Software Plattform Embedded Systems 2020

GEFÖRDERT VOM

Bundesministerium
für Bildung
und Forschung

- Deliverable 5.3.D: Werkzeugprototyp für minimalinvasives Debugging -

Version: 1.0

Projektbezeichnung	SPES 2020	
Verantwortlich	Hartmut Lackner	
QS-Verantwortlich		
Erstellt am	01.01.2012	
Zuletzt geändert	29.01.2012 17:56	
Freigabestatus		Vertraulich für Partner:
		Projektöffentlich
	X	Öffentlich
Bearbeitungszustand		in Bearbeitung
		vorgelegt
	X	fertig gestellt

Weitere Produktinformationen

Erzeugung	Hartmut Lackner
Mitwirkend	Julian Godesa

Änderungsverzeichnis

Änderung			Geänderte Kapitel	Beschreibung der Änderung	Autor	Zustand
Nr.	Datum	Version				
1	29.01.12	1.0	Alle	Produkterstellung	Hartmut Lackner & Julian Godesa	Fertig gestellt

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **2**
2.1 Debugging 2
2.1.1 Fehlerklassifikation von Concurrency Bugs 4
2.1.2 Ansätze und Werkzeuge für Concurrency Bugs 4
2.2 Mustererkennung am Beispiel vom Lost-Update 5
2.3 Modellierung des Lost-Update Detektors 5

3 Modellierung eines allgemeinen Detektors **8**
3.1 Anforderungen an die Modellierung 8
3.2 UML Vorlagen für die Modellierung der Detektors 9
3.3 Modellierung am Beispiel des Lost-Update-Detektors 10
3.4 Abbildung der Modellelemente auf Code-Fragmente 12
3.4.1 Das Zusammenspiel der einzelnen Komponenten 12
3.4.2 Der Programmablauf des Detektors 13
3.4.3 Extrahieren von Adresskandidaten 14
3.4.4 Instrumentierung bei einem Lesezugriff 16
3.4.5 Instrumentierung bei einem Schreibzugriff 17

4 Diskussion **19**
4.1 Vergleich des Detektors mit Helgrind in Bezug auf das Lost-Update Beispiel 19
4.1.1 Szenario A: Kein Data Race 20
4.1.2 Szenario B: Einfaches Data Race 20
4.1.3 Szenario C: Komplexes Data Race 21
4.2 Herausforderung Nichtdeterminismus 21

5 Zusammenfassung & Ausblick **23**

A Die Verwendung des Detektors **25**
A.1 Voraussetzungen 25
A.2 Installation 26
A.3 Beispielhafte Anwendung 26

A.3.1 Beschreibung der Parameter 26
A.3.2 silent mode - Ausführung ohne Breakpoint 27
A.3.3 debug mode - Ausführung mit Breakpoint 28

B Beispielprogramme **30**
B.1 dataRace_free 30
B.2 dataRace_simple 32
B.3 dataRace_complex 33

Kapitel 1

Einleitung

Der anhaltenden Trend hin zu Multicore-Computern, stellt die Softwareentwicklung als ganzes vor neue Herausforderungen. Neben den schon existierenden Problemen bei der Entwicklung von komplexen Softwaresystemen kommt nun das Problem des Nichtdeterminismus voll zu Tragen. Zum einen kann Software nicht mehr nur nebenläufig sondern sogar echt parallel auf mehreren Prozessorkernen ausgeführt werden, was die Performance enorm steigern kann. Zum anderen ist aber die Kontrolle der zeitlichen Ausführung durch Optimierungen wie die Out-Of-Order Execution und preemptives Multitasking des Betriebssystems nicht mehr in der Hand des Programmierers.

Klassische Debugging-Methoden setzten eine Wiederholbarkeit der Tests voraus. Da sich durch die oben beschriebenen Gründe einzelne Programmabläufe voneinander unterscheiden, greifen normale Debugging-Methoden nicht mehr. Durch die Verwendung von Threads zur Parallelisierung wurde neben dem Nichtdeterminismus noch zusätzlich eine neue Klasse von Defekten eingeführt, im Folgenden *Concurrency Bugs* genannt. Sie hängen eng mit der zeitlichen Reihenfolge der ausgeführten Threads zusammen und sind nur sehr schwer aufzuspüren.

In diesem Deliverable wird eine Methode zur Modellierung von Zugriffsmustern auf der Ebene von Speicherzellen vorgestellt. Die Modellierung mit gängigen Formalismen erlaubt eine abstraktere Perspektive auf die Zugriffsmuster, als konkrete Implementierungen sie bieten. Für ein mit der Modellierung spezifiziertes Muster wird prototypisch ein Musterdetektor implementiert, der die Machbarkeit des Ansatzes demonstriert.

In Deliverable 5.3.B wurden bereits die Grundlagen und die Problemstellung ausführlich behandelt. Diese Dokument wiederholt deshalb nur vereinzelt Teile des Vorgängerdokumentes um den Lesefluss nicht zu hemmen. Das Deliverable 5.3.B wird dennoch zum Verständnis dieses Dokumentes vorausgesetzt.

Kapitel 2

Grundlagen

2.1 Debugging

Seit der Softwarekrise und den damit verbundenen NATO Konferenzen 1968 und 1969 wurden viele Prozesse und Vorgehensmodelle entwickelt, die die hohe Komplexität und damit verbundene Fehleranfälligkeit von Software handhabbar machen sollten. Aus dem Wasserfallmodell entwickelte sich das V-Modell in dem das Erstellen sowie die Ausführung von Tests eine wichtige Rolle spielt. Ein Großteil der Defekte wird heute durch Codereviews, Modultests, Integrations- und Abnahmetests gefunden, aber eben nicht alle.

In einer im deutschsprachigem Raum durchgeführten Softwaretestumfrage im Jahre 2011 hat sich gezeigt, dass 27 % der ausgelieferten Software noch immer einige schwerwiegende Fehler enthielt [1]. Unabhängig davon, ob nun eine agile Entwicklungsmethode oder nach dem klassischem V-Modell entwickelt wurde. Das trotz dieser hohen Fehlerrate das Einsatzgebiet von Software immer weiter voran getrieben wird, ist in der technischen Entwicklung wohl einmalig.

Gerade in der agilen Softwareentwicklung spielt das Testen von Software auf korrektes Verhalten eine besondere Rolle. In dem „Manifesto for Agile Software Development“ [2] wird als letzter Punkt die Reaktion auf Veränderung hervorgehoben. Erst eine gute Testsuite erlaubt einem Programmierteam auf Veränderungen schnell zu reagieren [3].

Die Relevanz von Softwaretests nimmt also stetig zu. Dennoch gilt der folgende Satz von Edsger W. Dijkstra: „Program testing can be used to show the presence of bugs, but never to show their absence!“ [4]. Es ist also weiterhin wichtig Werkzeuge zu entwickeln, die die bestehende Software (automatisch) untersuchen und dem Entwicklerteam dabei helfen die Anzahl der Fehler so gering wie möglich zu halten.

Wenn ein fehlerhaftes Verhalten detektiert wurde, so muss der verantwortliche Fehler gefunden und bereinigt werden. Dies wird als Debugging bezeichnet. Es existieren eine Reihe von Debugging-Methoden, welche hier der Übersicht halber kurz wiederholt

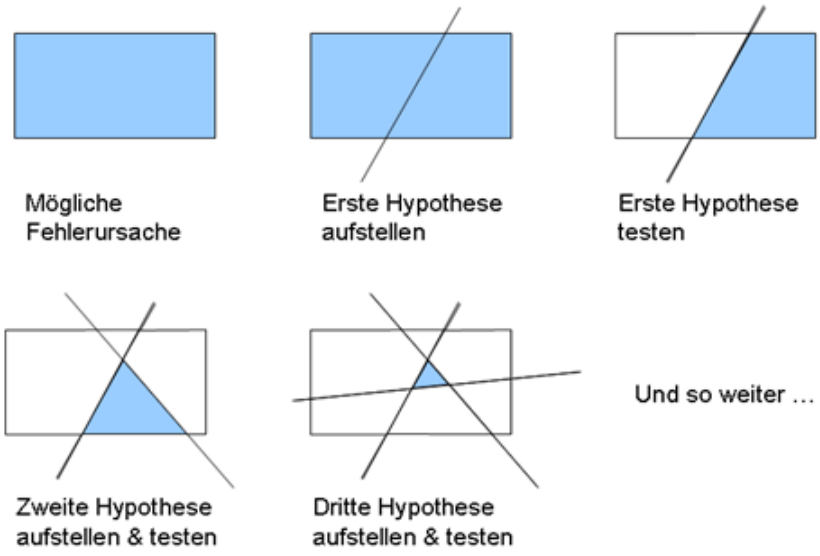

Abbildung 2.1: Verfeinern von Hypothesen im idealen Debuggingprozess.

werden (mehr Informationen sind dazu im Deliverable 5.3.B zu finden):

Printf-Debugging steht für intuitives Debugging. Oft hat ein Entwickler einen Verdacht, welche Zeile Programmcode für das fehlerhaftes Verhalten verantwortlich ist. Vielleicht wurde ein bestimmter Pfad nicht ausgeführt oder eine Variable wurde auf einen unerlaubten Wert gesetzt. Um diese Vermutung bestätigen zu können wird oft auf das Ausschreiben von Hinweisen (daher printf-Debugging) zurückgegriffen. In den meisten Fällen artet aber diese Methode in wildes Probieren aus und ist nicht sonderlich effektiv.

