Statisches Erkennen von race conditions

Einsatz von P# zur Modellierung von Nebenläufigkeit

Thomas Ulrich

21. Dezember 2015

Inhaltsverzeichnis

1 Einführung

Nebenläufige Anwendungen nutzen mehrere Threads oder Prozesse, um beispielsweise eine reaktive Benutzeroberfläche bereitzustellen oder um die Rechenlast auf mehrere Prozessorkerne zu verteilen, um in kürzerer Zeit das gewünschte Ergebnis zu berechnen. Die nebenläufige Ausführung zweier Berechnungen führt dabei häufig zu einem Nichtdeterminismus in der Anwendung, der von verschiedenen Faktoren herrührt, wie dem CPU-Scheduler oder Latenzzeiten beim Zugriff auf eine Ressource über ein Netzwerk.

Teilen sich zwei Berechnungen, die nebenläufig ausgeführt werden eine Ressource wie z.B. ein Objekt im RAM, kann dies zu unerwünschten Nebeneffekten führen. Dies ist insbesondere der Fall, wenn beide Berechnungen zeitgleich versuchen, auf dieses Objekt zuzugreifen und mindestens eine Berechnung versucht, dieses Objekt zu verändern. Man spricht in diesem Fall von einer *race condition*, bei der die Ausführungsreihenfolge der Berechnungen entscheidenden Einfluss auf das Ergebnis beider Berechnungen haben kann. Auf race conditions basierende Programmfehler sind schwer zu finden, da sie aufgrund des Nichtdeterminismus der Anwendung nicht in jedem Programm-Durchlauf auftreten müssen und damit nicht zwingend durch regressive Testverfahren wie Unit-Tests erkannt werden.

Um race conditions zu verhindern, müssen Zugriffe auf Objekte synchronisiert werden. Dazu können sogenannte *locks* genutzt werden, die sicherstellen, dass (Schreib)zugriffe auf eine Ressource exklusiv erfolgen, also kein anderer Thread in dieser Zeit das Objekt lesen oder verändern kann. Locks bieten Schutz vor race conditions, erfordern jedoch die erhöhte Aufmerksamkeit des Programmierers, da sie zu anderen Problemen führen können. Beispielsweise kann es zu einem sogenannte *deadlock* kommen, wenn für eine bestimmte Operation der Besitz von zwei Locks erforderlich ist. Wenn nun zwei Threads beide diese Operation durchführen, und je ein Thread eines der Locks hält, kann kein Thread fortfahren da er auf den jeweils anderen wartet.

Deligiannis et al. präsentieren in [2] die Programmiersprache P#, die auf der Modellierung von Zustandsautomaten basiert und eine statische Analyse des Codes bezüglich möglicher race conditions erlaubt. P# selbst basiert auf C# und erlaubt es, C# Code bei der Portierung nach P# weiterzuverwenden. Im Folgenden wird diese Programmiersprache vorgestellt und anhand von Beispielen gezeigt, wie damit race conditions gefunden und vermieden werden können.

2 P# Grundlagen

P#-Programme basieren auf endlich vielen, deterministischen Zustandsautomaten. Ein Zustandsautomat wird dabei als Subklasse der vordefinierten abstrakten Klasse *Machine* modelliert. Zustandsautomaten können untereinander über *Events* kommunizieren, wobei ein Event leer sein oder Nutzdaten (sog. *payload*) enthalten kann. Außer über Events dürfen keine Daten ausgetauscht werden, d.h. ein Automat darf keine öffentlichen Methoden oder Variablen enthalten auf die andere Automaten zugreifen können (z.B. public fields oder public properties).[1]

In jedem Zustandsautomaten wird über *Actions* definiert, ob und wie der Zustandsautomat in einem bestimmten Zustand auf ein bestimmtes Event reagiert. Actions sind beliebige in C# geschrieben Methoden, mit der Einschränkung, dass sie keine neuen Threads starten und keine Synchronisationsmechanismen nutzen dürfen. Über eine Zustandsübergangsfunktion definiert jeder Automat, mit welcher Action in einem bestimmten Zustand auf ein bestimmtes Event reagiert werden soll. Die Zustandsübergangsfunktion wird pro Zustand über Attribute definiert, sodass ein Automat auf ein Event in unterschiedlichen Zuständen unterschiedlich reagieren kann. Empfängt ein Zustandsautomat ein Event, für das die Zustandsübergangsfunktion im aktuellen Zustand nicht definiert ist, wird eine Exception ausgelöst.

Die eigentliche Business-Logik wird in den Actions und Zuständen der Automaten gekapselt, die Zustandswechsel werden über Events getriggert. Über Attribut-Annotationen wird pro Automat ein Zustand als Initialzustand deklariert. Es können mehrere Instanzen eines Automatentypen initialisiert werden. Es müssen aber alle Automatentypen vom Programmierer zur Compilezeit bei der P#-Runtime registriert worden sein, d.h. es dürfen zur Laufzeit keine neuen Automatentypen definiert werden. [2] Diese Einschränkung ist erforderlich, um eine statische Analyse zu ermöglichen. Siehe dazu 3.

Events können als Nutzdaten entweder konkrete Werte oder Referenzen auf Objekte auf dem von allen Zustandsautomaten gemeinsam genutzten Heap enthalten. Letzteres erlaubt es den Automaten auf Serialisierung oder Cloning zu verzichten, bringt aber das bei pass-by-reference bekannten Problem des *aliasing* mit sich. Als Aliasing wird das Vorhalten von mehreren Referenzen auf das gleiche Objekt im Speicher bezeichnet. Kritisch ist dies vor allem, wenn unterschiedliche Threads über diese Referenzen gleichzeitig auf das referenzierte Objekt zugreifen, was zu zu race conditions führen kann. Da P# das Versenden von Referenzen in Events erlaubt, sind P#-Programme nicht per Design frei von race conditions.

