Seminarvortrag

Analyse nebenläufiger Programme

Von Peer Brauer

Im Rahmen des Seminars Programmierwerkzeuge
bei Prof. Dr. Michael Hanus

Betreuer:
Dr. Frank Huch

Gliederung

1 Einleitung
1.1 Motivation
1.2 Welche Ansätze gibt es?
2 Vorstellung zweier Tools zur Analyse nebenläufiger Programme
2.1 KISS
2.1.1 Die Arbeitsweise von KISS
2.1.2 KISS am Beispiel eines exemplarischen Bluetooth-Treibers
2.1.3 KISS Analyse einer einfachen parallelen Sprache
2.1.4 Programm-Transformation
2.1.5 Vorteile und Schwächen
2.2 ZING
2.2.1 Vorstellung des Tools und seiner Arbeitsweise
2.2.2 ZING Modell eines Treibers
2.2.3 Verwenden des Tools
2.2.4 ZING-Modell eines Treibers
2.2.5 Vorteile und Schwächen
3 Fazit
4 Literaturverzeichnis

1 Einleitung

In diesem Seminarvortrag zum Thema „Analyse Nebenläufiger Programme“ im Rahmen des Seminars Programmierwerkzeuge werde ich erläutern, warum es wichtig ist, dass es ausgereifte Werkzeuge zur Verifikation und Analyse Nebenläufiger Programme gibt, welche verschiedenen Ansätze existieren und wie zwei dieser Tools funktionieren.

1.1 Motivation

Nebenläufige Programme werden in vielen Bereichen eingesetzt, z.B. als Betriebssystem-Treiber, bei parallelen Rechnersystemen, ... , dabei stehen die Entwickler solcher Anwendungen immer vor ähnlichen Problemen. Diese Probleme sind unter anderem

- Erzeugen und Starten von Prozessen
- Koordinieren von Ressourcen
- Synchronisierung des Zugriffs auf Ressourcen
- Abgleich von Daten
- ...

Diese Probleme erfordern dabei nicht selten komplexe und komplizierte Lösungen, bei denen leicht Fehler auftreten können. Dabei ist jedoch nicht leicht ersichtlich, wo ein Fehler auftritt, da nicht nur ein einzelner Prozess betrachtet werden kann, sondern auch seine Wechselwirkung mit anderen Prozessen eines Programms beachtet werden muss. Erst so lässt sich ermitteln, wo der Fehler seinen Ursprung genommen hat und welche Auswirkung er hat.

Genau hier liegt eine der Schwierigkeiten bei der Analyse Nebenläufiger Programme, denn sobald verschiedene Prozesse eines Programms aufeinander angewiesen sind und miteinander interagieren müssen, wird es schwierig, den Ursprung eines Fehlers zu ermitteln. Denn es steht nicht von vornherein fest, dass der Fehler aus einem Stück fehlerhaftem Programmcode resultiert, sondern es besteht auch immer die Möglichkeit, dass die Ursache für den Fehler in der Kommunikation der einzelnen Prozesse untereinander oder aber in der fehlerhaften Verwendung von gemeinsamen Ressourcen liegt.

Um solchen und anderen Fehlern auf die Spur zu kommen, ist es wichtig Werkzeuge zur Verfügung zu haben, die in der Lage sind die komplizierten Wechselwirkungen in parallelen Programmen zu untersuchen und den dafür notwendigen Aufwand in einem überschaubaren Rahmen zu halten.

1.2 Ansätze

In diesem Paper werde ich zwei verschiedene Ansätze vorstellen, die dazu

entwickelt wurden eben diese Voraussetzungen zu erfüllen. Zum einen werde ich zeigen, wie KISS[1] ein paralleles Programm in ein sequentielles Programm überführt, das dann mittels bereits bekannter und getesteter Debugger,in diesem Fall SLAM, für sequentielle Programme debuggt wird. KISS transformiert dann anschließend das Ergebnis des Dubuggingvorgangs wieder derart, dass es für das parallele Programm gültig ist. Wie dies genau funktioniert, werde ich in Kapitel 2.1.2 erläutern.

Der andere Ansatz auf den ich eingehen werde, ist der Ansatz, den das Tool ZING[2] verfolgt. ZING verwendet den Ansatz des Modelchecking, um automatisch aus einem bestehenden parallelen Programm ein korrektes Modell zu erzeugen, das jedoch nur so viele Informationen über das zu testende Programm enthält wie unbedingt notwendig, um damit möglichst performant die Eigenschaften des Programms zu untersuchen.

Neben diesen beiden Ansätzen gibt es auch noch weitere Bemühungen, die Korrektheit von nebenläufigen Programmen zu beweisen, auf die ich in diesem Vortrag jedoch nicht eingehen möchte da dies den Rahmen dieser Arbeit sprengen würde.

2. Vorstellung zweier Tools zur Analyse nebenläufiger Programme

Beide Ansätze und Tools die ich im folgenden vorstellen werde wurden von dem Microsoft Research Team[3] entwickelt, das sein Arbeitsziel wie folgt definiert,

> *Microsoft Research is dedicated to conducting both basic and applied research in computer science and software engineering. Its goals are to enhance the user experience on computing devices, reduce the cost of writing and maintaining software, and invent novel computing technologies. Microsoft Research also collaborates openly with colleges and universities worldwide to broadly advance the field of computer science.*

Zitat 1: Team Vorstellung auf der Microsoft-Homepage

Dementsprechend ist vor allem KISS kein fertiges Produkt, sondern mehr als ein, wenn auch weit entwickelter, Denkanstoß in eine neue Richtung zu sehen.

2. KISS

Das Tool KISS wurde 2004 bei Microsoft Research von Byron Cook, Shaz Qadeer und Dinghao Wu entwickelt um neue Wege bei der Analyse nebenläufiger Programme zu untersuchen. Ihr Ziel war es dabei die Zeit, die es dauert um ein Programm zu analysieren, deutlich zu verringern.

