig und alle Methoden müssen aus diesem Grund atomar sein. Diese Form einer Atomizitätsverletzung wird von der Datenabhängigkeitsanalyse nicht erkannt und das Werkzeug korrigiert die Datenwettläufe auf die separaten Variablen nicht zusammen und nicht in geschachtelten `lock`-Blöcken (siehe Abbildung 6.7 (b)). Als Ergebnis können zwei Threads beispielsweise dieselbe Instanz parallel invertieren. Bei einer Thread-Verschränkung treten keine Datenwettäufe auf die einzelnen Variablen mehr auf, bleibt die Instanz aber trotzdem in einem ungültigen Zustand. Eine erfolgreiche Synchronisierung verwendet eine Sperre und schützt die ganzen Methodekörper.

Nachdem die Änderungen gespeichert wurden, wurde das Beispiel automatisch neu kompiliert und mit CHESS getestet. CHESS hat keine Datenwettläufe im Code erkannt.

Eine weiterführende Arbeit kann an dieser Stelle diese Atomizitätsverletzung untersuchen und Vorschläge für das Erkennen machen.

6.3.2.5 Statement Diversity

Das Statement Diversity-Beispiel (siehe Abbildung 6.8 (a)) wurde speziell für die Verifikation zusätzlicher Anweisungstypen entwickelt. Das sind Fälle, die in den anderen Beispielen nicht zu finden sind. Das Beispiel besteht aus einer Methode, die parallel ausgeführt wird. Dabei entstehen Datenwettläufe auf die Variablen `raceInIf`, `raceInWhile`, `raceInCondition`, `doWhileRace`. Auf `ifCond` wird in dieser Methode nur lesend zugegriffen. `loops` ist eine lokale Variable.

Komponententests

Die zwei Komponententests führen die `ExecStatements()`-Methode parallel, mit verschiedenen Werten für die `ifCond`-Variable aus.

Korrekturergebnisse

Die Anwendung wird ganz ohne Synchronisation getestet und korrigiert. CHESS erkennt insgesamt 16 Datenwettläufe auf die vier Instanzvariablen. Nach der Entfernung der Duplikate hat das Verfahren 4 Datenwettlaufgruppen für die Fehlerstellen erstellt.

Abbildung 6.8 (b) zeigt den automatisch erzeugten Quelltext. Die Fehlerstellen in der `if/else`-Anweisung wurden separat geschützt, weil keine Datenabhängigkeit dazwischen gefunden wurde. Sie gehören nie zu demselben Ausführungspfad. Bei der ersten `while`-Schleife wurde nur der Körper geschützt. Bei der Zweiten sowie bei der `do/while`-Schleife wurden Datenwettläufe auf dieselbe Variable sowohl im Körper als auch in der Bedingung erkannt. Aus diesem Grund wurden die ganzen Schleifen geschützt.

Nachdem die Änderungen gespeichert wurden, wurde das Beispiel automatisch neu kompiliert und mit CHESS getestet. CHESS hat keine Datenwettläufe im Code erkannt.

```
1.  public bool ifCond = true;
2.  public int raceInIf = 0;
3.  public int raceInWhile = 0;
4.  public int raceInCondition = 5;
5.  public int doWhileRace = 5;
6.
7.  public void ExecStatements() {
8.      if (ifCond) {
9.          raceInIf++;
10.     } else {
11.         raceInIf--;
12.     }
13.     int loops = 5;
14.     while (loops > 0) {
15.         raceInWhile++;
16.         loops--;
17.     }
18.     while (raceInCondition > 0) {
19.         raceInCondition--;
20.     }
21.     do {
22.         doWhileRace--;
23.     } while (doWhileRace > 0);
24. }
```

(a)

```
1.  public void  ExecStatements() {
2.      if (ifCond) {
3.          lock (LockClass.StaticLockObject_raceInIf) {
4.              raceInIf++;
5.          }
6.      } else {
7.          lock (LockClass.StaticLockObject_raceInIf) {
8.              raceInIf--;
9.          }
10.     }
11.     int loops = 5;
12.     while (loops > 0) {
13.         lock (LockClass.StaticLockObject_raceInWhile) {
14.             raceInWhile++;
15.         }
16.         loops--;
17.     }
18.     lock (LockClass.StaticLockObject_raceInCondition) {
19.         while (raceInCondition > 0) {
20.             raceInCondition--;
21.         }
22.     }
23.     lock (LockClass.StaticLockObject_doWhileRace) {
24.         do {
25.             doWhileRace--;
26.         } while (doWhileRace > 0);
27.     }
28. }
```

(b)

Abbildung 6.8: (a) - Quelltext von StatementDiversity vor der Korrektur. (b) - Quelltext von StatementDiversity nach der Korrektur.