Statt race conditions durch das Design der Sprache auszuschließen, erlaubt P# eine statische Analyse des Codes um mögliche race conditions aufzudecken. Dazu wird eine *ownership-basierte* Analyse durchgeführt. Der Grundsatz dieser Analyse ist, dass eine Action eines Automaten an definierten Stellen den Besitz eines Objekts im Speicher übernimmt und aufgibt. Zugriffe auf ein Objekt, dessen Besitz aufgegeben wurde, verstoßen gegen das ownership-Prinzip und führen in der Analyse zu Fehlermeldungen.

[1] Properties sind ein C#-Konstrukt, die von außen wie public Variablen aussehen, aber Programmlogik zum Lesen oder Setzen des Wertes kapseln können. Siehe auch `https://msdn.microsoft.com/en-us/library/x9fsa0sw.aspx`

3 Ownership-basierter Analyseansatz

Der ownership-basierten Analyse liegt folgender Leitgedanke zugrunde: Objekte auf dem Heap gehören Zustandsautomaten. Nur der Besitzer des Objekts darf dieses lesen oder schreiben. Der Besitz eines Objekts o geht von Automat m auf Automat m' über, wenn m ein event an m' sendet, das eine direkte oder indirekte Referenz auf o enthält. Ein Zugriff von m auf Objekt o nach Übergabe des Besitzes ist nicht zulässig. Wird ein solcher Zugriff festgestellt, liegt eine mögliche race condition vor. [2]

Um zu überprüfen, ob diese Bedingung für alle Objekte in einem P#-Programm erfüllt ist, müssen folgende Fragen geklärt werden:

1. Welche Zustandsautomaten haben eine Referenz auf ein bestimmtes Objekt?
2. Wann übernimmt ein Automat den Besitz eines Objekts und wann gibt er ihn auf?
3. Greifen ein oder mehrere Automaten auf ein referenziertes Objekt zu, ohne Besitzer dieses Objekts zu sein?

Die Analyse erfolgt dabei auf Methoden-Ebene. Dazu wird zu jeder Methode die *give-up* Menge bestimmt, die die Menge der Variablen umfasst, deren Besitz beim Verlassen der Methode aufgegeben wird. Deligiannis et al. beschreiben die give-up Menge wie folgt:

> „[...]we define a give-up set for each method m. For each formal parameter *fp* of m, if there exists an object o such that (i) o is reachable from *fp* (either directly or transitively following references) and (ii) ownership of o is transferred by m, then *fp* is in the give-up set for m.“ [2]

Ist die give-up Menge jeder Methode bestimmt, kann anschließend überprüft werden, ob eine race condition auftreten kann. Dazu wird das folgende Szenario betrachtet (nach [2]):

1. Ein Parameter *fp* ist in der give-up Menge einer Methode m' enthalten.
2. Eine Methode m ruft m' auf und übergibt die Variable v als Wert für den Parameter *fp*.
3. Nach Aufruf von m' greift m auf eine Variable v' zu.

Eine race condition kann ausgeschlossen werden, wenn kein Objekt o existiert, auf das direkt oder indirekt über v und v' zugegriffen werden kann.

Für die Analyse werden die Methoden als *control flow graph (CFG)* dargestellt. Jeder CFG hat genau einen Anfangs- und einen Endknoten, die mit *Entry* und *Exit* gekennzeichnet werden. Für Methoden werden folgende Annahmen getroffen:

1. Die einer Methode übergebenen Variablen können nicht verändert werden.
2. Die einer Methode übergebenen Variablen sind paarweise verschieden, d.h. es erfolgt kein Aliasing.

Für die Analyse werden nur Referenzen auf Objekte auf dem Heap betrachtet, da Parameter, die per call-by-value übergeben werden Kopien der tatsächlichen Werte sind und keine race conditions auftreten können.

3.1 Control flow graphs

Pro Methode wird ein control flow graph (CFG) erstellt, der genau einen Anfangs- und einen Endknoten hat. Jeder Knoten im Graph entspricht genau einem Statement in der Methode. Das nachfolgende Listing zeigt eine Methode, in der alle geraden Zahlen zwischen einer unteren und einer oberen Grenze ausgegeben werden.

```
1  public void AnnounceEven(int lowerBounds, int upperBounds)
2  {
3      for (int i = lowerBounds; i <= upperBounds; i++)
4      {
5          if (i % 2 == 0)
6          {
7              Console.WriteLine("Even: {0}", i);
8          }
9      }
10 }
```

Listing 1 : Ausgabe aller geraden Zahlen zwischen zwei Grenzen.

Diese Methode lässt sich als CFG wie folgt visualisieren:

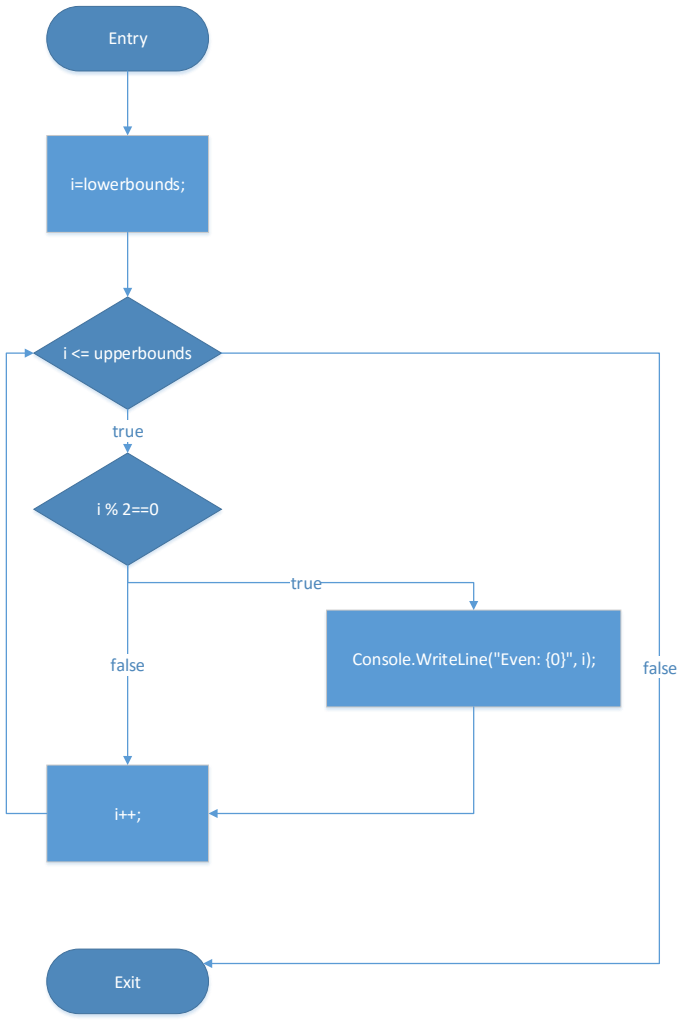

Abbildung 1: CFG zur Funktion AnnounceEven

Da es sich um einen gerichteten Graphen handelt, können pro Knoten Vorgänger und Nachfolger bestimmt werden. Die Knoten können durchnummeriert werden, wobei die Nummerierung der Knoten der Reihenfolge der Statements im Quellcode entsprechen. Mittels dieser Knoten-Informationen kann nun überprüft werden, ob Variablen in unterschiedlichen Knoten des Graphs auf das gleiche Objekt auf dem Heap verweisen (heap overlap).

3.2 Heap overlap Analyse

Die heap overlap Analyse zeigt, ob es für Knoten N, N' eines CFG für Methode m Variablen v, v' gibt, über die direkt oder indirekt ein Objekt o referenziert werden kann. Dazu wird in [2] ein Prädikat $may\_overlap$ definiert, das zwei Fälle unterscheidet:

1. $may\_overlap_{m,in}^{(N,v)}(N', v')$ ist erfüllt, wenn ein Objekt o existiert, das bei Eintritt in Knoten N über Variable v erreichbar ist und das bei Eintritt in Knoten N' über Variable v' erreichbar ist.
2. $may\_overlap_{m,out}^{(N,v)}(N', v')$ ist erfüllt, wenn ein Objekt o existiert, das bei Eintritt in Knoten N über Variable v erreichbar ist und das bei Verlassen von Knoten N' über Variable v' erreichbar ist.

In beiden Fällen werden die Objekte betrachtet, die beim Eintritt in N über v erreichbar sind. Hintergrund ist die Übertragung des Besitzes. So kann der Knoten N das Versenden eines events enthalten, bei dem der Besitz für ein erreichbares Objekt an einen anderen Automaten übertragen wird. In 3.3 wird näher darauf eingegangen.

In der Praxis haben Deligiannis et al. die heap overlap Analyse als eine *taint tracking Analyse* implementiert. Eine Variable v' wird dabei als *tainted* (engl. verdorben, vergiftet) bezeichnet, wenn es ein Objekt o gibt, das sowohl von v' als auch von einer Variable v erreichbar ist. Dazu wird eine Funktion $tainted$ definiert:

> „Given a method m, a local variable or formal parameter v of m and a node N of m, the analysis yields a function $tainted_m^{(v,N)}$ that maps nodes N' of m to variables v' such that if v is assumed to be tainted on entry to N, then v' is tainted on exit from N'. In other words, $v' \in tainted_m^{(v,N)}(N')$ implies that there may exist an object o such that v' can reach o on exit from N' if v can reach o on entry to N." [2]

Bei der Implementierung der Funktion wird die Analyse dahingehend vereinfacht, dass – sollte eine Member-Variable eines Objekts als tainted markiert werden – das gesamte Objekt als tainted markiert wird.

Damit lässt sich das Prädikat $may\_overlap$ nun vereinfacht wie folgt definieren:

1. $may\_overlap_{m,in}^{(N,v)}(N', v') \triangleq v' \in \bigcup_{P \in pred(N')} tainted_m^{(v,N)}(P)$
   wobei $pred(N')$ die Menge der Vorgänger des Knotens N' bezeichnet.
2. $may\_overlap_{m,out}^{(N,v)}(N', v') \triangleq v' \in tainted_m^{(v,N)}(N')$

Das folgende Listing zeigt eine einfache Funktion m, die ein Array und eine neue Variable erstellt und diese neue Variable ins Array einträgt.

```
1 private void m()
2 {
3     object[] store = new object[2];
4     object v = new object();
5     store[0] = v;
6 }
```

Listing 2 : Funktion m.

Der CFG für diese Funktion sieht wie folgt aus:

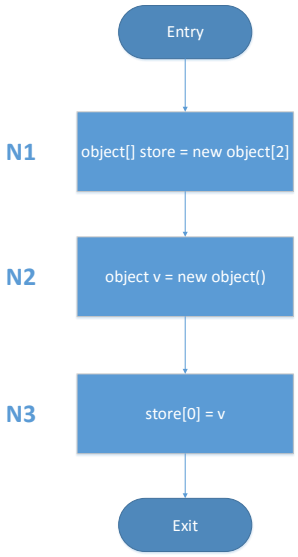

Abbildung 2: CFG zur Funktion m

Die nachfolgende Tabelle zeigt die Werte für die Taint-Tracking-Analyse.

	x = store	x = v
$tainted_m^{(x,N1)}(N1)$	$\emptyset$	$\emptyset$
$tainted_m^{(x,N1)}(N2)$	$\emptyset$	$\emptyset$
$tainted_m^{(x,N1)}(N3)$	$\emptyset$	$\emptyset$
$tainted_m^{(x,N2)}(N1)$	$\{store\}$	$\emptyset$
$tainted_m^{(x,N2)}(N2)$	$\{store\}$	$\emptyset$
$tainted_m^{(x,N2)}(N3)$	$\{store\}$	$\emptyset$
$tainted_m^{(x,N3)}(N1)$	$\{store\}$	$\emptyset$
$tainted_m^{(x,N3)}(N2)$	$\{store\}$	$\{v\}$
$tainted_m^{(x,N3)}(N3)$	$\{store\}$	$\{v, store\}$

Tabelle 1: Ergebnisse der Taint-Tracking-Analyse für m

Es zeigt sich, dass es beim Verlassen des Knotens N3 ein Objekt gibt, das sowohl durch Variable v als auch durch die Variable $store$ referenziert werden kann. Da $store \in tainted_m^{(v,N3)}(N3)$, gilt folglich: $may\_overlap_{m,out}^{(N3,v)}(N3, store) = true$.