1 http://research.microsoft.com/kiss/
2 http://research.microsoft.com/zing/
3 http://research.microsoft.com/default.aspx

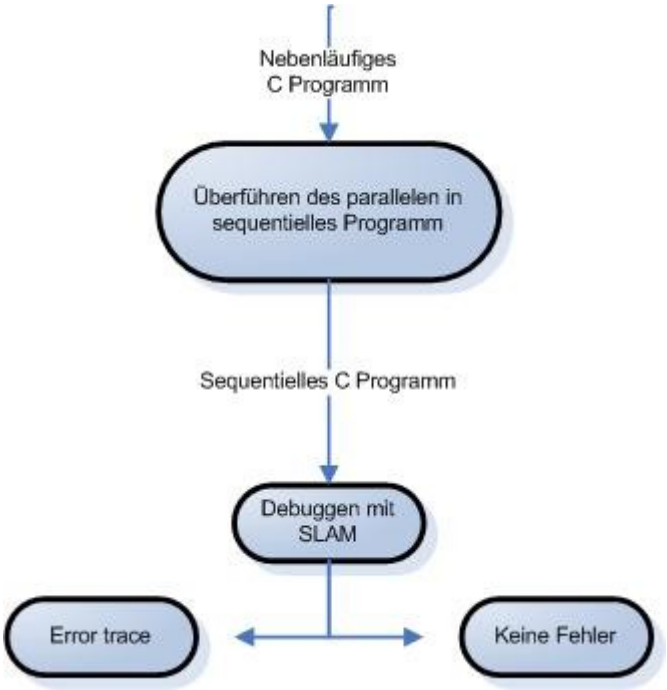
> Since the number of thread interleavings increases exponentially with the number of threads, such analyses have high computational complexity. KISS provides a method to avoid this exponential complexity.

Zitat 2: Zielsetzung auf der Projekt Homepage

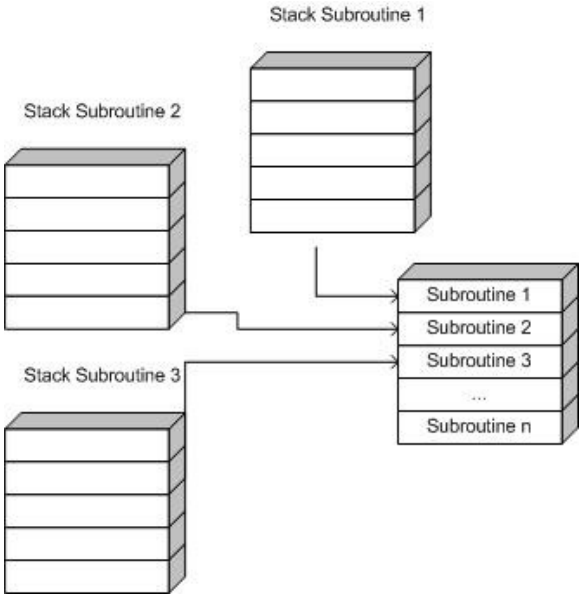
2.1.1 Die Arbeitsweise von KISS

KISS ist, wie weiter oben bereits erwähnt, kein einfacher Model-Checker. Der Ansatz, den die Microsoft-Entwickler bei diesem Tool wählten, ist ein gänzlich anderer. Einen Hinweis auf die Arbeitsweise dieses Tools gibt bereits der Name des Programms, denn KISS steht für **K**eep **i**t **s**equentiell and **s**imple.

KISS überführt ein nebenläufiges Programm P in ein sequentielles Programm P'.



Grafik 1: der Arbeitsablauf von KISS



Grafik 2: Stacktransformation in KISS

Threads werden dabei durch verschiedene Subroutinen emuliert, wobei ein bestimmter Thread t_1 in eine einzige Subroutine s_1 überführt wird. Dabei tritt jedoch ein Problem auf: in einem nebenläufigen Programm hat jeder einzelne Thread seinen eigenen Stack. In einem sequentiellen Programm jedoch müssen sich alle Subroutinen einen einzelnen Stack teilen. Um dieses Problem zu lösen, wurde in KISS ein nicht-

deterministischer Scheduler implementiert, der in der Lage ist, mehrere von einander unabhängige Stacks auf einem einzigen Stack zu emulieren. Dafür teilt der Scheduler den gesamten Stack in mehrere Bereiche, wobei ein Bereich immer den Stack eines bestimmten Threads des nebenläufigen Programms abbildet. Der Scheduler ist nun in der Lage, um das Verhalten eines nebenläufigen Programms mit Hilfe des sequentiellen Programms zu simulieren, jederzeit eine Subroutine des sequentiellen Programms zu stoppen, den Inhalt ihres Teilstacks vom Gesamtstack zu entfernen und mit der Ausführung der Methode deren Teilstack durch den nächsten Bereich des Stacks definiert wird fortzufahren oder eine beliebige andere Subroutine zu beginnen. Für diese Funktionen werden nur wenige globale Variablen zusätzlich benötigt. Dies ist wichtig, denn die Performance des Programms hängt unter anderem von der Anzahl der globalen Variablen und der Zugriffe auf sie ab.

Der Algorithmus für den Scheduler basiert dabei auf zwei verschiedenen Ideen:

- Die erste Idee erlaubt dem Scheduler einen Thread, also die ihn repräsentierende Subroutine, jederzeit, wie bereits oben erwähnt, nicht deterministisch während seiner Ausführungszeit zu beenden. Dazu wird eine neue globale Variable *raise* eingeführt und mit *false* initialisiert. Weiterhin wird jedem Statement in jeder Funktion des Quellcodes ein weiteres Stück Code voran gestellt und angefügt. Das vorangestellte Stück Code ruft entweder mittels dem im nächsten Punkt vorgestellten Multiset eine andere Funktion auf oder gibt *true* zurück, wobei es in diesem Fall einen Return- Wert zurück gibt. Dieser Return-Wert wird dann durch den hinten angefügten Code an die anderen Prozeduren weitergereicht. Der hintere Teil übernimmt es auch, bei Bedarf weitere Teile einer Subroutine auszuführen, so dass eine einzige Subroutine bei Bedarf auch in einem Stück ausgeführt wird und nicht durch andere Prozesse unterbrochen werden kann.
- Die zweite Idee ist ein Scheduler, der die Ausführungsreihenfolge der einzelnen Subroutinen festlegt. Dazu wurde eine weitere globale Variable *ts*, ein Multiset mit fester, aber zufälliger Größe eingeführt, in welchem die auszuführenden Subroutinen und ihre Reihenfolge gespeichert werden. Dieses Multiset ist in seiner Größe beschränkt um bestimmte Eigenschaften nebenläufiger Programme zu erläutern. Wird nun eine Subroutine f aufgerufen wird sie entweder für den Fall, dass *ts* bereits voll ist, sofort ausgeführt oder aber hinten an *ts* angefügt.