6.3.3 Alle zusammen im großen Kontext

Abschließend wurden alle Beispiele im Kontext einer Anwendung zur parallelen und sequenziellen Desktop-Suche eingebaut. Zusammen mit dem neuen Code und den Tests hat die Anwendung 1700 Codezeilen. Der existierende Code wurde angepasst, damit ausschließlich unterstützte Sprachfeatures verwendet werden.

Die 26 Komponententests haben erfolgreich alle 111 Datenwettläufe erkannt. Diese wurden genauso korrigiert, wie in den einzelnen Beispielen. Dafür wurden 16 neue Sperren erzeugt.

Bei dieser Korrektur wurden keine anderen Teile der Anwendung geändert, bis auf die neu erstellte Klasse für die Sperren und die notwendige Änderung der Projektdatei.

Nach der Korrektur konnte CHESS keine Datenwettläufe mehr finden. Die Anwendung konnte kompiliert und gestartet werden.

6.4 Zusammenfassung

In diesem Kapitel wurde das in dieser Diplomarbeit vorgestellte Verfahren zur automatischen Korrektur von Datenwettläufen mit sieben Beispielanwendungen evaluiert. Das implementierte Werkzeug hat bei allen sieben Beispielen die erkannten Datenwettläufe erfolgreich beseitigt. Die Ergebnisse zeigen auch, dass das Verfahren unter Umständen bei bereits existierender Synchronisierung und vor allem nicht annotierten Sperren im entsprechenden Code-Bereich suboptimale bis nicht funktionierende Lösungen liefern kann. In einem Fall wurde eine Verklemmung eingebaut, öfters wurde ein doppeltes Schützen von Regionen beobachtet. Doppeltes Schützen durch `lock`-Blöcke derselben Sperre ist nur bei einer Kombination aus alter und neuer Synchronisierung aufgetreten. Die Testumgebung hat während der Evaluierung auch Probleme verursacht, die in einem Fall die fehlerfreie Ausführung des Werkzeugs verhindert haben. CHESS konnte vorhandene Datenwettläufe nicht erkennen. Die gemachte Evaluierung gibt gute Hinweise, in welcher weiteren Richtung das Verfahren erweitert werden soll:

1. Entfernen und Anpassen alter Synchronisierung.
2. Bessere, Methodenübergreifende Regionenanpassung.
3. Weitere Optimierung der neuen `lock`-Blöcken und das Schützen mehrere Variablen mit nur einem `lock`-Block.

7. Abschluss

7.1 Zusammenfassung

In dieser Arbeit wurde ein Verfahren zur automatischen Korrektur von Datenwettläufen im parallelen und getesteten Code vorgestellt und sogar implementiert. Fehlerstellen im Programmcode und in externen Bibliotheken werden durch Komponententests erkannt und durch sichere Synchronisationsblöcke und Austausch von parallel unsicheren Datenstrukturen ohne Benutzerbeteiligung im Quelltext korrigiert. Das definierte Code Object Modell ermöglicht eine aufwendige Codeanalyse, die auf Quelltext- und Maschinencodeinformationen basiert, sowie die Behandlung von Schleifen, bedingten Sprüngen und weiteren zusammengesetzten Anweisungen. Das Verfahren erkennt und effektiv beseitigt Atomizitätsverletzungen anhand Datenabhängigkeiten und des vorgestellten Stopper-Konzepts. Eine statische Kontroll- und Datenflussanalyse ermöglicht mithilfe der vorgestellten simulierten Ausführung nicht nur das korrekte Erkennen von Datenabhängigkeiten, sondern auch das Verfolgen von Objekten und das Bestimmen von Laufzeittypen. Der Austausch von Datenstrukturen bei Datenwettläufen, die durch Aufrufe im externen Code verursacht werden, vermeidet das Auftreten neuer Datenwettläufe auf die Datenstruktur als Resultat der Weiterentwicklung des Codes. Die vorgestellte Adapter-Datenstruktur ermöglicht einen einfachen Austausch nicht nur bei syntaktischer, sondern auch bei semantischer Äquivalenz. Das Verfahren ist für die gleichzeitige Korrektur mehrerer Datenwettläufe konzipiert und bietet eine Regionenanpassung sowie eine automatisch generierte Sperrenordnung um Korrektheit und Verklemmungsfreiheit des erzeugten Codes zu gewährleisten. Die erstellte Lösung wird abschließend sogar noch mit den Komponententests, welche die Datenwettläufe erkannt haben, auf Korrektheit überprüft. Somit sind alle Anforderungen erfüllt.