3.3 Gives-up Analyse

Die *gives-up Analyse* untersucht eine Methode m dahingehend, für welche Parameter die Methode den Besitz aufgibt. Dies kann grundsätzlich aufgrund von zwei Dingen passieren:

- Das für den Parameter übergebene Objekt wird per event weitergegeben.
- Das für den Parameter übergebene Objekt wird als Wert für einen Parameter einer anderen Methode eingesetzt.

Formal lässt sich die Menge der Parameter, deren Besitz aufgegeben wird, über eine Fixpunkt-Iteration berechnen. Da die Anzahl der Methoden und ihrer Parameter endlich ist, terminiert diese Iteration.

```
1 gives_up(m) = ∅
2 repeat
3   for all m do
4     gives_up(m) := ⋃_{N∈m} gives_up_m^fp(N)
5   end for
6 until gives_up no longer changes
```

Listing 3 : Pseudocode der gives-up Berechnung

Die Funktion $gives\_up_m^{fp}(N)$ ist nachfolgend definiert. Dabei bezeichnet $fp(m)$ die Menge der Parameter der Funktion m, $send_{dst}evt(v)$ das Versenden eines Events und $v'.m'(v_1, ..., v_n)$ den Aufruf einer Methode.

$$gives\_up_m^{fp}(N) = \begin{cases} \{w \in fp(m) | may\_overlap_{m,out}^{(N,v)}(Entry, w)\} & \text{falls } N = send_{dst}\, evt(v) \\ \quad \bigcup_{1 \le i \le n} \{w \in fp(m) | fp_i \in gives\_up(m') & \\ \quad\quad \wedge may\_overlap_{m,out}^{(N,v_i)}(Entry, w)\} & \text{falls } N = v := v'.m'(v_1, ..., v_n) \\ \quad\quad \emptyset & \text{Sonst} \end{cases}$$

Das nachfolgende Listing zeigt eine Funktion m, die drei Objekte als Parameter nimmt und eine Methode $m2$ aufruft sowie ein Event versendet.

```
1 private void m(object param1, object param2, object param3)
2 {
3     object value = this.m2(param1);
4     this.Send(param2, value);
5     string text = value.ToString() + param3.ToString();
6 }
```

Listing 4 : Funktion m.

Der CFG zu dieser Methode enthält drei Knoten, wobei Knoten N1 Zeile 3 des Listings entspricht. Analog dazu entspricht der Knoten N2 Zeile 4 und Knoten N3 Zeile 5. Im Rahmen dieses Beispiels gelte $gives\_up(m2) = \{param1\}$. Dann gilt:

$gives\_up_m^{fp}(\text{N1}) = \{param1\}$
$gives\_up_m^{fp}(\text{N2}) = \{param2\}$
$gives\_up_m^{fp}(\text{N3}) = \emptyset$
$gives\_up(m) = \{param1, param2\}$

3.4 Respects-Ownership Analyse

Mit den Informationen aus der heap overlap- und der gives-up Analyse kann nun ermittelt werden, ob der Zugriff auf Objekte eines Programms nur durch den jeweiligen Besitzer dieses Objekts geschehen, oder ob Methoden auf Objekte zugreifen, die sie nicht besitzen. In diesem Fall kann es zu race conditions kommen.

Wie unter 3.3 gezeigt, wird der Besitz eines Objekts nur dann transferiert, wenn eine Referenz auf dieses Objekt per event verschickt oder als Parameter-Wert bei einem Methodenaufruf eingesetzt wird. Daher beschäftigt sich die respects-ownership Analyse nur mit diesen beiden Fällen.

Deligiannis et al. definieren in [2] drei Bedingungen, die eingehalten werden müssen, damit in einer Methode keine ownership-Konflikte auftreten. Sei N ein Knoten im CFG einer Methode m und sei $vars(N)$ die Menge der Variablen, die in N vorkommen. Für eine Variable w, deren Besitz im Knoten

N aufgegeben wurde, liegen keine ownership-Konflikte vor, wenn die folgenden Bedingungen für alle Knoten N' des CFG von m eingehalten werden:

1. Wenn es einen Pfad von $Entry$ zu N über N' gibt, dann muss gelten: $\neg\, may\_overlap_{m,out}^{(N,w)}(N', this)$

   Diese Bedingung sichert zu, dass der Automat zu dem die Methode m gehört keine Referenz auf ein Objekt o hält, das über die aufgegebene Variable w und über $this$ aufgelöst werden kann, also z.B. ein privates Feld im Automaten. In diesem Fall wäre der Zugriff auf o trotz Besitzaufgabe von w weiterhin möglich.

2. Wenn N= N', dann muss gelten $w \neq this$ und
   $\{ v \in vars(N') \mid may\_overlap_{m,in}^{(N,w)}(N', v) \}$ = $\{ w \}$

   Diese Bedingung sichert zu, dass $w \neq this$, denn der Besitz eines Automaten kann nicht aufgegeben werden. Zusätzlich verhindert diese Bedingung, dass es beim Eintritt in den Knoten N Aliasing für die in diesem Knoten genutzten Variablen gibt, dass also keine Variablen v, w existieren, sodass v und w auf das gleiche Objekt o im Speicher verweisen. Auch in diesem Fall wäre dann weiterhin der Zugriff auf o (über v) möglich, obwohl der Besitz von w aufgegeben wurde.