Auf diese Art und Weise ist es möglich, komplexe parallele Programme zu simulieren. Die Größe von *ts* bietet dabei die Möglichkeit die Anzahl der „nebeneinander“ ausgeführten Threads zu steuern. Je mehr Nebenläufigkeit zugelassen wird, desto größer muss der Größe von *ts* sein. Allerdings nimmt auch damit die Komplexität des sequentiellen Programms stark zu und seine Ausführungszeit erhöht sich.

Dadurch, dass das sequentielle Programm das nebenläufige Programm abbildet ist es möglich, alle möglichen Änderungen und Kombinationen der einzelnen Threads zu testen um so versteckte Fehler zu ermitteln.

2.1.2 KISS am Beispiel eines exemplarischen Bluetooth-Treibers

Die Funktion von KISS lässt sich am vereinfachten Modell eines Windows-Bluetooth-Treibers erläutern. Ein Windows-Treiber eignet sich aus dem Grund, dass Treiber viele nebenläufige Anfragen und Prozesse tätigen, gut hierfür. Dabei treten am häufigsten zwei Fehlerarten auf.

- Zum einen muss ein zu einem Treiber gehörender Prozess bei jedem Aufruf auf eine Datenstruktur Namens device_extension zugreifen, die die Arbeit synchronisiert und die beim Laden des Treibers initialisiert wird. Das häufigste auftretende Problem ist nun, dass mehrere Prozesse gleichzeitig versuchen, auf die gleichen Felder dieser Datenstruktur zu zugreifen.
- Der zweite Fehler, der häufig bei Treibern auftritt, ist dass bei Beendigung des Treiber durch einen externen Prozess nicht wieder alle durch den Treiber reservierten Ressourcen rechtzeitig oder korrekt freigegeben werden

```
struct DEVICE_EXTENSION {
      int pendingIo;
      bool stoppingFlag;
      bool stoppingEvent;
};

bool stopped;

void main() {
      DEVICE_EXTENSION *e = malloc(sizeof(DEVICE_EXTENSION));
      e->pendingIo = 1;
      e->stoppingFlag = false;
      e->stoppingEvent = false;
      stopped = false;
      async BCSP_PnpStop(e);
      BCSP_PnpAdd(e);
}

void BCSP_PnpAdd(DEVICE_EXTENSION *e) {
      int status;
      status = BCSP_IoIncrement (e);
      if (status == 0) {
            // do work here
            assert !stopped;
      }
      BCSP_IoDecrement(e);
}

void BCSP_PnpStop(DEVICE_EXTENSION *e) {
      e->stoppingFlag = true;
      BCSP_IoDecrement(e);
```

```
      assume e->stoppingEvent; // release allocated resources now
      stopped = true;
}

int BCSP_IoIncrement(DEVICE_EXTENSION *e) {
      if (e->stoppingFlag)
            return -1;
      atomic {
            e->pendingIo = e->pendingIo + 1;
      }
      return 0;
}

void BCSP_IoDecrement(DEVICE_EXTENSION *e) {
      int pendingIo;
      atomic {
            e->pendingIo = e->pendingIo - 1;
            pendingIo = e->pendingIo;
      }
      if (pendingIo == 0)
            e->stoppingEvent = true;
}
```

Quellcode 1: Das Vereinfachte Modell eines Bluetooth-Treibers

Die Datenstruktur Device_Extension dieses Treibers hat 3 verschiedene Felder, die das Verhalten des Treibers steuern. Außerdem gibt es eine globale Variable, die das korrekte Beenden des Treibers steuert.

- PendingIO ist ein Integer-Wert, der die Anzahl der zu Zeit aktiven Threads des Treibers enthält
- stoppingFlag ist ein Boolean der den Wert true annimmt, sobald der Treiber durch einen Thread gestoppt werden soll, ansonsten ist der Wert dieser Variablen false.
- Die boolsche Variable stoppingEvent bildet ein Event ab, das eintritt, sobald der Treiber gestoppt werden soll. stoppingEvent nimmt dann den Wert true an.
- Die globale, boolsche Variable stopped nimmt den Wert true an, sobald ein Thread versucht den Treiber zu stoppen. Dadurch wird verhindert, dass neue Threads gestartet werden können.

Im folgenden wird gezeigt, wie KISS verschiedene Fehler erkennt und wie dies funktioniert.

Ein Beispiel zu Race-Conditions

Das Erkennen einer Race-Condition wird hier am Beispiel der Variable stopping_Flag der Device-Extension gezeigt. Um das Beispiel einfach und überschaubar zu halten wird angenommen, dass das Multiset ts die Größe 0 hat. Dies hat den Effekt, dass die asynchronen Funktionsaufrufe durch synchrone ersetzt werden können.

Wenn man nun den Model-Checker auf das transformierte Programm

anwendet, entdeckt er den folgenden fehlerhaften Pfad. Der Pfad startet mit der Ausführung von Main und der damit verbundenen synchronen Ausführung der Funktion BCSP_PnpStop. In genau dieser Funktion, nachdem stopping_Flag gesetzt wurde, wird raise auf true gesetzt und ein Return-Statement ausgeführt.

Wenn die Ausführung nun zur Main-Funktion zurückkehrt, wird raise wieder auf false gesetzt und BSCP_PnpAdd aufgerufen, die wiederum BSCP_IoIncrement aufruft. Da diese Funktion versucht die Variable stopping_Flag auszulesen, wurde hierdurch eine Race Condition aufgezeigt.