Der ideale Debugging-Prozess versucht wildem Probieren vorzubeugen und auf geordnete Art und Weise den Fehler zu lokalisieren und zu beheben. Im Gegensatz zum Printf-Debugging werden hier Hypothesen bewusst aufgestellt, die durch ein Experiment bestätigt und dann in einer neuen Iteration verfeinert werden. Durch das Verfeinern der aufgestellten Hypothesen wird so der Suchraum in dem sich der Fehler (wahrscheinlich) befindet strukturiert und nachvollziehbar verkleinert. Dieser Prozess ist in Abbildung 2.1 dargestellt.

Delta-Debugging ist eine relativ neue Debugging-Methode [5]. Hier werden Faktoren, die womöglich zum fehlerhaften Verhalten beigetragen haben, meist automatisch bzw. mit Unterstützung von Tools schrittweise eliminiert. Es ist im Grunde die automatisierte Version vom idealen Debugging-Prozess.

Alle drei genannten Methoden haben eine gemeinsame Voraussetzung: Das fehlerhafte Verhalten muss reproduzierbar sein. Diese Voraussetzung ist für regelmäßig nicht aber

für unregelmäßig auftretende Defekte gegeben. Gerade das Printf-Debugging führt hierbei außerdem zu dem „Probe-Effect“. Es ist das grundsätzliche Problem, dass das zu beobachtende System durch die Beobachtung selbst beeinflusst wird. Werner von Heisenberg stellte schon 1927 fest, dass es keine neutrale Beobachtung gibt:

> [...], dass jeglicher Vorgang aufgrund der ontologischen Struktur des verwendeten Beobachtungsapparates den Gegenstand, und damit das Resultat der Beobachtung, verändert. Es gibt keine neutrale Beobachtung, die den Beobachtungsgegenstand unverändert hinterlässt.

2.1.1 Fehlerklassifikation von Concurrency Bugs

Software-Defekte werden nach Gray [6] in permanente und unregelmäßig auftretende Defekte eingeteilt. In Anlehnung an die Heisebergsche Unschärferelation tauft er die letztere Heisenbugs. Durch den Einsatz von Multicore Systemen, in denen Software jetzt nebenläufig oder sogar parallel ausgeführt werden kann, stellt das Auffinden von Heisenbugs eine besondere Herausforderung dar.

Im Dokument Deliverable 5.3.B wurden die Klasse der Heisenbugs noch weiter in folgende Kategorien aufgesplittet, die in diesem Dokument in ihrer Gesamtheit als „Concurrency Bugs“ bezeichnet werden:

Data Race Ein Data Race existiert, wenn zwei nicht synchronisierte Zugriffe auf eine geteilte Ressource nebenläufig stattfinden, wobei mindestens einer der Zugriffe schreibend ist.

Atomicity Violation Eine Atomicity Violation existiert, wenn für das korrekte Arbeiten eine Menge von Anweisungen atomar ausgeführt werden muss, diese aber durch eine andere Anweisung gestört wird.

Deadlock Eine Menge von Threads befindet sich in einem Deadlock-Zustand, wenn jeder Thread auf die Freigabe einer Ressource wartet, die nur von einem anderen Thread aus der Menge freigegeben werden kann.

Neben diesen Defekten wird die Performance von Software durch eine fehlerhafte oder auch nur ungünstige Threadprogrammierung negativ beeinflusst. So kann z.B. häufiges Zugreifen auf eine gemeinsame Ressource, welche korrekt durch den Einsatz von Locking-Mechanismen gesichert ist, die Ausführungszeit enorm verlängern, da hier die Parallelität serialisiert wurde.

2.1.2 Ansätze und Werkzeuge für Concurrency Bugs

In der folgenden Tabelle 2.1.2 listen wir für jede hier genannte Klasse der Concurrency Bugs ein gängiges Werkzeug und geben eine kurze Beschreibung:

Name	Concurrency Bug	Beschreibung
Helgrind	Deadlocks	Helgrind [7] ist ein Detektor, der ein in der Valgrind VM ausgeführtes Programm zur Laufzeit untersucht. Hierbei werden die Aquisitation und das Freigeben der Locks in einem Lock-Order-Graph überwacht.
Lockset	Datarace	Der Lockset Algorithmus überprüft, ob für geteilte Ressourcen ein kohärentes Locking verwendet wurde. Das Werkzeug ThreadSanitizer [8] von Google nutzt unter anderem diesen Algorithmus.
CTrigger	Atomicity Violation	CTrigger [9] instrumentiert das SUT mit dem Ziel, die Software vom Programmierer nicht vorgesehene Interleavings ausführen zu lassen.

2.2 Mustererkennung am Beispiel vom Lost-Update

Die Arbeitsweise von Helgrind und ThreadSanitizer kann zusammenfassend auch als Überwachung von Zugriffen auf Ressourcen beschrieben werden. Im Deliverable 5.3.B wurde ausgehend von dieser Arbeitsweise ein möglicher Detektor beschrieben, der ein Lost-Update detektiert. Im Folgenden beschreiben wir exemplarisch zwei Zugriffsmuster, die ein solches Lost-Update beschreiben, um in einem nächsten Schritt das Vorgehen generalisieren zu können. Durch eine Generalisierung von Zugriffsmustern ist es möglich verschiedene Detektoren zu erzeugen, die dann das SUT untersuchen. Als durchgehendes Beispiel wird als Zugriffsmuster ein Lost-Update verwendet. Ein Lost-Update ist eine spezielle Form eines Data Races, und gehört damit zur Klasse der Concurrency Bugs.

Definition 1. *Ein Lost-Update hat stattgefunden, wenn mindestens zwei Threads den Wert einer Variable nebenläufig erhöhen, also den alten Wert lesen und dann verändert zurückschreiben, und einer der Threads das Update des anderen überschreibt.*

Durch den Einsatz von Sequenzdiagrammen ist die Reihenfolge, die zu einem unzulässigem Zugriff führt, klar ersichtlich. Die Arbeitsweise eines dazugehörigen Detektors lässt sich durch diese Beschreibung aber nicht ableiten.

2.3 Modellierung des Lost-Update Detektors

Das Erkennen der Schreib- und Lesemuster lässt sich als Zustandsmaschine beschreiben. Bei jedem Zugriff auf eine geteilte Speicherressource wird die Zustandsmaschine aktualisiert. Die Zustandsmaschine speichert die aktuell erlaubten Zugriffe auf einen Speicherbereich. Wenn ein Thread einen nicht erlaubten Zugriff tätigt, wird dies als potenzielles Lost-Update angesehen. Die Speicherstruktur ist eine Liste von Tupeln (t, x) mit t als Threadid und x als Speicheradresse.

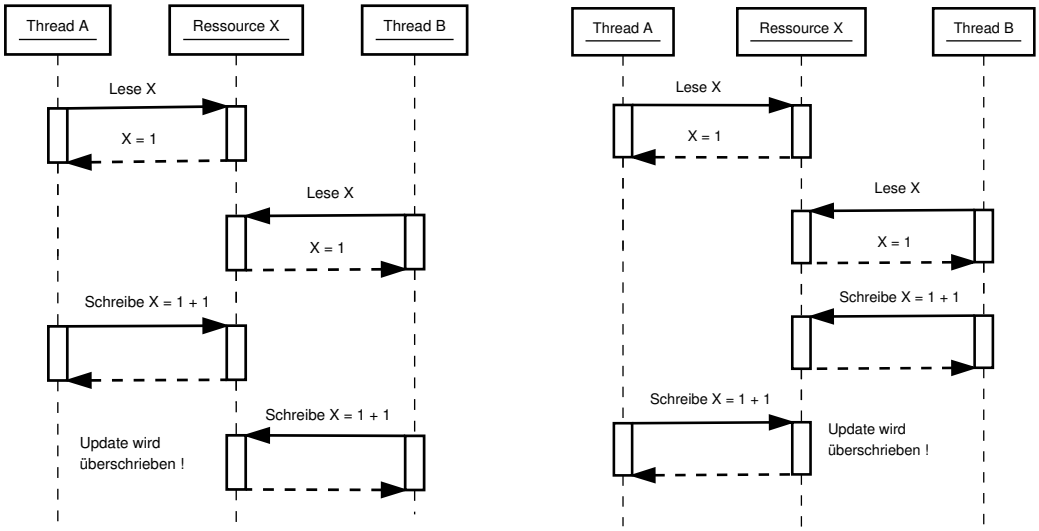

Abbildung 2.2: Schreib- und Lese Muster, die als Lost-Update gelten

Die Verwendung einer Speicherstruktur erlaubt es, die Vergangenheit bzw. Hinweise auf den aktuellen Zustand zu speichern.

Folgende Zustände kann ein Tupel besitzen:

$(t, x) = r$ gibt an, dass der letzte Zugriff von Thread t auf die Speicheradresse x lesend war.

$(t, x) = w$ gibt an, dass der letzte Zugriff von Thread t auf die Speicheradresse x schreibend war.

$(t, x) = lost$ gibt an, dass der nächste schreibende Zugriff von Thread t auf die Speicheradresse x ein vorheriges Update überschreiben wird. Es wird also nicht gespeichert, was in der Vergangenheit geschehen ist, sondern was in Zukunft passieren kann.

In Abbildung 2.3 ist der Algorithmus zur Erkennung eines Lost-Updates dargestellt. Der Algorithmus wird bei jedem Schreib- und Lesezugriff eines Threads t auf eine Speicheradresse ausgeführt. Der Algorithmus aktualisiert dabei die Tupel (t, x) wie oben beschrieben.