3. Wenn es einen Pfad von N zu $Exit$ über N' gibt, dann muss gelten:
   $\{ v \in vars(N') \mid may\_overlap_{m,in}^{(N,w)}(N', v) \}$ = $\emptyset$

   Diese Bedingung sichert zu, dass es in keinem Nachfolgerknoten $N' \in successor(N)$ eine Variable v gibt, die auf das gleiche Objekt o im Speicher verweist wie w.

Anhand dieser Bedingungen werden nun alle Methoden aller Automaten des Programms analysiert. Da die Anzahl der Automatentypen endlich ist (vgl. 2) und jeder Automat nur endlich viele Methoden definieren kann, terminiert die Analyse. Wird eine Verletzung der obigen Bedingungen festgestellt, wird dies bei der Analyse als eine mögliche race condition vermerkt.

4 Beispiel

4.1 Dictionary mit race condition

Im Folgenden wird eine typische race condition bei Benutzung eines *Dictionarys* dargstellt. Ein Dictionary in C# ist ein Objekt, das zu einem Schlüssel maximal ein Objekt als Wert enthalten kann. Wird versucht, ein Objekt unter einem bestehenden Schlüssel einzufügen, löst dies eine Exception aus. Im nachfolgenden Listing teilen sich zwei Threads ein solches Dictionary.

Beide Threads arbeiten die gleiche Methode ab und versuchen unabhängig voneinander 999999 Elemente in das Dictionary einzufügen. Dabei prüfen sie vor dem Einfügen, ob bereits ein Objekt unter dem gewünschten Schlüssel existiert. Auf den ersten Blick scheint diese Prüfung auszureichen, um zu verhindern, dass beide Threads an der gleichen Stelle versuchen ein Objekt einzufügen. Da jedoch keine locks genutzt werden, kann es durch präemptives Scheduling zu einer race condition kommen (siehe Zeile 27), sog. *check-then-act* race condition. Um sie zu entdecken ist entsprechende Erfahrung des Programmierers bzw. das Durchspielen aller möglichen Schedules erforderlich.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Threading;
4
5  namespace RacyDictionary
6  {
7  public class Program
8  {
9      private static Dictionary<int, object> _dictionary;
10
11     static void Main(string[] args)
12     {
13         _dictionary = new Dictionary<int, object>();
14         Thread t1 = new Thread(Work);
15         Thread t2 = new Thread(Work);
16         t1.Start();
17         t2.Start();
18         Console.ReadLine();
19     }
20
21     private static void Work()
22     {
23         for (int i = 0; i < 999999; i++)
24         {
25             if (!_dictionary.ContainsKey(i))
26             {
27                 // Potential race condition if scheduler switches between threads at
       this point
28                 try
29                 {
30                     _dictionary.Add(i, new object());
31                 }
32                 catch (Exception e)
33                 {
34                     // Most likely exception: Item already exists.
35                     Console.WriteLine("Exception: {0}", e.Message);
36                 }
37             }
38         }
39     }
40 }
41 }
```

Listing 5 : Dictionary mit race condition.

4.2 Dictionary ohne race condition

Es wird nun eine mögliche Lösung des vorgestellten Dictionary-Problems in P# erarbeitet. Grundgedanke ist dabei, dass das Dictionary zu jedem Zeitpunkt nur genau einem Automaten gehören darf. Ob dies der Fall ist, lässt sich anschließend über die P# static analysis zeigen.

Für die Umsetzung definieren wir zwei Automatentypen und eine Menge von Events, über die die Automaten miteinander kommunizieren. Als Automaten definieren wir:

- *DictionaryAutomaton* - dieser Automatentyp enthält das eigentliche Dictionary sowie die nötige Programmlogik um das Dictionary an Automaten zu versenden und ein aktualisiertes Dictionary zu empfangen.
- *WorkerAutomaton* - dieser Automatentyp enthält die eigentliche Business-Logik und Methoden, um das Dictionary vom DictionaryAutomaton abzufragen und (verändert) zurückzusenden.

Als events definieren wir:

- *ConfigurationEvent* - über dieses event werden grundlegende Konfigurationsinformationen an die Automaten versendet.
- *DictionaryReceivedEvent* - über dieses event wird ein Automat über den Erhalt des Dictionary informiert. Als Payload enthält das event das versendete Dictionary.
- *DictionaryRequestEvent* - über dieses event kann ein Automat den Besitz des Dictionary beantragen. Als Payload enthält das event einen *MachineIdentifier*, der die ID und den Namen des anfragenden Automaten enthält.
- *NextStateEvent* - dieses Event weist den Automaten an, in den nächsten Zustand zu wechseln.

Die beiden Automatentypen können wie folgt visualisiert werden:

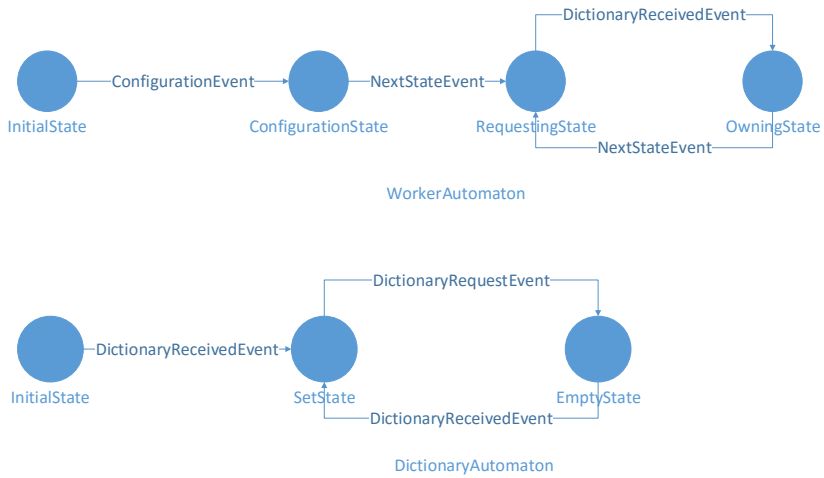

Abbildung 3: WorkerAutomaton und DictionaryAutomaton

Nachfolgend wird die Modellierung dieser Automaten und der Events in P# gezeigt.

4.3 DictionaryAutomaton