2.1.3 KISS Analyse einer einfachen parallelen Sprache

Die Idee, die hinter KISS steckt lässt sich am einfachsten an einer einfachen parallelen Programmiersprache zeigen, die in der Lage ist einfache nebenläufige Programme zu modellieren. Die Syntax der Sprache ist auf der folgenden Abbildung abgebildet (Grafik 3).

function names	f	::=	$f_0 \mid f_1 \mid \ldots$
integers	i	::=	$\ldots \mid -1 \mid 0 \mid 1 \mid \ldots$
boolean constants	b	::=	$true \mid false$
constants	c	::=	$i \mid b \mid f$
primitives	op	::=	$+ \mid - \mid \times \mid ==$
variables	v	::=	$v_0 \mid v_1 \mid \ldots$
values	u	::=	$v \mid c$
statements	s	::=	$v_0 = c$
		$\mid$	$v_0 = \&v_1$
		$\mid$	$v_0 = *v_1$
		$\mid$	$*v_0 = v_1$
		$\mid$	$v_0 = v_1 \ op \ v_2$
		$\mid$	$assert(v_0)$
		$\mid$	$assume(v_0)$
		$\mid$	$atomic\{s\}$
		$\mid$	$v = v_0()$
		$\mid$	$async \ v_0()$
		$\mid$	$return$
		$\mid$	$s_1; s_2$
		$\mid$	$choice\{s_1 \ [] \ \ldots \ [] \ s_n\}$
		$\mid$	$iter\{s\}$

Grafik 3: Syntax der paralellen Sprache

Diese Sprache unterstützt asynchrone Prozeduraufrufe (async), atomare Statements (atomic) und blockierende Statements (asume). Weiterhin unterstützt diese Sprache auch Pointer-Operationen, auf Arrays jedoch wurde der Einfachheit halber verzichtet, obwohl diese von KISS unterstützt werden.

Die einzelnen Sprachbestandteile haben dabei die folgenden Funktionen:

- *async v_0()*
  Erzeugt einen neuen Thread, dessen Startfunktion der Wert der Variablen v_0 ist. Nach seiner Erzeugung interagiert dieser Thread mit den bereits existierenden anderen Threads.
- *assume (v_0)*
  Blockiert die Ausführung des Threads, bis v_0 den Wert true an nimmt.
- atomic{s}

$$lock\_acquire(l) \stackrel{\text{def}}{=} atomic\{assume(*l == 0); *l = 1;\}$$
$$lock\_release(l) \stackrel{\text{def}}{=} atomic\{*l = 0;\}$$

Grafik 4: einfache Mechanismen zur Synchronisation

  Führt das Statement s atomar aus. Durch die Kombination von atomic und assume können einfache Konstrukte zur Synchronisation der einzelnen Threads erzeugt werden. Siehe Grafik 4.
- choice
  Führt zufällig einen der möglichen Programmzweige aus
- iter{s}
  Führt das übergeben Statement s eine zufällige Anzahl von Malen hintereinander aus.

In dieser Sprache wurde darauf verzichtet boolsche Variablen einzuführen, da sie sich auch mit der Hilfe von Expressions modellieren lassen.

2.1.4 Programm Transformation

Mit Hilfe der hier jetzt vorgestellten Sprache lässt sich erläutern, wie KISS ein nebenläufiges Programm in ein sequentielles überführt. Dabei wird ein nebenläufiges Programm, dass in der Sprache aus 2.1.3 geschrieben wurde in ein Programm der gleichen Sprache, jedoch ohne die nebenläufigen Bestandteile. Die Funktion $[\![.]\!]$, welche das nebenläufige Programm in ein sequentielles überführt, ist in Grafik 5 definiert.

$$
\begin{array}{lcl}
[\![v_0 = c]\!] & = & schedule(); choice\{skip \ [] \ RAISE\}; v_0 = c \\
[\![v_0 = \&v_1]\!] & = & schedule(); choice\{skip \ [] \ RAISE\}; v_0 = \&v_1 \\
[\![v_0 = *v_1]\!] & = & schedule(); choice\{skip \ [] \ RAISE\}; v_0 = *v_1 \\
[\![*v_0 = v_1]\!] & = & schedule(); choice\{skip \ [] \ RAISE\}; *v_0 = v_1 \\
[\![v_0 = v_1 \ op \ v_2]\!] & = & schedule(); choice\{skip \ [] \ RAISE\}; v_0 = v_1 \ op \ v_2 \\
[\![assert(v_0)]\!] & = & schedule(); choice\{skip \ [] \ RAISE\}; assert(v_0) \\
[\![assume(v_0)]\!] & = & schedule(); choice\{skip \ [] \ RAISE\}; assume(v_0) \\
[\![atomic\{s\}]\!] & = & schedule(); choice\{skip \ [] \ RAISE\}; s \\
[\![v = v_0()]\!] & = & schedule(); choice\{skip \ [] \ RAISE\}; v = [\![v_0]\!](); if \ (raise) \ return \\
[\![async \ v_0()]\!] & = & schedule(); choice\{skip \ [] \ RAISE\}; \\
 & & if \ (size() < MAX) \ put(v_0) \\
 & & else \ \{[\![v_0]\!](); raise = false\} \\
[\![return]\!] & = & schedule(); return \\
[\![s_1; s_2]\!] & = & [\![s_1]\!]; [\![s_2]\!] \\
[\![choice\{s_1[]\ldots[]s_n\}]\!] & = & choice\{[\![s_1]\!] \ [] \ \ldots \ [] \ [\![s_n]\!]\} \\
[\![iter\{s\}]\!] & = & iter\{[\![s]\!]\}
\end{array}
$$

Grafik 5: Transformationsfunktion

Jedes Statement s wird hierbei in ein Statement $[\![s]\!]$ ohne asynchrone Funktionsaufrufe überführt. Analog hierzu wird jede vorhandene Funktion f in eine Funktion $[\![f]\!]$ überführt, die wiederum eine Funktionsrumpf $[\![s]\!]$ enthält. Um abzubilden, das Threads jederzeit beendet und fortgesetzt werden können, wird eine neue boolsche Variable mit dem Namen raise eingeführt. Um jetzt einen „Thread“ zu beenden wird diese Variable auf true gesetzt und ein vorher

$$RAISE \stackrel{\text{def}}{=} raise = true; return$$

festgelegtes Return-Statement ausgeführt. Dieses hat dabei die folgende Form . Um nun jederzeit raise nicht deterministisch ausführen zu können, wird an allen wichtigen Stellen des Programmcodes der folgende Code eingefügt. Skip steht dabei für assume(true).

$$choice\{skip \ [] \ RAISE\}$$

Nachdem ein Thread jedoch raise auf true gesetzt hat, können noch eine bestimmte Anzahl an Funktionen auf dem Stack des Threads verbleiben. Um den Thread jedoch zu beenden müssen alle diese Funktionen entfernt werden. Um dies zu gewährleisten, wird nach jedem Funktionsaufruf ein Statement aufgerufen, dass den Wert von raise überprüft und zurück gibt, falls dieser true sein sollte. Dies ist wichtig, da, wie später erklärt wird, für den Fall, dass alle Funktionen vom Stack entfernt wurden der Wert von raise auf false gesetzt wird.