Zur Erklärung der Zustandsmaschine wird im Folgenden das linke der zwei Zugriffsmuster in Abbildung 2.2 exemplarisch ausgeführt und das Lost-Update erkannt.

1. Der erste Zugriff von Thread $Thread_A$ auf die Speicheradresse $Ressource_X$ ist lesender Natur. Also wird das Tupel $(Thread_A, Ressource_X)$ auf r gesetzt.

2. Der erste Zugriff von Thread $Thread_B$ liest die Speicheradresse $Ressource_X$. Das Tupel $(Thread_B, Ressource_X)$ wird ebenfalls auf r gesetzt.

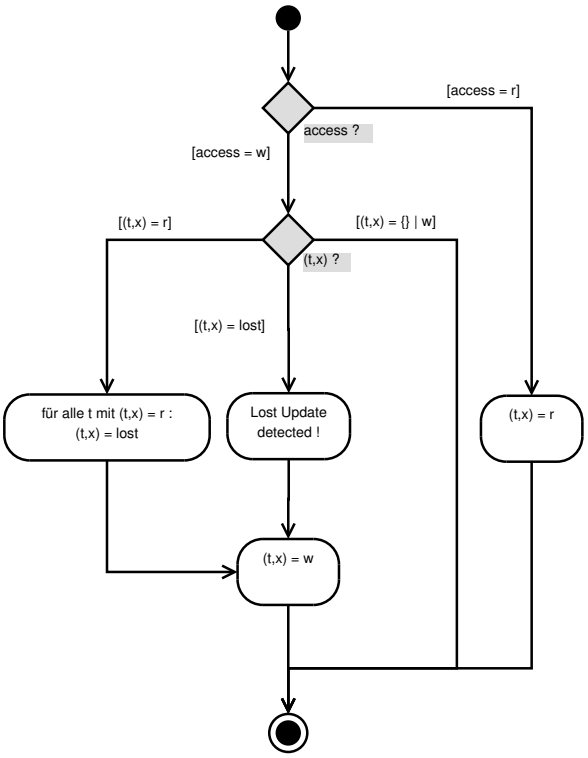

Abbildung 2.3: Die Aktualisierung der Zustände und das Erkennen eines Lost-Updates.

3. Nun erfolgt ein schreibender Zugriff auf die Speicheradresse $Ressource_X$ von $Thread_A$ und das Tupel $(Thread_A, Ressource_X)$ wird auf w gesetzt. Zusätzlich werden alle vorherigen lesenden Zugriffe auf die Speicheradresse $Ressource_X$ auf den Wert $lost$ gesetzt. Dies geschieht in unserem Fall mit dem Tupel $(Thread_B, Ressource_X)$.

4. Im letzten Schritt greift $Thread_B$ schreibend auf die Speicheradresse $Ressource_X$ zu. Der Wert des Tupels wurde im vorherigen Schritt auf $(Thread_B, Ressource_X) = lost$ gesetzt und weist im jetzigen Schritt auf ein Lost-Update hin. Eine Fehlermeldung wird ausgegeben. Um weitere Lost-Updates zu detektieren, wird nach der Ausschrift der Wert des Tupels auf w gesetzt.

Kapitel 3

Modellierung eines allgemeinen Detektors

In diesem Kapitel stellen wir das Konzept für die Generalisierung von Speicherzugriffsmustern vor. Wie im letzten Abschnitt gezeigt, ist es möglich Zugriffsmuster mit Sequenzdiagrammen zu beschreiben. Sequenzdiagramme sind in ihrer Ausdruckskraft jedoch beschränkt, denn für jede Zugriffsfolge muss ein eigenes Diagramm beschrieben werden. Darüber hinaus bieten UML-Sequenzdiagramme die Möglichkeit alternative Sequenzen, Schleifen und Bedingungen zu formulieren. Die Anwendung dieser Möglichkeiten wirkt sich jedoch negativ auf die Lesbarkeit (insbesondere größerer) Modelle aus.

Erfolgversprechender ist die Modellierung des Detektors in Form eines Zustandsdiagrammes. Hierbei werden die unzulässigen Muster in einem Zustandsdiagramm modelliert. Die zulässigen Muster sind nur soweit es zur Anwendung des Zustandsautomaten notwendig ist enthalten. Dies erleichtert nicht nur die Modellierung des Detektors, sondern auch die Generierung von Instrumentierungsvorschriften für den Zustandsautomaten.

3.1 Anforderungen an die Modellierung

Aus der vorangegangenen Arbeit können folgende Anforderungen an die Modellierungsmethode abgeleitet werden: Zum Markieren der letzten Zugriffsart eines Threads auf eine Speicherzelle, ist eine Tabelle nötig, in welcher die letzte Zugriffsart eines Threads gespeichert ist. Als Zugriffsarten sind lesender und schreibender Zugriff vorgesehen, sowie die Erweiterung um benutzerdefinierte Zugriffsarten. Außerdem sind zwei vordefinierte Ereignisse nötig, um auf Lese- und Schreibzugriffe zu reagieren. Beide Ereignisse tragen die Parameter ThreadID und Ressource. Der ungültige Zugriff muss markiert werden, hierzu sind ein Schlüsselwort oder besonders ausgezeichnete Modellelemente vorzusehen.

Wir schränken den Ansatz auf Shared-Memory-Architekturen ein. Auf andere Speicherarchitekturen, z.B. Cluster-Computing, ist dieser Ansatz nicht anwendbar. In der an die Realisierung des Konzepts anschließende Fallstudie werden ausschließlich x86-Architekturen betrachtet. Dies ist ausreichend, da das Konzept unabhängig von der Systemarchitektur auf alle Systeme mit Shared-Memory anwendbar ist.

3.2 UML Vorlagen für die Modellierung der Detektors

In diesem Abschnitt beschreiben wir die notwendigen UML-Vorlagen zur Erfüllung der oben genannten Anforderungen. Das Klassendiagramm in Abbildung 3.1 zeigt die statische Sicht auf den Detektor. Die abstrakte Klasse `AccessPatternDetector` hält die Tabelle zur Speicherung der letzten Zugriffsart `lastAccess`, die angebundenen Speicherzellen `memorycells` und die zu beobachtenden Prozesse `threads`. Die vordefinierten Ereignisse zum Empfangen von Lese- und Schreibzugriffen sind in der Klasse als Operationen deklariert.

Die abstrakte Enumeration `AccessTypes` deklariert die vordefinierten Zugriffsarten. Zur Anpassung für spezielle Bedürfnisse ist die Enumeration um weitere Literale erweiterbar.

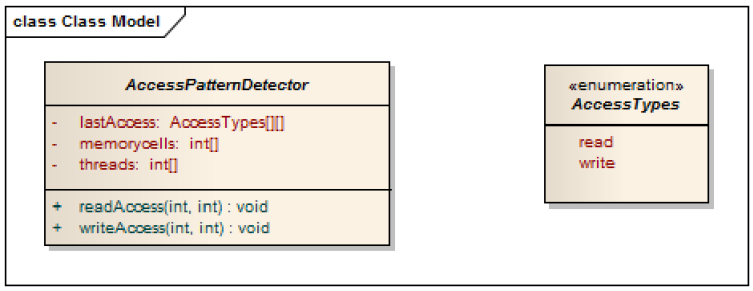

Abbildung 3.1: Vorlage des Klassendiagramms für den Detektor.

Auch für das Verhalten des Detektors haben wir eine Vorlage in Form eines Zustandsdiagrammes vorbereitet (Abb. 3.2). Typischerweise ist zunächst eine Unterscheidung in Lese- und Schreibzugriffe vorzunehmen. Hierfür gehen vom ersten Zustand `Wait for Access` zwei Kanten ab, die mit den Triggern für Lese- und Schreibzugriffe versehen sind. Das Erreichen eines Endzustands (ExitPoint) interpretieren wir als Auftreten eines unzulässigen Speichermusters. Damit ist auch die Forderung nach einem speziellen Modellelement oder Schlüsselwort erfüllt, welches ein unzulässiges Muster markiert. Die Modellierung des weiteren Verhalten ist dem Anwender überlassen. Auch die Konsequenzen falls ein unzulässiges Muster gefunden wurde, sind vom Anwender zu definieren.

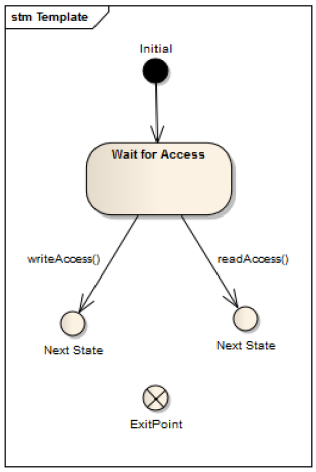

Abbildung 3.2: Vorlage für das allgemeine Vorgehen zur Modellierung der Zustandsmaschine.

3.3 Modellierung am Beispiel des Lost-Update-Detektors

Am Beispiel des in Abschnitt 2.2 eingeführten Lost-Update Detektors zeigen wir hier die Anwendung der UML-Vorlagen. Für die Erkennung eines Lost-Updates müssen wir das neue Literal `lost` einführen. Hierfür leiten wir eine neue Enumeration `LostUpdateTypes` aus der bereits bestehenden `AccessTypes` ab und fügen das neue Literal hinzu. Im Zuge dessen muss auch die Klasse `AccessPatternDetector` angepasst werden: Wir überschreiben den Typ des Attributs `lastAccess` in der abgeleiteten Klasse `LostUpdateDetector` mit dem Typ `LostUpdateTypes[][]`.