```
1  using Microsoft.PSharp;
2  using RaceFreeDictionaryAutomaton.Common;
3  using RaceFreeDictionaryAutomaton.Events;
4  using System;
5  using System.Collections.Generic;
6
7  namespace RaceFreeDictionaryAutomaton.Machines
8  {
9  public class DictionaryAutomaton : Machine
10 {
11     private Dictionary<int, object> _dictionary;
12
13     [Start]
14     [OnEventGotoState(typeof(DictionaryReceivedEvent), typeof(SetState))]
15     private class InitialState : MachineState
16     {
17         protected override void OnEntry()
18         {
19             Console.WriteLine("Dictionary is initializing.");
20             this.Raise(new DictionaryReceivedEvent(new Dictionary<int, object>()));
21         }
22     }
23
24     [DeferEvents(typeof(DictionaryReceivedEvent))]
25     [OnEventGotoState(typeof(DictionaryRequestEvent), typeof(EmptyState))]
26     private class SetState : MachineState
27     {
28         protected override void OnEntry()
29         {
30             DictionaryReceivedEvent received = this.ReceivedEvent as
       DictionaryReceivedEvent;
31             (this.Machine as DictionaryAutomaton)._dictionary = received.Payload as
       Dictionary<int, object>;
32             Console.WriteLine("Dictionary is set.");
33         }
34     }
35
36     [DeferEvents(typeof(DictionaryRequestEvent))]
37     [OnEventGotoState(typeof(DictionaryReceivedEvent), typeof(SetState))]
38     private class EmptyState : MachineState
39     {
40         protected override void OnEntry()
41         {
42             DictionaryRequestEvent request = this.ReceivedEvent as
       DictionaryRequestEvent;
43             MachineIdentifier id = request.Payload as MachineIdentifier;
44             Console.WriteLine("Handing over Dictionary to {0}", id.MachineName);
45             this.Send(id.Machine, new DictionaryReceivedEvent((this.Machine as
       DictionaryAutomaton)._dictionary));
46             (this.Machine as DictionaryAutomaton)._dictionary = null;
47         }
48     }
49 }
50 }
```

Listing 6 : DictionaryAutomaton in P#.

Die Zustände des DictionaryAutomaton werden als Klassen definiert, die von der P#-Frameworkklasse *MachineState* erben. Dabei wird die Klasse *InitialState* als Start-Zustand des Automaten definiert (Zeile 13). Pro Zustand wird über Attribute definiert, bei welchen Events in welchen Folgezustand gewechselt werden soll (Zeilen 14, 25, 37). Events, auf die im jeweiligen Zustand nicht reagiert werden sollen, werden

über das Attribut *DeferEvents* deklariert. Diese events werden in eine Warteschlange gestellt und so bald es geht abgearbeitet, d.h. im nächsten Zustand, der diese events akzeptiert.

4.4 WorkerAutomaton

```
using Microsoft.PSharp;
using RaceFreeDictionaryAutomaton.Common;
using RaceFreeDictionaryAutomaton.Events;
using System;
using System.Collections.Generic;

namespace RaceFreeDictionaryAutomaton.Machines
{
public class WorkerAutomaton : Machine
{
    private MachineId _dictionaryAutomaton;
    private int _counter = 0;
    private string _machineName;

    [Start]
    [OnEventGotoState(typeof(ConfigurationEvent), typeof(ConfigurationState))]
    private class InitialState : MachineState
    {
        protected override void OnEntry()
        {
        }
    }

    [OnEventGotoState(typeof(NextStateEvent), typeof(RequestingState))]
    private class ConfigurationState : MachineState
    {
        protected override void OnEntry()
        {
            Console.WriteLine("Worker is configuring itself...");
            ConfigurationItem item = (this.ReceivedEvent as ConfigurationEvent).
    Payload as ConfigurationItem;
            (this.Machine as WorkerAutomaton)._dictionaryAutomaton = (MachineId)item
    .InitialData;
            (this.Machine as WorkerAutomaton)._machineName = item.MachineName;
            this.Raise(new NextStateEvent(null));
        }
    }

    [OnEventGotoState(typeof(DictionaryReceivedEvent), typeof(OwningState))]
    private class RequestingState : MachineState
    {
        protected override void OnEntry()
        {
            WorkerAutomaton worker = this.Machine as WorkerAutomaton;
            Console.WriteLine("{0} has entered requesting state.", worker.
    _machineName);
            this.Send(worker._dictionaryAutomaton, new DictionaryRequestEvent(new
    MachineIdentifier(worker.Id, worker._machineName)));
        }
    }

    [OnEventGotoState(typeof(NextStateEvent), typeof(RequestingState))]
    private class OwningState : MachineState
    {
        protected override void OnEntry()
        {
            WorkerAutomaton worker = this.Machine as WorkerAutomaton;
            Console.WriteLine("{0} has entered owning state.", worker._machineName);
```