Alle Konzepte bis auf den Austausch von Datenstrukturen wurden in einem Werkzeug implementiert und mit verschiedenen Beispielanwendungen in verschiedenen Situationen getestet. Die Evaluation hat gezeigt, dass das Verfahren Datenwettläufe erfolgreich korrigiert und dabei einen korrekten lauffähigen Code erzeugt. Die Evaluierung hat auf ainige Problemstellen hingewiesen, an denen das Verfahren suboptimalen Code und bei bereits existierender Synchronisierung Verklemmungen erzeugt.

Diese Diplomarbeit bietet eine solide und breite Basis an Konzepten zur automatischen Korrektur von Datenwettläufen an und definiert viele Stellen, an denen diese erweitert werden können. Abschnitt 7.2 stellt die nächsten Schritte vor, durch welche das Verfahren erweitert werden kann.

7.2 Ausblick

In diesem abschließenden Abschnitt wird ein Blick in die Zukunft des Verfahrens geworfen. Es werden verschiedene Verbesserungs- und Erweiterungsideen vorgestellt.

Erweiterung des Code Object Models

Das vorgestellte Code Object Model ist auf eine Darstellung des Methodenkörpers und eine Teilmenge der existierenden Sprachfeatures eingeschränkt. Seine Erweiterung wird nicht nur die Menge an kompatiblen Anwendungen erhöhen, sondern auch die Code-Mutation verbessern. Momentan ist sie eine String-basierte Bearbeitung des Quelltextes, die für komplexe Änderungen ungeeignet ist. Wird das Code Object Model auf eine Darstellung des gesamten Codes erweitert, so können die Code-Anpassungen direkt in einer veränderbaren Kopie des Code Object Models eingebaut werden. Beim Speichern der angepassten Kopie wird der neue Quelltext erzeugt.

Virtuelle Methoden

Der vorgestellte Ansatz kann mit virtuellen Methoden nicht arbeiten [E4]. Bei diesen wird die direkt referenzierte Methode evaluiert, weil die zur Laufzeit aufgerufene Methode nicht vom Quelltext an dieser Stelle ableitbar ist.

Eine Erweiterung des Konzepts könnte mithilfe der vorgestellten simulierten Ausführung und des Datenflussgraphen das Aufrufobjekt bis zu all seinen Quellen zurückzuverfolgen und so die Zielmethoden bestimmen. Eine änliche Analyse wurde in Abschnitt 4.10.5.3 zum Erkennen von Datenstrukturen vorgestellt.

Das Bestimmen der Methoden könnte nach dem folgenden Muster erfolgen:

1. Verfolge das Instazobjekt bis zu all seinen Erstellungsquellen.
2. Bestimme den Typ des Instanzobjekts anhand des aufgerufenen Konstruktors.
3. Nutze die Typinformation, um die aufgerufenen Methoden zu bestimmen.

Erweiterung des Stopper-Konzepts

Das in Abschnitt 4.10.3.3 definierte Stopper-Konzept wurde in dieser Arbeit experimentell beim Bestimmen der korrekten Datenabhängigkeiten eingesetzt. Eine Erweiterung des Konzeptes besteht darin nicht nur die Stopper-Liste zu erweitern, sondern auch eine im besten Fall unbewusste Kommunikation mit dem Entwickler herzustellen.

Stopper können beliebig definiert werden, beispielsweise:

- Methodenaufrufe
- Spezieller Codeaufbau
- Kommentare
- Annotationen

Wird ein Stopper erkannt, können zum Beispiel automatisch Fadenkommunikationsmechanismen eingebaut werden, um effizient auf eine Änderung des Programmszustands zu warten. Solche Mechanismen sind zum Beispiel:

- `Monitor.Wait()`
- `Monitor.Signal()/Notify()`

Editieren bereits vorhandener Synchronisierung

Wie die Evaluierung gezeigt hat, kann die Existenz bereits vorhandener Synchronisierung zu suboptimalen Ergebnissen und Verklemmungen führen. Eine weiterführende Arbeit kann das Verfahren um das Anpassen und Editieren vorhandener Synchronizationsblöcke erweitern, um diese Probleme zu beseitigen.