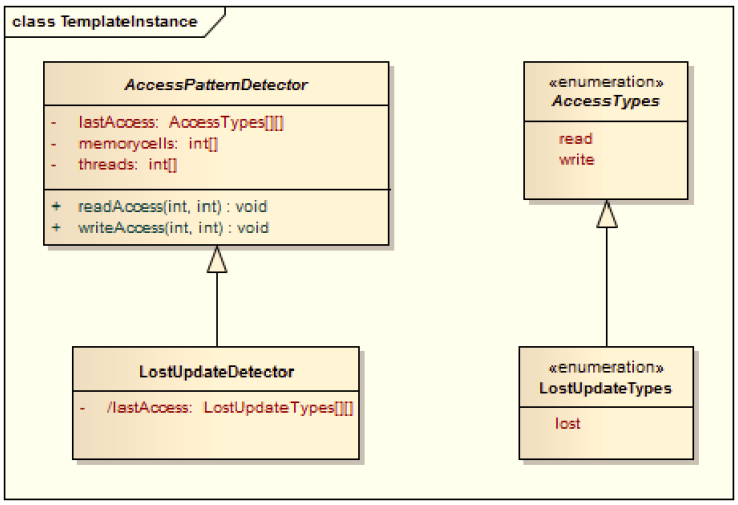

Abbildung 3.3: Anwendung des Klassendiagramms auf den Lost-Update Detektor.

Nach dem die statische Sicht auf den Detektor vollständig ist, können wir das Verhalten des Lost-Update-Detektors mit einem UML-Zustandsdiagramm beschreiben. Im vorgegeben Zustand `Wait for Access` wartet der Zustandsautomat auf einen der beiden Zugriff-Trigger. Beim Auslösen des Lesezugriff-Triggers `readAccess()` wird in der Tabelle `lastAccess` an der Stelle des betreffenden Prozesses und der Speicherzelle der Lesezugriff markiert. Der Zustandsautomat kehrt danach in den Wartezustand zurück.

Der schreibende Zugriff auf eine Speicherzelle löst ein komplexeres Verhalten aus. Wenn der letzte Zugriffe auf die Zelle des aktuellen Prozesses lesend war, so wird für alle Prozesse, die auf diese Speicherzelle zugreifen in der Tabelle `lastAccess` die Markierung `lost` gesetzt. Dies ist ein Flag für einen zukünftigen Zugriff. Semantisch wird hier also nicht die Vergangenheit gespeichert, sondern vielmehr ein Hinweis auf eine mögliche Zukunft gegeben. Danach wird wieder der Wartezustand angenommen.

Schreibt ein Thread nun auf eine Speicherzelle, die mit `lost` gekennzeichnet ist, wird der Zustandsautomat beendet. Die ist das Zeichen dafür, dass der Detektor ein Lost-Update erkannt hat und nun entsprechende Folgemaßnahmen zu treffen sind (Ausführung anhalten, Ablauf loggen und replay, Debugger starten, etc.). Für den Fall das ein Prozess schreibend auf eine Zelle zugreift, aber noch keine Zugriffsmarkierung hinterlegt wurde oder die Marke `write` gesetzt wurde, kehrt der Zustandsautomat umgehend in den Wartezustand zurück.

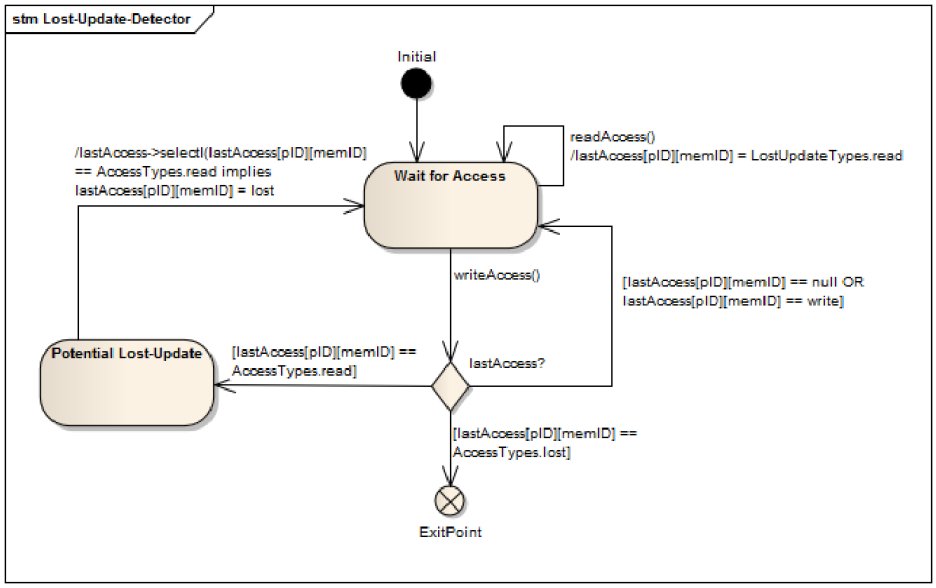

Abbildung 3.4: Anwendung des Zustandsdiagramms auf das Lost-Update Problem

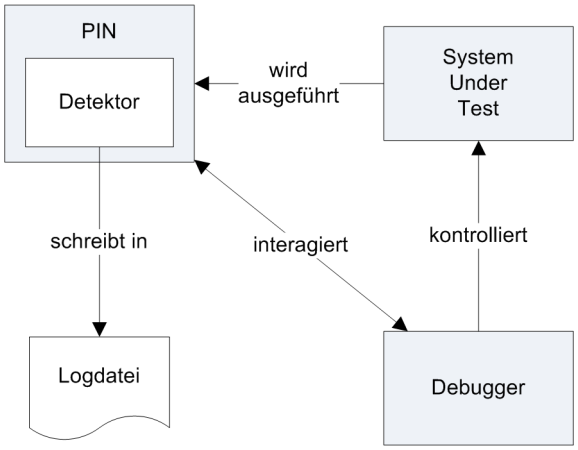

Abbildung 3.5: Die Interaktion und Kommunikation der Komponenten.

3.4 Abbildung der Modellelemente auf Code-Fragmente

Auf der oben beschriebenen Grundlage wurde ein Detektor für das Lost-Update Zugriffsmuster implementiert. Der Detektor ist ein PIN-Tool und kann als solches für jedes Binärprogramm, welches durch das PIN-Framework ausgeführt wird, verwendet werden.

3.4.1 Das Zusammenspiel der einzelnen Komponenten

Ein PIN-Tool sagt dem PIN-Framework, an welchen Stellen das zu untersuchende Programm instrumentiert werden soll. Neben den Hinweisen zur Instrumentierung sammelt der Detektor Informationen über die Schreib- und Lesezugriffe auf geteilte Speicherbereiche und aktualisiert auf dieser Basis die Zustandsmaschine (Abbildung 2.3 bzw. Abbildung 3.4). Wird der Detektor ohne das Kommando $-appdebug$ gestartet, wird das zu untersuchende Programm nicht durch einen Breakpoint angehalten. Dies ermöglicht den Einsatz des Detektors in unbeaufsichtigten Testläufen. Wird ein Lost-Update erkannt, wird dies in einer Logdatei vermerkt.

Der hier vorgestellte Detektor besteht aus folgenden Komponenten. Deren Interaktion ist in Abbildung 3.5 dargestellt:

- Dem PIN-Tool „pin_LostUpdateDetector“,
- dem „GNU Debugger GDB“
- und einem zu untersuchenden Programm.

Die PIN-API erlaubt die Kommunikation mit dem GNU Debugger GDB. Das Zusammenspiel der Komponenten ermöglicht es, komplexe semantische Breakpoints zu setzen, die mit einem normalen Debugger so nicht möglich sind. Wenn die $-appdebug$ Flag bei dem

Start des Pin-Tools gesetzt wurde, so wird der Programmfluss bei einem detektierten Lost-Update angehalten und die Kommunikation zum Debugger hergestellt. Der Entwickler ist so in der Lage die Situation genauer untersuchen zu können. Da das zu untersuchende Programm echt auf der Hardware ausgeführt wird, stehen die Funktionen des Debuggers im vollem Umfang zur Verfügung und spiegeln den realen Zustand des Programms wieder. Die Instrumentierung des zu untersuchenden Programms verläuft dabei für den Debugger transparent.

Der letzte Baustein ist das zu untersuchende Programm. Für die Untersuchung ist das Vorhandensein des Quelltextes nicht nötig. Um aber aussagekräftige Fehlermeldungen erzeugen zu können, versucht der Detektor Debugging-Informationen aus einem bereitgestellten Image zu extrahieren. Um ein Programm-Image mit solchen Informationen zu erzeugen, muss im Kompilierprozess die Debugginginformationen integriert werden. In der Fallstudie verwendete Kompiler GCC geschieht dies mit der Zusatzflag $-g$.

Eine beispielhafte Anwendung des Detektors ist im Anhang A zu finden.

Im Folgenden wird genauer auf die Programmteile eingegangen, die für das Instrumentieren von Schreib- und Lesezugriffen zuständig sind und die Funktion, welche die zu überwachenden Speicheradressen extrahiert.