```
55
56              // Set dictionary
57              DictionaryReceivedEvent received = this.ReceivedEvent as
    DictionaryReceivedEvent;
58              Dictionary<int, object> _dictionary = (Dictionary<int, object>)received.
    Payload;
59
60              // Do actual work
61              for (int i = worker._counter; i < 999999; i++)
62              {
63                  if (!_dictionary.ContainsKey(i))
64                  {
65                      // Race condition cannot occur since worker owns dictionary
    exclusively
66                      try
67                      {
68                          _dictionary.Add(i, new object());
69                          Console.WriteLine("{0} has successfully submitted object
    {1}.", worker._machineName, i);
70
71                          // break from loop after successful insert to avoid
    starvation
72                          break;
73                      }
74                      catch (Exception e)
75                      {
76                          Console.WriteLine("Exception: {0}", e.Message);
77                      }
78                  }
79                  worker._counter = i;
80              }
81
82              // give up dictionary
83              Console.WriteLine("{0} is giving up dictionary.", worker._machineName);
84              this.Send(worker._dictionaryAutomaton, new DictionaryReceivedEvent(
    _dictionary));
85
86              // go to requesting state
87              this.Raise(new NextStateEvent(null));
88          }
89      }
90  }
91  }
```

Listing 7 : WorkerAutomaton in P#.

Wie beim DictionaryAutomaton werden die Zustände über Klassen definiert. Es fällt auf, dass der Automat selbst keine Referenz auf das Dictionary hält. Nur im *OwningState* existiert eine solche Referenz, wobei das Dictionary hier über ein DictionaryReceivedEvent empfangen wird (Zeile 57). Der Besitz dieses Dictionary wird in Zeile 84 mit dem Versenden des aktualisierten Dictionary an den DictionaryAutomaton aufgegeben.

Der WorkerAutomaton fügt im OwningState nur ein Objekt ins Dictionary ein, bevor er den Besitz des Dictionary aufgibt. Dies ist nicht zwingend erforderlich, sorgt aber dafür, dass es nicht zur *starvation* anderer WorkerAutomatons kommt. Andernfalls würde der Automat das Dictionary so lange besitzen, bis die for-Schleife abgeschlossen ist.

4.5 Events

Events erben von der P#-Frameworkklasse *Event*. Sie haben stets den gleichen Aufbau, sodass hier stellvertretend für alle unter 4.2 genannten Events das DictionaryReceivedEvent gezeigt wird. Das im Konstruktor übergebene Objekt kann speziellere Typen als *object* haben, es wird im Event jedoch in einer Property vom Typ object gespeichert, sodass beim Zugriff auf die Payload des Events entsprechend gecastet werden muss.

```
using Microsoft.PSharp;
using System.Collections.Generic;

namespace RaceFreeDictionaryAutomaton.Events
{
    public class DictionaryReceivedEvent : Event
    {
        public DictionaryReceivedEvent(Dictionary<int, object> dictionary) : base()
        {
            this.Payload = dictionary;
        }

        public object Payload { get; set; }
    }
}
```

Listing 8 : WorkerAutomaton in P#.

4.6 Aufruf der Automaten

Nach Modellierung der Automaten und der Events müssen diese nun noch durch die P#-Runtime ausgeführt werden. Dies erfolgt in der Program-Klasse des Programms, die folgenden Inhalt hat.

```
using Microsoft.PSharp;
using RaceFreeDictionaryAutomaton.Common;
using RaceFreeDictionaryAutomaton.Events;
using RaceFreeDictionaryAutomaton.Machines;
using System;

namespace RaceFreeDictionaryAutomaton
{
class Program
{
    static void Main(string[] args)
    {
        MachineId dictionaryMachine = PSharpRuntime.CreateMachine(typeof(
    DictionaryAutomaton));

        MachineId worker1 = PSharpRuntime.CreateMachine(typeof(WorkerAutomaton));
        MachineId worker2 = PSharpRuntime.CreateMachine(typeof(WorkerAutomaton));

        PSharpRuntime.SendEvent(worker1, new ConfigurationEvent(new
    ConfigurationItem("Worker 1", dictionaryMachine)));
        PSharpRuntime.SendEvent(worker2, new ConfigurationEvent(new
    ConfigurationItem("Worker 2", dictionaryMachine)));
        Console.ReadLine();
    }
}
}
```

Listing 9 : Program-Klasse zum Starten der Automaten.

Die Maschinen werden über die P#-Runtime erzeugt (Zeilen 13, 15, 16). Als Konfigurationsinformationen werden an die beiden Worker-Automaten jeweils ihr Name und eine Referenz auf den DictionaryAutomaton übergeben (Zeilen 18, 19).

4.7 Statische Analyse des Beispiels

Die gezeigte Implementierung ist lauffähig und erlaubt es beiden Worker-Automaten auf das Dictionary zuzugreifen und es zu verändern. Die statische Analyse des Programms erfolgt über den P#-Compiler, der Teil des Frameworks ist.