Es reicht jedoch nicht einen Thread zu einem beliebigen Zeitpunkt beenden zu können, man muss außerdem auch in der Lage sein mit der Ausführung eines beliebigen anderen Threads zu jedem Zeitpunkt fortzufahren. Um dieses zu ermöglichen wurde eine globale Variable ts eingeführt. ts speichert alle Threads, die zwar erzeugt aber noch nicht abgearbeitet wurden (siehe 2.1.1). ts

ist ein „Multiset“ von Funktionsnamen, wobei jeder dieser Namen für die Anfangsfunktion eines Threads steht, der noch nicht ausgeführt wurde. Die Größe von ts wird dabei durch MAX begrenzt.

Um ts zu verändern existieren drei verschiedene Mutatoren. Der Mutator get setzt voraus, dass ts nicht leer ist und wählt ein zufälliges Element aus ts aus, entfernt es aus dem Multiset und gibt es zurück. Put hingegen setzt voraus, dass die Menge der Elemente in ts geringer als der Wert von MAX ist und fügt dem Multiset ts einen als Argument übergebenen Funktionsnamen hinzu. Die letzte Funktion schedule hingegen erlaubt es uns eine zufällige Anzahl von Funktionen aus ts auszuführen. Die Anzahl von Funktionen muss dabei natürlich geringer sein als der Wert von MAX.

```
schedule() {
    var f;
    iter {
        if (size() > 0) {
            f = get();
            [[f]]();
            raise = false;
        }
    }
}
```

Die Funktion schedule kapselt dabei die Schedule-Policy des nebenläufigen Programms.

Die Übersetzungsfunktion, die das nebenläufige Programm in ein sequentielles überführt, stellt nun den meisten Statements einen Aufruf dieser Schedule-Funktion, gefolgt von einem nicht deterministischen Aufruf von raise voran.

Wenn eine Funktion f nun asynchron aufgerufen wird überprüft KISS zunächst die momentane Größe des Multisets ts. Ist diese Größe kleiner als der Wert von MAX, wird der Funktionsname dem „Multiset“ hinzugefügt. Ist jedoch der Grenzwert Max bereits erreicht wird die übersetzte Funktion von f $[\![f]\!]$ sofort ausgeführt. MAX bestimmt also den Grad der Nebenläufigkeit, der durch das Tool simuliert werden kann.

Für ein übergebenes, nebenläufiges Programm s wird das übersetzte, durch das externe Verifikationstool zu testende sequentielle Programm dabei wie folgt definiert.

$$Check(s) \stackrel{\text{def}}{=} raise = false;\ ts = \emptyset;\ [\![s]\!];\ schedule();$$

Das Programm Check(s) initialisiert dabei raise und ts mit angemessenen zufälligen Werten, führt dann zunächst $[\![s]\!]$ aus und arbeitet dann die nicht ausgeführten Threads ab. Jeder dabei entstehende Pfad in dem resultierende

sequentiellen Programm repräsentiert dabei eine mögliche Ausführung des nebenläufigen Programms. Es werden jedoch nicht alle Möglichkeiten abgedeckt.

2.1.5 Vorteile und Schwächen

KISS bietet dem Anwender einen neuen und interessanten Ansatz um nebenläufige Programme zu analysieren, die Vorteile die KISS bietet wurden bei einem Testlauf sichtbar, den die Entwickler des Tools mit bereits bestehenden Windows-Treibern durchführten. Als Hardware-Plattform für diesen Test diente ihnen ein 2,2 GHz PC, den Speicher limitierten sie für diesen Versuch auf 800mb und begrenzten die Ausführungszeit für den Check eines jeden Treibers auf 20 Minuten. Die Größe der Treiber variierte dabei zwischen 500 und 9200 Zeilen Quellcode. Von den 18 Windowstreibern, die das Team testete, erwiesen sich dabei 15 als fehlerhaft. Insgesamt meldete das Team dem Microsoft-Qualitätsteam über 30 neu entdeckte Bugs.

Ein Hauptgrund, warum KISS so viele neue Fehler, die bisher unentdeckt geblieben waren, ans Licht brachte, ist seine Flexibilität in den Kernbereichen des Debuggers.

- Der Checker ist einfach erweiterbar. Der Mechanismus des Tools kann einfach angepasst werden, um weitere Fehlerarten zu erkennen.
- Um Fehlalarme zu vermeiden, kann die Laufzeitumgebung des Treibers genau spezifiziert werden. So werden wirklich nur die Fehler gemeldet, die auch wirklich auftreten können.
- Die Fehlerspezifikation ist anpassbar. So wird wirklich nur das als Fehler erkannt, was auch ein Fehler ist.

All diese Anpassungen sind möglich, da KISS bei der Transformation des nebenläufigen Codes in einen sequentiellen mit simulierter Nebenläufigkeit jedem Statement ein Stück Code vor und hinten anstellt, das sein Verhalten beeinflusst.

Ein weiterer Vorteil dieses Tools ist jedoch, dass es die Analyse des sequentiellen Programms nicht selber vornimmt, sondern diesen Teil der Arbeit einem bereits existierendem Analysetool überlässt und nur dessen Trace „zurück übersetzt“. Der Vorteil dieses Ansatzes liegt darin, dass auf ein bereits erprobtes Analysetool zurückgegriffen werden kann, das seine Zuverlässigkeit bereits unter Beweis gestellt hat.

Ein entscheidender Nachteil ist jedoch, dass KISS zwar keine Fehlalarme ausgibt, da es nur das als Fehler erkennt, was vom Anwender als solcher spezifiziert wurde, dass jedoch nicht sicher gestellt ist, dass alle Fehler erkannt wurden.

2.2 ZING

Das Tool ZING, das ich im folgenden kurz als Alternative zu KISS vorstellen möchte, wurde ebenfalls bei Microsoft Research entwickelt. Die Definition des Ziels fiel dabei jedoch anders aus als bei KISS.