Die in Abbildung 3.4 verwendeten Trigger `writeAccess()` und `readAccess()` lassen sich direkt auf die Funktionsaufrufe `RecordMemWrite()` und `RecordMemRead()` in Listing A.3.3 bzw. Listing 3.3 mappen.

3.4.2 Der Programmablauf des Detektors

```
1  int main( int argc, char *argv[] )
2  {
3    // use debugging information for PIN_GetSourceLocation
4      PIN_InitSymbols();
5
6      // print usage
7      if ( PIN_Init( argc, argv ) ) return Usage();
8
9      // Write to a file since cout and cerr maybe closed by the application
10     out.open( KnobOutputFile.Value().c_str() );
11     out.setf( ios::showbase );
12
13     // init locks
14     InitLock( &out_lock );
15
16     // extract globally shared data
17     readCandidatesFromImage();
18
19     // add instrumentation
20     INS_AddInstrumentFunction( Instruction, 0 );
21
22     // add finalize function
```

```
23      PIN_AddFiniFunction( Fini, 0 );
24
25      // Never returns
26      PIN_StartProgram();
27  }
```

Listing 3.1: Die main() Methode

Im Listing 3.1 ist die $main()$ Methode des Pin-Tools aufgelistet. Diese Methode stellt die Verbindung mit dem PIN-Framework her. Um dem Benutzer aussagekräftige Fehlermeldungen präsentieren zu können, werden in Zeile 4 die Symbolinformationen aus dem angegebenen Programm-Image gelesen.

Nicht alle Programme werden über ein Terminal gestartet oder besitzen eine Aussage in Form eines Terminals. Für diesen Fall werden keine Ausgaben des Detektors direkt in ein Terminal geschrieben sondern die Meldungen in einer Logdatei gespeichert. Der Pfad der Logdatei wird dabei mithilfe der PIN-API aus den mitgelieferten Argumenten extrahiert. Das Öffnen der Logdatei findet in Zeile 10 statt.

Der Umstand, dass das zu untersuchende Programm real auf der Hardware ausgeführt wird, birgt auch in diesem Detektor die Gefahr von Concurrency Bugs. Damit die Ausschrift in die Logdatei nicht von mehreren Threads gleichzeitig stattfindet, wird hierfür ein Lock genutzt. Die Initialisierung des Locks ist in Zeile 14 zu finden.

Bevor das zu untersuchende Programm gestartet und instrumentiert wird, wird in Zeile 17 die Methode $readCandidatesFromImage()$ aufgerufen. Diese extrahiert potenzielle Adressen, auf denen ein Lost-Updates stattfinden kann.

In Zeile 20 wird dem PIN-Framework mitgeteilt, welche Funktion die Instrumentierungsorte definiert.

Nach dem das zu untersuchende Programm in Zeile 26 gestartet wurde, muss noch das Filehandle zu der Logdatei geschlossen werden. Dies geschieht in der Funktion $Fini()$. In Zeile 23 wird dem PIN-Framework mitgeteilt, dass diese Funktion nach Beendigung aufgerufen werden muss.

3.4.3 Extrahieren von Adresskandidaten

```
1  void readCandidatesFromImage()
2  {
3    // try to load the image
4      IMG img = IMG_Open(KnobInputFile);
5      if (!IMG_Valid(img))
6      {
7          out << "Could not open " << KnobInputFile.Value() << endl;
8          exit(1);
9      }
10
11     // go over all sections to find global data
12     for (SEC sec = IMG_SecHead(img); SEC_Valid(sec); sec = SEC_Next(sec))
```

```
13      {
14        if(SEC_Name(sec).compare(".data") == 0 || // global initialized data
15          SEC_Name(sec).compare(".bss") == 0)    // global uninitialized data
16        {
17          for(ADDRINT i = SEC_Address(sec); i < SEC_Address(sec)+SEC_Size(sec); i++)
18          {
19            addCandidate( (void*) i );
20          }
21          }
22       }
23      IMG_Close(img);
24  }
```

Listing 3.2: Die readCandidatesFromImage() Methode

Es werden nur Lost-Updates detektiert, die auf global geteilten Variablen stattfinden. Dies ergibt sich aus der weiter oben zu finden Definition eines Data Races. Da das PIN-Framework auf der Binärebene instrumentiert, ist ein Auffinden von global geteilten Variablen nicht ohne weiteres möglich.

Das logische Konstrukt einer Variable wurde von dem Kompiler bei der Übersetzung des Quelltextes in Binärcode vernichtet. Eine Variable stellt nun eine Folge von Speicheradressen dar. Das binäre Speicherformat eines Programmes unter Linux heißt ELF, welches für „Executable and Linking Format“ steht. Die Variable ist nicht mehr eindeutig aufzufinden, die Speicheradressen jedoch schon.

Eine ELF Binärdatei besteht folgenden Bereichen:

- Der *Elf-Header* enthält Informationen über die Programmgröße, den genauen Typ und den Adresseinstiegspunkt an dem das Programm nach dem Laden gestartet wird.
- Der *Programm-Header* beschreibt die Zusammensetzung des Programms näher. Hier werden unter anderem die existierenden Sektionen aufgelistet. Sektionen sind Bereiche von Speicheradressen in denen zusätzliche Daten vom Kompiler abgelegt werden können. Zum Beispiel enthält die Sektion .rodata Daten die gelesen aber nicht geschrieben werden.
- Die *Symboltabelle* ist eine Sektion mit besonderer Bedeutung. Hier werden alle globalen Variablen und Funktionen abgelegt, die vom Programm implementiert bzw. verwendet werden. Diese Sektion wird in der Main Methode in Listing 3.1 ausgelesen.
- Der letzte Bereich ist die *Sektions-Headertabelle*. Sie dient der internen Verwaltung der Sektionen.

Um an die interessanten Adressen zu gelangen muss dieses Format näher untersucht werden. Dies geschieht in Listing 3.2.

Folgende Sektionen sind für das Extrahieren der Adresskandidaten von Interesse:

- Die Sektion .data enthält alle initialisierten globalen Variablen. Ein Zugriff auf eine Adresse im angegebenen Bereich muss also zu den Kandidaten dazu gezählt werden. Dies geschieht in Zeile 14 vom Listing 3.2.
- Die Sektion .bss enthält alle nicht initialisierten globalen Variablen. Ein Zugriff auf eine Adresse im angegebenen Bereich muss ebenso zu den Kandidaten dazu gezählt werden. Dies geschieht in Zeile 15 vom Listing 3.2.

Die Adressen werden in einer internen Liste gespeichert. Greift das zu untersuchenden Programm auf eine Speicheradresse zu, die ein Element der Kandidatenliste ist, so wird dieser Zugriff instrumentiert und die Zustandsmaschine aktualisiert.

3.4.4 Instrumentierung bei einem Lesezugriff

```
1  ADDRINT RecordMemRead(VOID * ip, VOID * address, THREADID threadid)
2  {
3    GetLock( &out_lock, threadid+1 );
4
5    // only consider accesses to candidates
6    if( isCandidate( address ) ) {
7
8      // write to log
9      out << "thread: " << dec << threadid
10       << "\taddress: "<< hex << address
11       << "\taccess: read" << endl;
12
13     // update the statemachine
14     setTuple( threadid, address, STATE_READ );
15   }
16
17   ReleaseLock( &out_lock );
18   return 0;
19 }
```

Listing 3.3: Die RecordMemRead() Methode

Wie in Abbildung 2.3 dargestellt wird bei einem Lesezugriff von Thread t auf eine Speicheradresse x nur der interne Zustand des Tupels (t, x) aktualisiert. Siehe Zeile 14 der Funktion $RecordMemRead()$ in Listing 3.3. Dies entspricht auch der in Abbildung 3.4 definierten Action `lastAccess[pID][memID] = LostUpdateTypes.read`.

Diese Funktion wird bei jedem Lesezugriff auf eine Speicheradresse von dem PIN-Framework ausgeführt. Da die Threads des zu untersuchenden Programms potenziell auf mehreren Prozessoren parallel ausgeführt werden können wird ein Lock in Zeile 3 verwendet, das den Zugriff serialisiert. Um unnötige Arbeit zu vermeiden werden nur Zugriffe auf Speicheradressen näher untersucht, welche in der Liste der Kandidaten aufgelistet sind. Um einen Livelock zu vermeiden wird das Lock in Zeile 17 wieder freigegeben.

3.4.5 Instrumentierung bei einem Schreibzugriff

```
1  ADDRINT RecordMemWrite( VOID * ip, VOID * address, THREADID threadid )
2  {
3    int returnval = 0; // set no breakpoint
4
5    GetLock( &out_lock, threadid+1 );
6
7    // only consider accesses to candidates
8    if( isCandidate( address ) ) {
9
10     // write to log
11     out << "thread: " << dec << threadid
12       << "\taddress: "<< hex << address
13       << "\taccess: write" << endl;
14
15     // update statemachine
16     int currentState = lookupTuple( threadid, address );
17     switch ( currentState )
18     {
19       case STATE_READ:
20         setAllTuplesHaving( address, STATE_READ, STATE_LOST);
21         setTuple( threadid, address, STATE_WRITE );
22       break;
23
24       case STATE_LOST :
25         reportLostUpdate( ip, threadid, address );
26         setTuple( threadid, address, STATE_WRITE );
27         returnval = 1;  // set breakpoint
28       break;
29
30       default:
31         setTuple( threadid, address, STATE_WRITE );
32       break;
33     }
34   }
35
36   ReleaseLock( &out_lock );
37   return returnval;
38 }
```

Listing 3.4: Die RecordMemWrite() Methode

Die in Listing A.3.3 angegebene Funktion $RecordMemWrite()$ wird bei jedem Schreibzugriff auf eine Speicheradresse von dem PIN-Framework ausgeführt. Auch hier führen nur Zugriffe auf Adressen in der Kandidatenliste zu einer Aktualisierung der Zustandsmaschine.