Weitere Synchronisationsmechanismen

In dieser Arbeit werden ausschließlich Synchronisationsblöcke und Austausch von Datenstrukturen verwendet, um Fehlerstellen zu korrigieren. Eine weiterführende Arbeit kann andere Synchronisationsmechanismen untersuchen, beispielsweise Semaphoren, Leseschreibsperren, Aktives Warten und Barrieren.

Schützen von Schleifen

Die Frage, ob bei einem Datenwettlauf im Schleifenkörper die ganze Schleife geschützt werden muss, wurde in dieser Arbeit offen gelassen. In Abschnitt 4.10.2.2 wurden Vor- und Nachteile der beiden Fälle vorgestellt. Eine weiterführende Arbeit kann sich mit diesem Problem auseinandersetzen.

Literaturverzeichnis

[Ale07] A. Alexandrescu, "Lock-Free Data Structures," p. 7, 12 2007.

[Bas] A. Basu, "C#: Fun with #line directive." [Online]. Available: http://blogs.msdn.com/b/abhinaba/archive/2005/10/10/479016.aspx

[BBC+09] T. Ball, S. Burckhardt, K. E. Coons, M. Musuvathi, and S. Qadeer, "Preempting seeling for efficient concurrency testing," p. 9, 10 2009. [Online]. Available: http://research.microsoft.com/pubs/103177/techreport.pdf

[BBM+] T. Ball, S. Burckhardt, M. Musuvathi, S. Qadeer, and P. de Halleux, "Microsoft Research CHESS," Microsoft Research. [Online]. Available: http://research.microsoft.com/en-us/projects/chess/

[BBMQ08] T. Ball, S. Burckhardt, M. Musuvathi, and S. Qadeer, "First-class concurrency testing and debugging," 2008. [Online]. Available: http://research.microsoft.com/en-us/projects/chess/ec2-submission.pdf

[BEL75] R. S. Boyer, B. Elspas, and K. N. Levitt, "Select - a formal system for testing and debugging programs by symbolic execution," *SIGPLAN Not.*, vol. 10, no. 6, pp. 234–245, Apr. 1975. [Online]. Available: http://doi.acm.org/10.1145/390016.808445

[BFVa] M. Barnett, M. Fahndrich, and H. Venter, "Common Compiler Infrastructure," Microsoft Research. [Online]. Available: http://research.microsoft.com/en-us/projects/cci/

[BFVb] ——, "Common Compiler Infrastructure: Code Model and AST," Microsoft Research. [Online]. Available: http://cciast.codeplex.com/

[BFVc] ——, "Common Compiler Infrastructure: Metadata," Microsoft Research. [Online]. Available: http://ccimetadata.codeplex.com/

[Bon11] F. Bondarenko, "Automatische Quellcodekorrektur von Wettlaufsituationen in parallelen Programmen," 2011.

[Bus] "The open group base specifications issue 6 - lock a spin lock object." [Online]. Available: http://pubs.opengroup.org/onlinepubs/009695399/functions/pthread_spin_lock.html

[Che08] X. Chen, *Verification of Hierarchical Cache Coherence Protocols for Futuristic Processors.* ProQuest, 2008.

[Deva] "MSDN: Devenv command line switches." [Online]. Available: http://msdn.microsoft.com/en-us/library/xee0c8y7.aspx

[Devb] V. Developers, "Helgrind: a thread error detector." [Online]. Available: http://valgrind.org/docs/manual/hg-manual.html

[Dij02] E. W. Dijkstra, “The origin of concurrent programming,” P. B. Hansen, Ed. New York, NY, USA: Springer-Verlag New York, Inc., 2002, ch. Cooperating sequential processes, pp. 65–138. [Online]. Available: http://dl.acm.org/citation.cfm?id=762971.762974

[Dim11] F. Dimitrov, “Automatische Testgenerierung für parallele .NET-Anwendungen,” 2011.