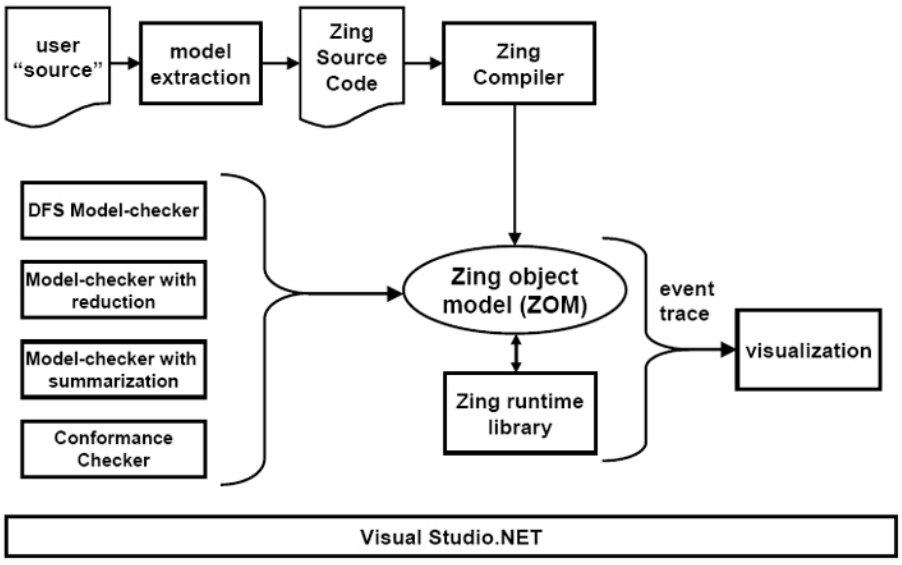
> Our goal is to build a flexible and scalable systematic state space exploration infrastructure for software. This infrastructure includes novel algorithms, and a modular software architecture, to push the frontier on exploring large state spaces of software. We believe that such an infrastructure can be used for verifying and finding bugs in software at various levels: high-level protocol descriptions, work-flow specifications, web services, device drivers, and protocols in the core of the operating system.

Zitat 3: Zielsetzung auf der Projekt Homepage

2.2.1 Vorstellung des Tools und seiner Arbeitsweise

Um das oben erwähnte Ziel zu erreichen entschieden sich die Entwickler von ZING das Tool in 4 verschiedene Module aufzuteilen, die jedes für sich genommen ein klar umrissenes Aufgabenfeld bedienen.

- Das erste Modul ist eine Modellierungssprache für nebenläufige Anwendungssysteme
- Das zweite Modul ist ein Compiler, der das ZING-Modell in ein ausführbares Programm überführt



Grafik 6: Die Architektur von ZING

- Ein ZING-Explorer dient zur Visualisierung der verschiedenen Zustände des Modells
- Das letzte Modul ist ein Generator, der in der Lage ist automatisch aus bereits vorhandenen Programmen, die in einer objektorientierten Programmiersprache wie C++ oder C# geschrieben wurden, ZING-Modelle zu erzeugen

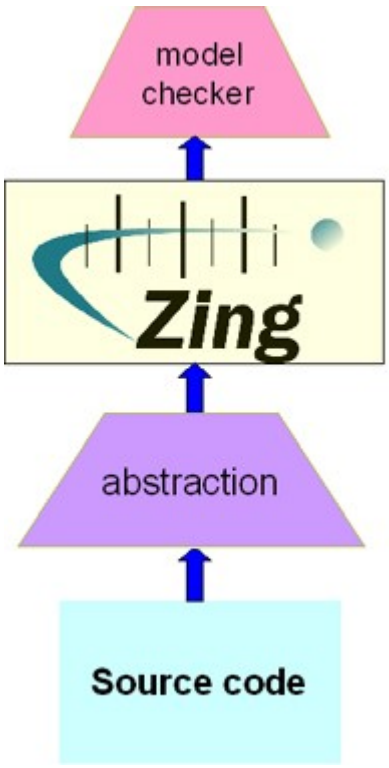
Um mit dieser vierstufigen Architektur ein Programm zu analysieren, sind die folgenden Arbeitsschritte nötig. Zunächst muss ein ZING-Modell erstellt werden, entweder manuell oder mittels des Generators, der in der Lage ist aus jedem Programm das dazu gehörige ZING-Modell zu erstellen. Dieses Modell wird dann mittels des ZING-Compilers in ein von dem ZING-Explorer, auch Viewer genannt, ausführbares Programm überführt. Das Zusammenspiel der einzelnen Komponenten von ZING ist dabei aus Grafik 6 ersichtlich.

Bei der Ausführung des ZING-Modells werden dabei Transitionen zwischen den einzelnen ZING-States erzeugt. Ein ZING State besteht dabei aus den folgenden drei Komponenten:

- **Stacks**
  jeder einzelne Thread in einer ZING-Runtime-Umgebung besitzt seinen eigenen Stack, in dem die lokalen Variablen und Return-Werte hinterlegt werden
- **Globaler Speicher**
  statische Klassenvariablen werden in einem gemeinsamen globalen Speicher abgelegt. Ihre Anzahl und ihre Typen werden während des Kompilierens ermittelt
- **Heap**
  ZING unterstützt die dynamische Zuordnung von Objekten, dynamisch zugeordnete Objekte werden dabei in einem Heap abgelegt, der über ein Array implementiert wurde.

Der Zustandsraum des Modells wird dabei dadurch ermittelt, dass alle möglichen Verzahnungen und Kombinationen der parallelen Ausführung der einzelnen Prozesse untereinander untersucht werden, sowie alle möglichen Ergebnisse nicht-deterministischer Funktionen ermittelt werden. Diese Tätigkeit übernimmt der ZING-Model-Checker, der alternativ zum manuellen Debuggen des Programms durch den Benutzer verwendet werden kann. Der Model-Checker findet dabei Bugs auf die Art und Weise, dass er Zustände identifiziert, die eines der folgenden Kriterien erfüllen.

- Eine explizit angegebene Zuordnung wurde verletzt
- Eine Verklemmung des Modells. Das bedeutet, dass zwar kein weiterer Zustand erreicht werden kann, aber nicht alle Prozesse korrekt beendet wurden
- Die ZING-Runtime Library hat eine Exception geworfen, hierbei kann es sich zum Beispiel um eine Null-Pointer Exception oder ähnliches handeln
- Eine durch das nebenläufige Programm angegebene Exception wurde nicht korrekt abgefangen



Grafik 7: ZING Arbeitsablauf

Um einem Anwender die Arbeit mit ZING zu erleichtern, kann man es als Plugin in Visual Studio 2003 integrieren. Spätere Versionen werden leider nicht unterstützt und eine Standalone-Version gibt es auch nicht.