Im Gegensatz zu der Funktion $RecordMemRead()$ ist in dieser Funktion der Rückgabewert nicht immer 0. Wie in Abbildung 2.3 angegeben, wird nur ein Breakpoint gesetzt, wenn ein Schreibzugriff auf eine Speicheradresse stattfindet, die in einem vorherigen Schritt auf $lost$ gesetzt wurde. Durch das Setzen des Rückgabewertes auf 1 in Zeile 28

wird, nach Beendigung der Funktion, das PIN-Framework einen Breakpoint setzen, welches den Programmfluss anhält und die Ausführungskontrolle des Programms an den Debugger abgibt.

Das Case-Statement entspricht den in Abbildung 3.4 definierten Transitionen nach dem Trigger `writeAccess()`.

Kapitel 4

Diskussion

In diesem Kapitel vergleichen wir unseren generischen Modellierungsansatz mit einem Werkzeug, das für das Auffinden von Data Races spezialisiert ist. Außerdem stellen wie Wege vor, um dem Nichtdeterminismus bei einem echt parallel ausgeführten Programm zu begegnen.

4.1 Vergleich des Detektors mit Helgrind in Bezug auf das Lost-Update Beispiel

Um eine Aussage über die Qualität des hier vorgestellten Detektors machen zu können, wird zum Vergleich das Programm „Helgrind“ der Valgrind Version 3.6.0.SVN verwendet. Helgrind wird in der Standardkonfiguration ausgeführt. Beide Tools instrumentieren das zu untersuchende Programm auf Binärebene und untersuchen das Programm auf Data Races.

Neben dem erfolgreichen Finden des Defektes und der False-Positive Rate werden noch folgende Eigenschaften in die Bewertung aufgenommen:

- *Ausführungszeit* - Wie lange hat das Untersuchen des Programms gedauert? Hier gilt, je schneller die Ausführung, desto besser.
- *Hinweise* - Wieviel zusätzliche Informationen liefert das Werkzeug, damit ein Entwickler den Defekt finden und entfernen kann?
- *Rauschen* - Wieviel unnötige Informationen liefert das Werkzeug? Je mehr unnötige Informationen, desto schwieriger ist es für einen Entwickler die relevanten Meldungen zu finden.

4.1.1 Szenario A: Kein Data Race

In diesem Szenario untersuchen die beiden Werkzeuge das korrekte Programm „dataRace_free“. Das Programm ist im Anhang B.1 zu finden und enthält kein Concurrency Bug.

	Helgrind	**pin_lostUpdateDetector**
Defekte gefunden	keine	keine
False Positives	keine	keine
Ausführungszeit	0.556137 Sek.	5.299595 Sek.
Hinweise	eine Zeile	keine
Rauschen	9 Zeilen	92 Zeilen

Beide Werkzeuge haben keine Defekte gemeldet und keine Situation fälschlicherweise als einen Defekt identifiziert. Die Unterschiede in der Ausführungszeit sind dagegen gravierend. Der Detektor hat bei der Instrumentierung und Ausführung um den Faktor 10 länger gebraucht. Das Werkzeug Helgrind informiert den Entwickler mit einer Zusammenfassung über die Anzahl der gefundenen Defekte. Diese Information fehlt im Detektor.

4.1.2 Szenario B: Einfaches Data Race

In diesem Szenario untersuchen die beiden Werkzeuge das fehlerhafte Programm „dataRace_simple“. Das Programm ist im Anhang B.2 zu finden und enthält einen Concurrency Bug.

	Helgrind	**pin_lostUpdateDetector**
Defekte gefunden	2	1
False Positives	2	keine
Ausführungszeit	0.561707 Sek.	5.274523 Sek.
Hinweise	5 Zeilen	3 Zeilen
Rauschen	39 Zeilen	15 Zeilen

Unter der Instrumentierung des Werkzeugs Helgrind hat sich der Defekt nicht manifestiert und das Programm hat die korrekten Werte ausgeschrieben. Trotzdem wurden zwei Defekte mit dem Hinweis „Possible data race during read of size 4 at 0x80497f8 by thread #3“ gemeldet. Da diese aber nicht zu einer Aktivierung des Defektes geführt haben, werden diese Hinweise zu den False-Positives gezählt.

Im Gegensatz dazu, wurde der Defekt unter der Instrumentierung durch den Detektor aktiviert und erkannt: „LOST UPDATE: thread: 1 osthread: 1261 address: 0x80497f8 on line 29 in dataRace_simple.c.“. Auch das Rauschverhältnis der Hinweise hat sich in diesem Szenario zu Gunsten des Detektors gewandelt.

4.1.3 Szenario C: Komplexes Data Race

In diesem Szenario untersuchen die beiden Werkzeuge das defekte Programm „dataRace_complex“. Das Programm ist im Anhang B.3 zu finden und enthält trotzt der Verwendung von Synchronisationsmechanismen einen Concurrency Bug.

	Helgrind	**pin_lostUpdateDetector**
Defekte gefunden	2	1
False Positives	2	keine
Ausführungszeit	0.560401 Sek.	5.284035 Sek.
Hinweise	5 Zeilen	3 Zeilen
Rauschen	39 Zeilen	161 Zeilen

Auch in diesem Szenario ist unter der Instrumentierung von Helgrind der Defekt nicht aktiviert worden. Die relevante Stelle wurde wiederum als „possible datarace“ gekennzeichnet. Das Rauschverhältnis ist nun wieder zu Gunsten von Helgrind ausgefallen. Auch die Ausführungszeit spricht für Helgrind.

4.2 Herausforderung Nichtdeterminismus

Die Wiederholbarkeit von Testläufen, in denen sich das Programm deterministisch gleich verhält ist wie schon in Abschnitt 2.1 beschrieben, von grundlegender Bedeutung beim Debuggen von Software. Der hier vorgestellte Detektor ist leider auch dem Nichtdeterminismus von gethreadeten Programm erlegen. Es ist immer noch möglich, dass ein Defekt sich in Testläufen nicht manifestiert und so unentdeckt bleibt.

Durch das Beobachten mit einem Debugger wird das Threading-Verhalten massiv beeinflusst, was unter Umständen nicht zu dem (erwarteten) Fehlverhalten des Programms führt. Auf Basis von dem Instrumentierungs-Framework PIN wurde „PINPlay“ [10] entwickelt, welches der Anforderung der Wiederholbarkeit Rechnung trägt. Durch PINPlay ist es möglich Programmläufe deterministisch wiederholen zu lassen. Die Autoren argumentieren, dass existierende PINTools (der Detektor ist ein solches PINTool) ohne viel Aufwand angepasst und unter PINPlay ausgeführt werden können. Durch eine solche Anpassung sind Entwickler nun in der Lage das zur Manifestierung beitragende Interleaving der Threads aufzuzeichnen und deterministisch wiederholen zu lassen.

Das Werkzeug CHESS [11] von Microsoft erlaubt es die Thread-Ausführung gezielt zu explorieren und so alle möglichen Interleavings ausführen zu lassen. Leider ist diese Möglichkeit nur auf Programme, welche auf dem .Net Framework basieren, anwendbar. Mit der hier vorgestellten Modellierung ist es aber grundsätzlich möglich auch einen Detektor für das .Net Framework zu generieren und somit die Interleaving-Exploration von CHESS zu verwenden. Der in CHESS demonstrierte Ansatz zeigt, dass es möglich ist,

eine große Menge ausgewählter Thread-Interleavings ausführen zu lassen und so das zu untersuchende Programm intensiver zu testen.

Um eine möglichst hohe Zahl an verschiedenen Thread-Interleavings vom Programm ausführen zu lassen ist es auch möglich gezielt künstliche Warteschleifen in den Programmcode einzufügen. Das auf Java basierende Testframework ConTest [12] verwendet diese Methode erfolgreich. Solch eine Instrumentierung ist auch mit dem PIN-Framework möglich. Hierbei stellt sich aber die grundsätzliche Frage, in wieweit das aktuell untersuchte Programm (mit den künstlich eingefügten Warteschleifen) dem zu untersuchendem Programm gleicht.

Kapitel 5

Zusammenfassung & Ausblick

In dem hier vorliegendem Deliverable wurde ein prototypische Detektor vorgestellt. Aufbauend auf dem vorangehenden Deliverable 5.3.B wurde eine Modellierungsmethode entwickelt, mit deren Hilfe unterschiedlichste Detektoren definiert werden können. Es wurde gezeigt wie der modellierte Detektor in Quellcode und schlussendlich in ein ausführbares Programm transformierbar ist.

Die zentrale Schwierigkeit beim Debuggen von gethreadeter Software liegt im Aufspüren von Defekten und in der deterministischen Wiederholbarkeit des Fehlverhalten der Software, welches ein Beheben überhaupt erst ermöglicht. Für die Klasse der Concurrency Bugs, welche durch die Ausführung von mehreren Kontrollflüssen von Threads hervorgerufen werden, bietet die hier vorgestellte Modellierungstechnik eine effektive und plattformübergreifende Debugging-Methode. Durch die Verwendung des PIN-Frameworks ist die Wiederholbarkeit ebenso gewährleistet (siehe dazu PINPlay [10]).