```
1  .\PSharpCompiler.exe /s:"D:\CAU\Semester5\PSharp\RaceFreeDictionaryAutomaton\
       RaceFreeDictionaryAutomaton.sln" /p:"RaceFreeDictionaryAutomaton" /t:testing /
       analyze
2  . Parsing
3  . Rewriting
4  . Compiling for Testing
5  ... Writing D:\CAU\Semester5\PSharp\RaceFreeDictionaryAutomaton\
       RaceFreeDictionaryAutomaton\bin\Debug\RaceFreeDictionaryAutomaton.dll
6  ... Linking Microsoft.PSharp.dll
7  ... Linking Microsoft.PSharp.BugFindingRuntime.dll
8  . Analyzing RaceFreeDictionaryAutomaton
9  Error: Potential source for data race detected. Method 'OnEntry' in state '
       EmptyState' of machine 'DictionaryAutomaton' sends payload '(this.Machine as
       DictionaryAutomaton)._dictionary', which contains data from a machine field.
10   ---- Point of sending the payload ----
11    at 'this.Send(id.Machine, new DictionaryReceivedEvent((this.Machine as
       DictionaryAutomaton)._dictionary))' in D:\CAU\Semester5\PSharp\
       RaceFreeDictionaryAutomaton\RaceFreeDictionaryAutomaton\Machines\
       DictionaryAutomaton.cs:line 45
12 ... Static analysis detected '1' error
13 . Done
```

Listing 10 : Ergebnisse der statischen Analyse des P#-Compilers.

Die statische Analyse weist auf eine mögliche race condition im DictionaryAutomaton in Zeile 45 hin (Zeile 11). An dieser Stelle wird das Dictionary an einen anfragen WorkerAutomaton versendet. Da das Dictionary in einem Feld des Automaten gespeichert ist, verstößt dies gegen die 3. Regel der respects-ownership-Analyse (siehe 3.4), da auf das Dictionary weiterhin über *this* zugegriffen werden kann.

Da in Zeile 46 des Automaten die Referenz auf das Dictionary jedoch auf *null* gesetzt wird, ist ein weiterer Zugriff auf das Dictionary an dieser Stelle ausgeschlossen. Erst mit Empfang eines Dictionary wird die Referenz aktualisiert (Zeile 31). Mit Empfang des Dictionary ist jedoch auch der DictionaryAutomaton wieder alleiniger Besitzer des Dictionary, sodass es zu keiner race condition kommen kann.

5 Evaluierung

Das P#-Framework bietet eine im Code gut dokumentierte API und abstrahiert einen großen Teil der Automaten-Definition und der Zustandsübergänge, sodass Automaten und Zustände mit relativ geringem Aufwand definiert werden können.

Der Automaten-basierte Ansatz erzeugt jedoch einen nicht unerheblichen Code-Overhead, wodurch der Quellcode unübersichtlicher wird. Dem steht gegenüber, dass es für einen deterministischen Automaten mit definierter Zustandsübergangsfunktion einfach nachzuvollziehen ist, wie sich ein Automat in einer bestimmten Situation verhält. Dadurch lassen sich Fehlerszenarien einfacher durchspielen und beheben.

Vor der Transformation von bestehenden C#-Code zu P#-Code muss zunächst die bestehende Programmlogik als Zusammenspiel von mehreren Automaten modelliert werden. Je nach Dokumentationslage der Anwendung ist dies recht aufwändig. Da sich mittels der in P# definierbaren Automaten jedoch auch eine Turing-Maschine simulieren lässt, können grundsätzlich alle Turing-berechenbaren Berechnungen auch in P# modelliert werden.[2] Die nebenläufige Ausführung der einzelnen Automaten wird dadurch erreicht, dass jeder Automat durch das Framework in einem eigenen Thread ausgeführt wird.

Bei einem lock-basierten Modell ist der Thread "Besitzer"des gelockten Objekts solange er sich im gelockten Bereich befindet. Andere Threads, die ebenfalls auf das Objekt zugreifen wollen, werden in eine Warteschlange gestellt. Mit Verlassen des gelockten Bereichs gibt der jeweilige Thread den Besitz auf und muss sich für erneuten Zugriff erneut darum bewerben. Die Warteschlange sorgt hierbei über das FIFO-Prinzip für eine gewissen Fairness. In einigen Fällen ist es jedoch möglich, dass der Thread der zuvor bereits das lock gehalten hat, erneut das lock zugeteilt bekommt, obwohl sich andere Threads vor ihm in der Warteschlange befinden.[3] Je nach Modellierung können P#-Programme events tatsächlich im FIFO-Prinzip abarbeiten oder aber die Abarbeitungsreihenfolge über Verzögern oder Ignorieren von events beeinflussen. Im Gegensatz zu locks hat der Programmierer allerdings die volle Kontrolle hierüber und kann eine FIFO-Abarbeitung durch entsprechendes Design sicherstellen.

Im konkreten Beispiel wird deutlich, dass der Besitz eines Objekts im Sinne der Fairness aktiv aufgegeben werden muss, damit es nicht zur starvation von anderen Automaten kommt. Wird der Besitz nicht aufgegeben, behält ein Automat unendlich lang den Zugriff auf ein Objekt. Das würde insbesondere auch im Fehlerfall gelten, wenn also ein Automat eine Berechnung mit einer Exception abbricht. In diesem Fall terminiert die P#-Runtime komplett. Locks hingegen werden im Fehlerfall automatisch wieder freigegeben, sodass ein abstürzender Thread nicht das gesamte Programm zum Absturz bringt. [1]

Die Stärke von P# liegt in der statischen Analyse des Programms. Sie erlaubt - bei korrekter Transformation von bestehendem Code - Rückschlüsse auf die Korrektheit des ursprünglichen C#-Codes. Deligiannis et al. zeigten in einer Fallstudie beim Portieren eines proprietären Microsoft Systems nach P#, dass gleich mehrere race conditions im Code enthalten waren, die trotz intensiver Tests über mehrere Releases hinweg unentdeckt blieben. [2].

Das P#-Framework befindet sich zum aktuellen Zeitpunkt (Dezember 2015) noch in der Entwicklung. Der Code des Frameworks ist offen und kann von GitHub heruntergeladen werden. [4]

[2] Mit der Einschränkung, dass aufgrund des endlichen Hauptspeichers des Rechners kein unendliches Speicherband simuliert werden kann.

[3] Dieses Verhalten erklärt sich dadurch, dass lock-Statements in C# syntaktischer Zucker für dieVerwendung der *Monitor*-Klasse sind. Monitore arbeiten Zugriffsanfragen nur annähernd nach dem FIFO-Prinzip ab, sie sind also nicht fair.[1] [3]

[4] Weitere Informationen dazu unter `http://multicore.doc.ic.ac.uk/tools/PSharp/PLDI15/`

Literatur

[1] Joseph Albahari. Threading in C# - Part 2: Basic Synchronization. `http://www.albahari.com/threading/part2.aspx`, 2011. [Online; Abgerufen am 14.11.2015].

[2] Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Akash Lal, and Paul Thomson. Asynchronous programming, analysis and testing with state machines. Technical report, Imperial College London, GB und Microsoft Research, Indien, 2015.

[3] Joe Duffy. *Concurrent Programming on Windows*. Addison Wesley, 2009.