2.2.3 Verwenden des Tools

An dieser Stelle möchte ich kurz auf die Verwendung des Tools eingehen. Wie bereits weiter oben erwähnt generiert der ZING-Compiler aus einem gegeben ZING-Modell einen Zustandsraum, der alle möglichen Zustände, die angenommen werden können und deren Verknüpfungen untereinander enthält. Dieser Zustandsraum kann dann entweder mit der Hilfe des ZING Model Viewers genauer betrachtet werden oder mit dem ZING Model Tester automatisch eingehend getestet werden.

Da ZING in Visual Studio von Microsoft integriert wird, geschieht die Erstellung eines Modells aus einem ZING-Projekt, dies ist ein Projekt-Typ in Visual Studio, automatisch. Das ZING-Projekt enthält dabei die Quellcodedateien des zu testenden Projekts.

Der „normale“ Weg ein Modell auf Fehler zu überprüfen ist dabei, den Model-Checker auf einem geöffneten ZING-Projekt auszuführen. Dies geschieht, in dem man den entsprechenden Menüpunkt aus dem Tools-Menü von Visual-Studio auswählt. Wird nun durch den Model-Checker ein Fehler entdeckt, öffnet sich ein Extrafenster, das den Fehler Trace zeigt. In diesem Fenster sind alle Informationen zu dem Fehler abrufbar, zum Beispiel Daten zu allen Prozessen und nicht deterministischen Funktionen, die zu diesem Fehler geführt haben.

Es gibt jedoch auch noch zwei andere Wege um ein Ausführungsprotokoll zu erstellen. Der erste Weg ist der Befehl „Generate random trace“. Dieser Befehl führt ein Tool aus, das zufällige die auszuführenden Prozesse auswählt, die ausgeführt werden. Für das Ergebnis hierbei gibt es drei verschiedene Möglichkeiten. Das Programm kann im günstigsten Fall, es ist auf diesem Weg fehlerfrei, terminieren, es kann eine Verklemmung auftreten oder die maximale Anzahl an Schritten für die Ausführung, dieser Wert kann individuell bestimmt werden, wurde erreicht.

Der andere Weg ein Modell auf Fehler zu überprüfen, ist es dieses Modell manuell mittels des Model-Viewers zu betrachten. Das Modell wird hier als eine Menge aus Knoten und Kanten dargestellt. Jeder Knoten steht hierbei für einen Zustand in dem sich das Programm befinden kann. Sollten während der Zustandsänderung von einem Zustand zum anderen wichtige Änderungen eingetreten sein, so können diese betrachtet werden, in dem man den Zustand vergrößert. Es ist weiterhin auch möglich in speziellen Traces zwei Modelle zeitlich genau gegenüber zustellen und so ihre gegenseitig Kommunikation zu überwachen und gegebenenfalls zu debuggen.

Um sich innerhalb der Zeitline des Traces vor und zurück zu bewegen kann man die F7 und F8 Tasten verwenden. Mittels Rechtsklicks auf einen bestimmten Knoten kann eine Detailinformation für eben diesen Knoten abgerufen werden. Diese gibt Aufschluss über die aktuellen Prozesse, Stacks, den Heap und die globalen Variablen.

Um ein Modell automatisch zu Verifizieren, muss man den Befehl verify auswählen. Der Modell-Checker testet nun das Modell bis zum ersten Fehler, den er findet und zeigt diesem samt aller wichtigen Informationen in einem extra Fenster an. Der Nutzer kann jetzt nacheinander von Fehler zu Fehler springen und diese einzeln beheben.

2.2.4 ZING Modell eines Treibers

```
static void PNP_Stop (DEVICE_OBJECT BtDevice){
      DEVICE_EXTENSION deviceExtension = BtDevice.deviceExtension;
      deviceExtension.driverStoppingFlag = true;
      IoDecrement(deviceExtension);
      KeWaitForStoppingEvent(deviceExtension);
      deviceExtension.stopped = true;
}

static void Dispatch(DEVICE_OBJECT BtDevice){
      DEVICE_EXTENSION deviceExtension = BtDevice.deviceExtension;
      int status;
      status = IoIncrement (deviceExtension);
      if(status > 0){
            // do work here
            assert(!deviceExtension.stopped);
      }
      BCSP_IoDecrement(deviceExtension);
}
```

```
static int IoIncrement(DEVICE_EXTENSION e){
      int status;
      bool driverStopping = e.driverStoppingFlag;
      if(driverStopping == true) status = -1;
      else{
            InterlockedIncrementPendingIo(e);
            status = 1;
      }
      return(status);
}

static void IoDecrement(DEVICE_EXTENSION e){
      int pendingIo;
      pendingIo = InterlockedDecrementPendingIo(e);
      if(pendingIo == 0) KeSetEventStoppingEvent(e);
}
```

Quellcode 2: ZING Modell eines Treibers

Der hier vorgestellte Treiber ähnelt dem Treiber, der in 2.1.2 vorgestellt wurde. Die Synchronisation der Threads untereinander läuft dabei über drei Variablen ab, ein Flag vom Typ Bool namens driverStoppingEvent, das mit false initialisiert wird, ein stoppingEvent vom Typ Event und die Zählvariable pendingIO, die mit 1 initialisiert wird.

Der Steuerungsablauf des Treibers ist dabei wie folgt. Der Steuerungsthread testet zunächst, ob der Treiber per driverStoppingEvent gestoppt werden soll. Ist dies der Fall, wird keine weiteren Operation ausgeführt. Ist dies jedoch nicht der Fall wird der Wert von pendingIO um 1 erhöht und die anstehenden IO-Operationen werden ausgeführt. Nach dieser Operation wird pendingIO wieder dekrementiert. Sollte der Wert dabei 0 erreichen wird das driverStoppingFlag gesetzt, das stoppingEvent ausgelöst und der Treiber beendet.