Die entwickelte Modellierungstechnik dient zur Beschreibung unzulässiger Zugriffsmuster für Shared-Memory-Architekturen. Modelliert wird in gewohnter UML-Syntax, sodass Nutzer sich schnell in die Technik einarbeiten können. Das fertige Modell besteht aus einem Klassen- und einem Zustandsdiagramm, welche möglicherweise aus Unterklassen bzw. Submaschinen bestehen. Dieses Modell enthält noch keine plattformspezifischen Informationen, sodass eine Transformation für beliebiges Framework oder Programmiersprache möglich ist. Exemplarisch haben wir diese Funktionalität an einem x86 System mit Linux und dem PIN-Framework demonstriert.

Der vorgestellte Ansatz ist außerdem sehr robust gegen nachträgliche Änderungen an den Anforderungen. Dank der abstrakten Modellierung können Änderungen an der Anforderungsspezifikation in kürzester Zeit nachgeführt werden. Mit dem vorgestellten Transformationsmechanismus kann zuverlässig Code aus den Modellen abgeleitet werden.

Bei dem hier entwickelten Werkzeug handelt es sich noch um einen Prototypen, d.h. Toolsupport ist noch nicht ausgereift. Wir haben in diesem Deliverable gezeigt, dass die systematische Transformation in ausführbaren Code für das PIN-Framework möglich ist.

Es fehlt aber noch an einer konkreten Implementierung des Transformators. Auch für weitere Frameworks, wie etwa .Net könnten noch Transformation entwickelt werden.

Als Zielsprache für das Debuggen von Programmen wäre auch AspectJ [13] denkbar. Neben der Einschränkung, dass es nur für Javaprogramme anwendbar ist hierbei der höhere Abstraktionsgrad eine Herausforderung. AspectJ arbeitet auf Objekten in der virtuellen Maschine und nicht wie oben weiter dargestellt auf einzelnen Speicherzellen. An dieser Stelle besteht noch Forschungsbedarf.

Anhang A

Die Verwendung des Detektors

A.1 Voraussetzungen

Betriebssystem & Hardware

Das PIN-Framework läuft unter Windows, Linux sowie unter Mac OS. Folgende Hardwarearchitekturen werden zur Zeit unterstützt:

- Windows: x86 IA32 and intel64 (Visual Studio v6, v9 und v10)
- Linux: IA32 and intel64 (gcc 3.4 und neuer) , IA64 (gcc 3.4) und ARM (gcc 3.3.1)
- MacOS: IA32 (gcc 4.0)

PIN-Framework

Der entwickelte Detektor ist ein PIN-Tool das auf dem PIN-Framework aufbaut. Das PIN-Framework kann unter $http://www.pintool.org$ kostenlos bezogen werden.

Detektor

Die hier entstandene Software muss unter Umständen für die jeweilige Platform neu kompiliert werden. Auf der beiliegenden CD [1] ist der Detektor unter Linux für die Hardwarearchitektur IA32 kompiliert worden. Hinweise zum Kompilieren sind der PIN Dokumentation zu entnehmen.

[1] Auf Anfrage erhältlich

A.2 Installation

Für die Installation des Detektors reicht es das Pin-Tool an einen für das PIN-Framework zugänglichen Ort zu kopieren. Für das Funktionieren müssen keine Umgebungsvariablen (Linux/MacOS) oder Registry-Werte(Windows) angepasst werden. Es ist wichtig, dass das Pin-Tool binärkompatibel zu dem Betriebssystem und der Hardwarearchitektur ist.

A.3 Beispielhafte Anwendung

Der Detektor unterstützt zwei Arten der Ausführung:

- *silent* - In diesem Mode wird kein Breakpoint bei einem gefundenen Lost-Update gesetzt. Das zu untersuchende Programm wird nicht angehalten. Es findet eine Ausschrift der wichtigen Lese- und Schreibzugriffe in eine Logdatei statt.
- *debug* - In diesem Mode wird ein Breakpoint gesetzt. Das Programm blokiert so lange, bis der angeschlossene Debugger die Ausführung des zu untersuchenden Programms weiterführt. Es findet eine Ausschrift der wichtigen Lese- und Schreibzugriffe in eine Logdatei statt.

A.3.1 Beschreibung der Parameter

Um z.B. das Programm $dataRace\_simple$ auf Lost-Updates zu untersuchen würde der Detektor mit folgendem Aufruf gestartet werden:

„pin -appdebug -t pin_lostUpdateDetector.so -i dataRace_simple -o some.log -- dataRace_simple“

Der Aufruf hat folgende Bedeutung:

- *pin* - Dies ist der Aufruf zu dem PIN-Framework. Dieses Programm führt das zu untersuchende Programm aus.
- *-appdebug* - Diese Flag ist optional und entscheidet ob ein Breakpoint gesetzt werden soll oder nicht. Um einen Breakpoint setzten zu lassen muss dieses Flag angegeben sein.
- *-t pin_lostUpdateDetector.so* - mit dem Flag $-t$ wir dem PIN Programm gesagt, welches PIN-Tool geladen werden soll. In unserem Fall ist dies das Tool $pin\_lostUpdateDetector$.
- *-i dataRace_simple* - Um aussagekräftige Meldungen zu erzeugen muss das Image des zu untersuchenden Programms geladen werden. Dies geschieht mit der Flag $-i$.
- *-o some.log* - Das Flag $-o$ sagt dem PIN Programm wohin die Logausschriften geschrieben werden sollen.

- -- *dataRace_simple* - Nach dem Doppelstrich wird das zu untersuchende Programm angegeben. In unserem Fall ist dies das Programm $dataRace\_simple$.

A.3.2 silent mode - Ausführung ohne Breakpoint

In diesem Mode wird das zu untersuchende Programm bei einem gefundenen Lost-Update nicht angehalten. Beispielhaft wird nun das Programm $dataRace\_simple$ auf Lost-Updates hin untersucht.

Das Programm $dataRace\_simple$ startet vier Threads, die die auf Null initialisierte Variable $global\_counter$ jeweils um Eins erhöhen. Dabei werden die Zugriffe nicht synchronisiert. Die korrekte Ausgabe lautet:

```
$ ./dataRace_simple
counter value = 0
counter value = 4
```

Das Programm ohne Instrumentierung gibt aber gelegentlich eine andere Ausgabe aus. Folgende Ausgabe weißt auf einen Concurrency Bug hin:

```
$ ./dataRace_simple
counter value = 0
counter value = 3
```

Um das Programm zu untersuchen, wird der Detektor mit folgendem Aufruf gestartet:

```
$ pin -t pin_lostUpdateDetector.so \
  -i dataRace_simple \
  -o some.log -- dataRace_simple
counter value = 0
counter value = 3
```

Nach dem das zu untersuchende Programm terminiert hat kann die Logdatei untersucht werden. Die Logdatei weißt auf ein Lost-Update auf der Speicheradresse $0x80497f8$ hin:

```
thread: 0 address: 0x80497f8  access: read
thread: 1 address: 0x80497f8  access: read
thread: 2 address: 0x80497f8  access: read
thread: 1 address: 0x80497f8  access: write
thread: 2 address: 0x80497f8  access: write
***
LOST UPDATE:
thread: 2 osthread: 3383  address: 0x80497f8
on line 29 in ../../dataRace_simple.c.
***
thread: 4 address: 0x80497f8  access: read
thread: 4 address: 0x80497f8  access: write
thread: 3 address: 0x80497f8  access: read
thread: 3 address: 0x80497f8  access: write
thread: 0 address: 0x80497f8  access: read
thread: 0 address: 0x80497f0  access: read
```

```
thread: 0 address: 0x80497f4  access: read
thread: 0 address: 0x80497f0  access: write
```

Die Logdatei zeigt alle Schreib- und Lesezugriffe auf geteilte Ressourcen. Dadurch kann folgende Situation rekonstrukiert werden, die zu einem Lost-Update geführt hat:

1. Thread 1 liest von Adresse $0x80497f8$.
2. Thread 2 liest von Adresse $0x80497f8$.
3. Thread 1 schreibt in Adresse $0x80497f8$ den erhöhten Wert $0 + 1$.
4. Thread 2 schreibt in Adresse $0x80497f8$ den erhöhten Wert $0 + 1$ und überschreibt damit das vorher geschriebene Erhöhen von Thread 1.

A.3.3 debug mode - Ausführung mit Breakpoint

Um einen Breakpoint vom Detektor setzten zu lassen muss die Flag $-appdebug$ mit angegeben werden. Beispielhaft wird im Folgenden das Programm $dataRace\_simple$ vom Detektor auf Lost-Updates untersucht und ein Breakpoint gesetzt. Um das Programm näher untersuchen zu können wird zusätzlich der Gnu Debugger gestartet.