[EA03] D. Engler and K. Ashcraft, “Racerx: effective, static detection of race conditions and deadlocks,” *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 237–252, Oct. 2003. [Online]. Available: http://doi.acm.org/10.1145/1165389.945468

[EFN+02a] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, “Contest - a user’s perspective,” 2002. [Online]. Available: http://www.research.ibm.com/haifa/projects/verification/contest/papers/conTestExp.ps

[EFN+02b] ——, “Multithreaded java program test generation,” *IBM Syst. J.*, vol. 41, no. 1, pp. 111–125, Jan. 2002. [Online]. Available: http://dx.doi.org/10.1147/sj.411.0111

[EFN+03] ——, “Framework for testing multi-threaded java programs,” 2003. [Online]. Available: http://www.research.ibm.com/haifa/projects/verification/contest/papers/conTestExp2.ps

[FF05] C. Flanagan and S. N. Freund, “Automatic synchronization correction,” 2005. [Online]. Available: http://research.microsoft.com/en-us/um/people/tharris/scool/papers/flanagan-freund-scool-final.pdf

[FF07] ——, “Type inference against races,” *Sci. Comput. Program.*, vol. 64, no. 1, pp. 140–165, Jan. 2007. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2006.03.006

[FLL+02] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, “Extended static checking for java,” in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, ser. PLDI ’02. New York, NY, USA: ACM, 2002, pp. 234–245. [Online]. Available: http://doi.acm.org/10.1145/512529.512558

[Gam11] E. Gamma, Ed., *Entwurfsmuster : Elemente wiederverwendbarer objektorientierter Software*, [6. aufl.] ed., ser. Programmers choice. München [u.a.]: Addison-Wesley, 2011, auflagenbez. dem Vorwort entnommen. [Online]. Available: http://swbplus.bsz-bw.de/bsz332627314cov.htm;http://deposit.d-nb.de/cgi-bin/dokserv?id=3541416&prov=M&dok_var=1&dok_ext=htm;http://digitool.hbz-nrw.de:1801/webclient/DeliveryManager?pid=3967202&custom_att_2=simple_viewer

[GHJV04] E. Gamma, R. Heim, R. Johnson, and J. Vlissides, *Design patterns : elements of reusable object-oriented software*, E. Gamma, Ed. Addison-Wesley, 2004.

[HSS09] T. Hansen, P. Schachte, and H. Sondergaard, “State joining and splitting for the symbolic execution of binaries,” in *Runtime Verification*, ser. Lecture Notes in Computer Science, S. Bensalem and D. Peled, Eds. Springer Berlin Heidelberg, 2009, vol. 5779, pp. 76–92, 10.1007/978-3-642-04694-0_6. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04694-0_6

[Inc11a] C. Inc., “Coverity Dynamic Analysis,” 2011. [Online]. Available: http://research.microsoft.com/en-us/projects/chess/osdi2008-chess.pdf

[Inc11b] ——, "Coverity Static Analysis," 2011. [Online]. Available: http://research.microsoft.com/en-us/projects/chess/osdi2008-chess.pdf

[Int06] E. International, *Standard ECMA-334 - C# Language Specification*, 4th ed. ECMA International, June 2006. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-334.htm

[Int10] ——, *Standard ECMA-335 - Common Language Infrastructure (CLI)*, 5th ed. ECMA International, December 2010. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-335.htm

[Int12] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, 2012. [Online]. Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[JBPT09] A. Jannesari, K. Bao, V. Pankratius, and W. F. Tichy, "Helgrind+: An efficient dynamic race detector," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, ser. IPDPS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–13. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2009.5160998

[JUn] "Junit." [Online]. Available: http://junit.sourceforge.net/

[Kar09] A. Karpov, "Interview with Dmitriy Vyukov - the author of Relacy Race Detector," p. 7, 4 2009.

[Kin76] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976. [Online]. Available: http://doi.acm.org/10.1145/360248.360252

[KLT+07] B. Krena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar, "Healing data races on-the-fly," in *Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging*, ser. PADTAD '07. New York, NY, USA: ACM, 2007, pp. 54–64. [Online]. Available: http://doi.acm.org/10.1145/1273647.1273658

[Kni] K. Knizhnik, "Jlint." [Online]. Available: http://artho.com/jlint/

[Kri71] S. Kripke, *Semantical Considerations on Modal Logic ; Naming and Necessity*. Oxford University Press, 1971. [Online]. Available: http://books.google.de/books?id=HeZpHQAACAAJ

[Lev85] N. Leveson, "Medical devices: The therac-25," 1985.