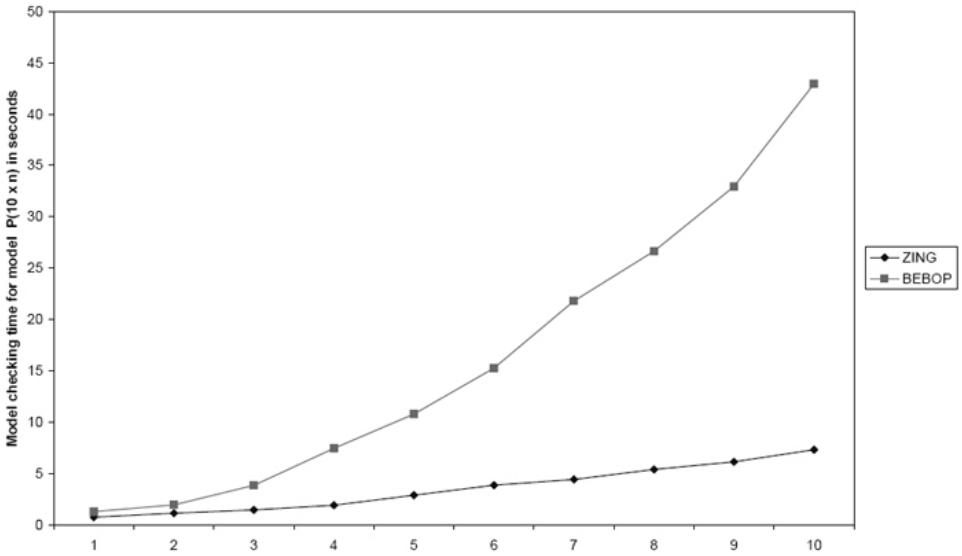
Ein Fehler kann jetzt auftreten, wenn während der Steuerungsthread das driverStoppingEvent bereits mit dem Wert false ausgelesen hat , den Wert von pendingIO aber noch nicht erhöht hat ein Signal zum Stoppen des Treibers eintrifft und pendingIO bereits dadurch decrementiert wurde.

Der ZING-Explorer erkennt diesen Fehler durch die Analyse der Zusammenhänge der einzelnen Komponenten des Treibers und der ZING-Browser erlaubt das schrittweise Debuggen des Fehlers und zeigt den gesamten Trace auf, der zu diesem Fehler führte.

2.2.5 Vorteile und Schwächen von ZING

Ein klarer Vorteil von ZING ist die strikte Trennung des Modells von seiner Spezifikation, egal ob es aus einem fertigen Programm oder aus einer ZING-Spezifikation erzeugt wurde. Dies ist wichtig, da es das Ziel der Entwickler war ein ZING-Modell möglichst stark zu abstrahieren, um so die Anzahl der für das Modell notwendigen States zu verringern.

Die Verringerung der States hat den Vorteil, dass zum einen die Größe des Modells begrenzt wird und das da durch zum anderen auch die Laufzeit des



Grafik 8: Laufzeit-Vergleich am parametrisierten sequentiellen Modell Bepop contra ZING

Model-Checkers verringert wird. Deutlich wird der Vorteil dieser Lösung in der Grafik 8. Hier wurden die Zeit, die ZING benötigte, um ein parametrisiertes Statechart zu analysieren, der Zeit gegenübergestellt, die Bebob, ein anderer Model-Checker, für die gleiche Tätigkeit benötigte. Getestet wurde dabei das Statechart eines Programms P(10*n), wobei n die Anzahl der globalen booleschen Variablen war. Deutlich ersichtlich ist der Laufzeitvorteil, den ZING durch sein abstrakteres Modell erlangt. Hier wächst die Ausführungszeit der Analyse linear zu Größe des Programms.

Ein Nachteil des Programms ist, dass es nicht immer terminiert, wenn in einem Modell eine zyklische Struktur oder andere ungünstige Konstruktionen auftauchen. ZING erlaubt es dem Nutzer dennoch das Modell bis zu diesem Punkt ausschnittsweise zu betrachten.

Für die Zukunft haben sich die Entwickler von ZING vorgenommen nicht deterministische Fälle nicht mehr durch Aufspalten in mehrere Fälle zu behandeln, sonder als symbolische Werte zu behandeln.

3. Fazit

KISS und ZING sind zwei Programme mit einem unterschiedlichen Ansatz, die beide das gleiche Aufgabenfeld bedienen. Während KISS versucht einen neuen Weg in der Analyse nebenläufiger Programme zu beschreiten, geht ZING eher den bereits von anderen Tools eingeschlagenen Weg des Model-Checking, versucht dies aber durch neu Ideen und Methoden zu beschleunigen.

KISS ist jedoch kein Tool, das eigenständig arbeitet. Es ist immer auf ein weiteres Tool angewiesen, das die Aufgabe übernimmt das von KISS erzeugte sequentielle Modell zu überprüfen und einen Auswertungsbericht dazu zu erstellen. ZING hingegen ist, sieht man einmal von der Integration in Microsoft Visual Studio 2003 und seiner Abhängigkeit zu genau dieser Version ab, ein Tool, das die gesamte Aufgabe der Analyse quasi aus einer Hand bietet und nicht auf Zusatztools angewiesen ist.

Insgesamt wirkt ZING wesentlich ausgereifter und für den praktischen Einsatz geeigneter als KISS, was wohl nicht zuletzt auch von der wesentlich weiteren Entwicklungsstufe und dem größeren Entwicklungsteam herrührt. Nichts desto trotz bieten beide Tools interessante Ansätze, die es lohnen sich mit ihnen auseinander zusetzten. Negativ fällt jedoch auf, dass die Informationen an vielen Stellen lückenhaft sind, da auf den Seiten zu den Tools Dokumente und Abbildungen fehlen, die jedoch in anderen Quellen als Erklärung für manche Effekte angegeben sind, so dass man versuchen muss die verbleibenden Informationen aus den vorhandenen Quellen zusammen zu suchen.

4. Literaturverzeichnis

Informationen zu KISS

- S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. Microsoft Research Technical Report 2004-70, July, 2004.
- S. Qadeer and D. Wu. KISS: Keep it Simple and Sequential. Accepted at the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2004.

Informationen zu ZING

- Projekt Homepage bei Microsoft [http://research.microsoft.com/zing/]
- ZING language specification Overview, Examples, and reference manual for the ZING language [http://research.microsoft.com/zing/]
- ZING user manual User guide for getting started with the ZING model checker [http://research.microsoft.com/zing/]
- T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie, ZING: A Model Checker for Concurrent Software, MSR Technical Report: MSR-TR-2004-10.
- T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie, ZING: Exploiting Program Structure for Model Checking Concurrent Software, in CONCUR 2004.
- ZING: Exploiting Program Structure for Model Checking Concurrent Software presented at CONCUR 2004
- ZING: A Systematic State Explorer for Concurrent Software