Der Detektor wird im debug-mode gestartet und wartet auf die Verbindung mit dem Debugger:

```
$ pin -appdebug -t pin_lostUpdateDetector.so -i dataRace_simple -o \
   some.log -- dataRace_simple
Application stopped until continued from debugger.
Start GDB, then issue this command at the (gdb) prompt:
  target remote :40424
```

Der Debugger wird gestartet und das Image des zu untersuchenden Programms wird eingelesen:

```
$ gdb dataRace_simple
GNU gdb (GDB) 7.1
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-slackware-linux".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from ../../dataRace_simple...done.
(gdb)
```

Der Debugger stellt eine Verbindung mit dem Detektor her und startet das zu untersuchende Programm mit dem Befehl $continue$:

```
(gdb) target remote :40424
Remote debugging using :40424
Reading symbols from /lib/ld-linux.so.2...(no debugging symbols found)...done.
Loaded symbols for /lib/ld-linux.so.2
0xb77fa810 in _start () from /lib/ld-linux.so.2
(gdb) continue
Continuing.
```

Das zu untersuchende Programm wird ausgeführt und es kommt zu einem Lost-Update.

```
counter value = 0
```

Der Debugger fängt den vom Detektor gesetzten Breakpoint. Das Programm kann nun vom Entwickler näher untersucht werden. Alle Funktionen des Debuggers stehen dazu im vollen Umfang zur Verfügung.

```
*** Lost-Update found ***[New Thread 1398]

Program received signal SIGTRAP, Trace/breakpoint trap.
[Switching to Thread 1398]
0x080484cb in increaseCounter () at dataRace_simple.c:29
29      if ( pthread_create( &threads[t], NULL, increaseCounter, &t ) ) {
(gdb)
```

Anhang B

Beispielprogramme

Alle Programme wurden mit dem GCC Compiler Version 4.4.4 kompiliert. Folgende Compilerflags wurden durchgehend verwendet:

- $-Wall$ - Das Programm wird mit höchster Warnstufe kompiliert. Jede Meldung wird als Warnung bewertet, wodurch das Programm erst erfolgreich kompiliert, wenn keine Warnungen mehr existieren.
- $-O0$ - Dieses Flag (Großbuchstabe o) setzt die Optimierungsstufe des Compilers auf Null, was die Optimierung komplett ausstellt. Dies führt dazu, dass das Binärprogramm mehr dem Quelltext des Orginalprogramms entspricht.
- $-g$ - Das Programm wird mit Debugging Informationen kompiliert. Das erzeugte Binärprogramm ist etwas umfangreicher, da nun zusätzliche Symbolinformationen enthalten sind.
- $-lpthread$ - Die Posix Thread Bibliothek wird verwendet.

B.1 dataRace_free

In diesem Programm führen vier Threads nebenläufig die Funktion $increase_counter$ aus. Es wird ein Mutex für die korrekte Synchronisation verwendet. Das Programm beinhaltet kein Concurrency Bug.

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4
5  #define NUMBER_OF_THREADS 4
6  int global_counter = 0;
7  pthread_mutex_t mutex1;
8
```

```
 9  // data race free
10  void *increaseCounter( )
11  {
12    pthread_mutex_lock( &mutex1 );
13
14    int tmp = global_counter + 1;
15    global_counter = tmp;
16
17    pthread_mutex_unlock( &mutex1 );
18
19    // exit
20    pthread_exit( NULL );
21    }
22
23  int main( int argc, char *argv[] )
24  {
25    printf( "counter value = %d\n", global_counter );
26
27    // initialize mutex
28    pthread_mutex_init( &mutex1, NULL );
29
30    //create and run a couple of threads
31    pthread_t threads[ NUMBER_OF_THREADS ];
32    long t;
33    for( t = 0; t < NUMBER_OF_THREADS; t++ ) {
34      if ( pthread_create( &threads[t], NULL, increaseCounter, &t ) ) {
35        printf( "ERROR: pthread_create(%ld)\n", t );
36        return -1;
37      }
38    }
39
40    //wait for all threads to finish
41    int i;
42    for ( i = 0; i < NUMBER_OF_THREADS; i++ ) {
43      if( pthread_join( threads[i], NULL ) ) {
44        printf( "ERROR: thread_join(%d)\n", i );
45        return -1;
46      }
47    }
48
49    // exit
50    printf( "counter value = %d\n", global_counter );
51    pthread_exit( NULL );
52  }
```

Listing B.1: dataRace_free.c

B.2 dataRace_simple

Diese Programm entspricht dem Programm in Listing B.1 mit dem Unterschied, dass hier die Synchronisation entfernt wurde. Dieses Programm enhält ein Data Race bzw. Lost-Update.

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  #define NUMBER_OF_THREADS 4
5  int global_counter = 0;
6
7  // contains data race, no lock
8  void *increaseCounter( )
9  {
10   int tmp = global_counter + 1;
11
12   //extra load -> might reveal data race
13   int i; for( i=0; i<2048; i++ ) ;
14
15   global_counter = tmp;
16
17   // exit
18   pthread_exit( NULL );
19 }
20
21 int main( int argc, char *argv[] )
22 {
23   printf( "counter value = %d\n", global_counter );
24
25   //create and run a couple of threads
26   pthread_t threads[ NUMBER_OF_THREADS ];
27   long t;
28   for( t = 0; t < NUMBER_OF_THREADS; t++ ) {
29     if ( pthread_create( &threads[t], NULL, increaseCounter, &t ) ) {
30       printf( "ERROR: pthread_create(%ld)\n", t );
31       return -1;
32     }
33   }
34
35   //wait for all threads to finish
36   int i;
37   for ( i = 0; i < NUMBER_OF_THREADS; i++ ) {
38     if( pthread_join( threads[i], NULL ) ) {
39       printf( "ERROR: thread_join(%d)\n", i );
40       return -1;
41     }
42   }
43
44   // exit
45   printf( "counter value = %d\n", global_counter );
46   pthread_exit( NULL );
```

```
47 }
```

Listing B.2: dataRace_simple.c

B.3 dataRace_complex

Dieses Programm ist die erweiterte Form des Programms in Listing B.2. Die Funktion erhöht zweimal den globalen Counter $global_counter$. Für die Synchronisation werden zwei unterschiedliche Mutexe verwendet. Dieses Programm ist fehlerhaft und enthält ein Data Race bzw. Lost-Update, da für eine Variable zwei verschiedene Locks verwendet werden.

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  #define NUMBER_OF_THREADS 4
5  int global_counter = 0;
6  pthread_mutex_t mutex1, mutex2;
7
8  // contains data race, despite locks!
9  void *increaseCounter( )
10 {
11   pthread_mutex_lock( &mutex1 );
12   int tmp = global_counter + 1;
13   //extra load -> might reveal data race
14   int i; for( i=0; i<1024; i++ ) ;
15   global_counter = tmp;
16   pthread_mutex_unlock( &mutex1 );
17
18
19   pthread_mutex_lock( &mutex2 );
20   tmp = global_counter + 1;
21   //extra load -> might reveal data race
22   int j; for( j=0; j<1024; j++ ) ;
23   global_counter = tmp;
24   pthread_mutex_unlock( &mutex2 );
25
26   // exit
27   pthread_exit( NULL );
28 }
29
30
31 int main( int argc, char *argv[] )
32 {
33   printf( "counter value = %d\n", global_counter );
34
35   // initialize mutexes
36   pthread_mutex_init( &mutex1, NULL );
37   pthread_mutex_init( &mutex2, NULL );
38
39   //create and run a couple of threads
```

```
40    pthread_t threads[ NUMBER_OF_THREADS ];
41    long t;
42    for( t = 0; t < NUMBER_OF_THREADS; t++ ) {
43      if ( pthread_create( &threads[t], NULL, increaseCounter, &t ) ) {
44        printf( "ERROR: pthread_create(%ld)\n", t );
45        return -1;
46      }
47    }
48
49    //wait for all threads to finish
50    int i;
51    for ( i = 0; i < NUMBER_OF_THREADS; i++ ) {
52      if( pthread_join( threads[i], NULL ) ) {
53        printf( "ERROR: thread_join(%d)\n", i );
54        return -1;
55      }
56    }
57
58    // exit
59    printf( "counter value = %d\n", global_counter );
60    pthread_exit( NULL );
61  }
```

Listing B.3: dataRace_complex.c

Literaturverzeichnis

[1] V. W. Habel, Spillner, *Umfrage 2011: Softwaretest in der Praxis*. Ringstraße 19 b, 69115 Heidelberg: dpunkt.verlag, 1 ed., 2012.

[2] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, *Manifesto for Agile Software Development*. Online verfügbar unter http://agilemanifesto.org/; letzter Zugriff 08.07.2010), 2001.

[3] R. C. Martin, *The Clean Coder: A Code of Conduct for Professional Programmers*. Prentice Hall International, 2011.

[4] E. W. Dijkstra, "Notes on Structured Programming." circulated privately, apr 1970.

[5] A. Zeller, "Automated debugging: Are we close," *Computer*, vol. 34, pp. 26–31, 2001.

[6] J. Gray, "Why do computers stop and what can be done about it?," in *Symposium on Reliability in Distributed Software and Database Systems*, pp. 3–12, 1986.

[7] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, pp. 89–100, June 2007.

[8] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: data race detection in practice," in *WBIA '09: Proceedings of the Workshop on Binary Instrumentation and Applications*, (New York, NY, USA), pp. 62–71, ACM, 2009.

[9] S. Park, S. Lu, and Y. Zhou, "Ctrigger: exposing atomicity violation bugs from their hiding places," in *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 25–36, ACM, 2009.

[10] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, (New York, NY, USA), pp. 2–11, ACM, 2010.

[11] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," in *Proceedings of the 8th*

USENIX conference on Operating systems design and implementation, OSDI'08, (Berkeley, CA, USA), pp. 267–280, USENIX Association, 2008.

[12] Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur, "Towards a framework and a benchmark for testing tools for multi-threaded programs: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 19, no. 3, pp. 267–279, 2007.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, (London, UK, UK), pp. 327–353, Springer-Verlag, 2001.