[LGDVFW12] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair," 2012.

[LGNFW12] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, Jan. 2012. [Online]. Available: http://dx.doi.org/10.1109/TSE.2011.104

[LHHW09] K. Leung, Z. Huang, Q. Huang, and P. Werstein, "Maotai 2.0: Data race prevention in view-oriented parallel programming," in *Proceedings of the 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, ser. PDCAT '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 263–271. [Online]. Available: http://dx.doi.org/10.1109/PDCAT.2009.12

[LT93] N. G. Leveson and C. S. Turner, "An investigation of the therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, Jul. 1993. [Online]. Available: http://dx.doi.org/10.1109/MC.1993.274940

[Mica] "Microsoft Visual Studio," Microsoft Corporation. [Online]. Available: http://msdn.microsoft.com/en-us/vstudio/

[Micb] "MSDN: Windows forms," Microsoft Corporation. [Online]. Available: http://msdn.microsoft.com/en-us/library/dd30h2yb.aspx

[Micc] ".NET Framework Developer Center," Microsoft Corporation. [Online]. Available: http://msdn.microsoft.com/en-us/netframework

[Micd] "WindowsClient.Net," Microsoft Corporation. [Online]. Available: http://windowsclient.net/

[MQB+08] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and Reproducing Heisenbugs in Concurrent Programs," p. 14, 2008. [Online]. Available: http://research.microsoft.com/en-us/projects/chess/osdi2008-chess.pdf

[Muc07] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 2007.

[Net] M. D. Network, ".NET Framework Class Library: Threadstate Enumeration," Microsoft Corporation. [Online]. Available: http://msdn.microsoft.com/en-us/library/system.threading.threadstate.aspx

[NS07] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, Jun. 2007. [Online]. Available: http://valgrind.org/docs/valgrind2007.pdf

[NWGH] K. Ng, M. Warren, P. Golde, and A. Hejlsberg, "The Roslyn Project," Microsoft Corporation. [Online]. Available: http://msdn.microsoft.com/en-us/roslyn

[Pat] "Java pathfinder." [Online]. Available: http://javapathfinder.sourceforge.net/

[Pdb] "MSDN: Program Database Files (C#, F#, and Visual Basic)." [Online]. Available: http://msdn.microsoft.com/en-us/library/ms241903.aspx

[Pou04] K. Poulsen, "Tracking the blackout bug," *SecurityFocus*, 2004. [Online]. Available: http://www.securityfocus.com/news/8412

[Que] "MSDN: Queue(t) class." [Online]. Available: http://msdn.microsoft.com/en-us/library/7977ey2c%28VS.110%29.aspx

[RBK+09] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman, "Detecting and tolerating asymmetric races," *SIGPLAN Not.*, vol. 44, no. 4, pp. 173–184, Feb. 2009. [Online]. Available: http://doi.acm.org/10.1145/1594835.1504202

[SBN+97] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, Nov. 1997. [Online]. Available: http://doi.acm.org/10.1145/265924.265927

[Sch10] V. Schimmel, Jochen ; Pankratius, "Tachorace: Exploiting performance counters for run-time race detection," Karlsruhe, 2010. [Online]. Available: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000017053

[Sta] M. Stall, "#line hidden and 0xfeefee sequence points." [Online]. Available: http://blogs.msdn.com/b/jmstall/archive/2005/06/19/feefee-sequencepoints.aspx

[Sze09] G. Szeder, "Unit testing for multi-threaded java programs," in *Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, ser. PADTAD '09. New York, NY, USA: ACM, 2009, pp. 4:1–4:8. [Online]. Available: http://doi.acm.org/10.1145/1639622.1639626

[Tan09] A. S. Tanenbaum, *Modern Operating Systems*, 3rd ed., T. D. Holm, Ed. Alan Apt, 2009.

[VHBP00] W. Visser, K. Havelund, G. Brat, and S. Park, "Model checking programs," in *Proceedings of the 15th IEEE international conference on Automated software engineering*, ser. ASE '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 3–. [Online]. Available: http://dl.acm.org/citation.cfm?id=786768.786967

[vol] "MSDN: Volatile fields." [Online]. Available: http://msdn.microsoft.com/en-us/library/aa645755%28VS.71%29.aspx

[Vyu] D. Vyukov, "Relacy Race Detector." [Online]. Available: http://www.1024cores.net/home/relacy-race-detector