MULTICORE-SOFTWAREFEHLER IM VISIER: AUTOMATISCHE FEHLERERKENNUNG IN ENTWÜRFEN PARALLELER PROGRAMME

Wie scheibt man parallele Programme, ohne dabei Wettlauf-Situationen einzubauen? Solche Fehler sind beim Testen schwer zu finden und treiben die Entwicklungskosten hoch. Dieser Artikel beschreibt ein neues Verfahren aus der Forschung, um Nebenläufigkeitsfehler möglichst frühzeitig und automatisch zu erkennen. Das Verfahren erzeugt aus UML-Entwurfsdiagrammen eine formale Spezifikation der parallelen Struktur der Software und erkennt darin typische Fehlermuster. Der Ansatz hilft dem Entwickler in der Praxis, Nebenläufigkeitsfehler zu vermeiden, noch bevor sie zu Code werden.

die autoren

Oliver Denninger
(denninger@fzi.de)
leitet am FZI Forschungszentrum Informatik die Forschung zum Thema Multicore-Software. Er arbeitet zusammen mit Industriepartnern an Techniken zur automatischen Fehlererkennung und an Testwerkzeugen für parallele Anwendungen.

Dr. Frank Padberg
(frank.padberg@kit.edu)
arbeitet am Karlsruher Institut für Technologie (KIT). Er forscht an der automatischen Parallelisierung von Software, Techniken der Qualitätssicherung und schlanken Entwicklungsmethoden. Er wurde in den CACM unter den Top 50 International Software Engineering Scholars gelistet.

Mit der zunehmenden Verbreitung von Multicore-Prozessoren steigt auch der Bedarf an Software, die die Leistungsfähigkeit der neuen Prozessoren ausnutzt, indem rechenintensive Teile der Anwendung parallel ausgeführt werden. Das fehlerfreie Entwerfen und Implementieren von paralleler Software ist schwierig. Im Vergleich zur herkömmlichen Softwareentwicklung, bei der sequenziell gedacht wird, muss der Entwickler nun mehrere, gleichzeitig stattfindende („nebenläufige") Aktivitäten im Programm unter Kontrolle behalten und so aufeinander abstimmen, dass das korrekte Ergebnis berechnet wird.

In parallelen Programmen entstehen deshalb neue Arten von Fehlern – so genannte Nebenläufigkeitsfehler – die man von sequenziellen Programmen her nicht kennt. Leicht kann es zu Wettlauf-Situationen kommen, bei denen sich nebenläufige Schreib- und Lesezugriffe auf Daten zeitlich überlappen und dadurch zu inkorrekten Ergebnissen führen (vgl. [Lu08]). Häufig zeigt sich auch, dass der Entwickler mit seiner gewohnten, sequenziellen Denkweise implizit Annahmen über die Reihenfolge bestimmter Anweisungen im Code gemacht hat, die aber von seinem parallelen Programm tatsächlich nicht garantiert sind. Auf der Konferenz OOP 2013 zeigen und diskutieren wir solche Arten von Fehlern anhand von realen Beispielen aus großen Open-Source-Projekten wie Mozilla oder MySQL.

Nebenläufigkeitsfehler sind mit den herkömmlichen Testtechniken nur schwer zu finden. Ob bei dem Zugriff zweier Programmfäden (*Threads*) auf eine gemeinsame Variable ein Wettlauf zustande kommt, hängt nicht nur von den Eingabedaten des Programms ab, sondern auch von der konkreten Laufzeitsituation im System, also von der Systemlast und der aktuellen Zuteilung von Prozessorkernen zu den Programmfäden (*Scheduling*). Dadurch sind Nebenläufigkeitsfehler, die beim Anwender im Betrieb auftreten, für die Softwareentwickler häufig nicht oder nur mit großem Aufwand reproduzierbar. Die Fehlersuche in parallelen Programmen wird schwieriger und teurer.

Ansatz zur automatischen Fehlererkennung

Wir haben das Ziel, Nebenläufigkeitsfehler möglichst frühzeitig in der Entwicklung automatisch zu erkennen und zu vermeiden. In unserem Forschungsprojekt „QualiCore" (vgl. [FZI11], siehe Kasten 1) entwickeln wir ein Verfahren, das potenzielle Fehler schon anhand von Entwurfsdiagrammen aufdeckt, also zu einem Zeitpunkt, zu dem Softwaretests mangels Code noch gar nicht möglich sind. Je früher mögliche Fehler erkannt werden, desto leichter und kostengünstiger sind sie zu beheben.

Unser technischer Ansatz besteht aus zwei Schritten:

- Im ersten Schritt wird aus Entwurfsdiagrammen, die einen Ausschnitt aus der Kommunikation der Objekte in der Software zeigen, automatisch eine *formale Spezifikation* erzeugt. Die Spezifikation fokussiert auf die Nebenläufigkeitsstruktur des parallelen Programms. Als Eingabe dienen meist Sequenzdiagramme, weil sie noch am ehesten in der Entwicklungspraxis anzutreffen sind. Für die formale Spezifikation verwenden wir den Prozesskalkül *Communicating Sequential Processes (CSP)* (vgl. [Hoa85]), der genau für solche Zwecke entwickelt wurde. Die erzeugte Spezifikation – also das CSP-Modell – erfasst natürlich nicht die ganze Software, sondern nur den gewählten Entwurfsausschnitt.
- Im zweiten Schritt wird in dem CSP-Modell des parallelen Programms automatisch nach *Fehlermustern* gesucht. Die Fehlermuster spiegeln typische Nebenläufigkeitsfehler wider. Sie sind generisch gehalten und werden ebenfalls mit CSP modelliert. Je mehr wir über Nebenläufigkeitsfehler lernen, desto größer und umfassender wird unsere Datenbank mit Fehlermustern.

Modellerzeugung

In diesem Artikel beschränken wir uns auf Sequenzdiagramme als Ausgangspunkt für unser Verfahren. Das CSP-Modell zu einem gegebenen Sequenzdiagramm beschreibt die Kommunikation zwischen den beteiligten Objekten (Programmfäden, gemeinsame Datenstrukturen). Die Objekte werden als CSP-Prozesse modelliert. Beim Konstruieren der CSP-Formeln für die einzelnen Prozesse gibt es zwei grundsätzliche Schwierigkeiten, die zu lösen sind.

Die *technische Schwierigkeit* besteht darin, die synchrone Semantik von CSP mit

QualiCore ist ein Verbundprojekt des KIT und des FZI mit den KMU-Partnern GPP Communications GmbH (Oberhaching bei München), Acellere GmbH (St. Augustin bei Bonn) und petaFuel GmbH (Freising bei München). QualiCore erforscht Methoden zur Qualitätssicherung für Multicore-Software mit dem Ziel, Softwarefehler frühzeitig in der Entwicklung aufzudecken und den Entwickler bei der Fehlersuche in der Praxis mit Werkzeugen zu unterstützen. QualiCore wird vom BMBF seit 2011 finanziell gefördert.

Kasten 1: Das Projekt „QualiCore“.

der üblichen Semantik von objektorientierten und prozeduralen Programmen in Einklang zu bringen. In CSP kommunizieren zwei Prozesse ausschließlich dadurch, dass sie „an gemeinsamen Ereignissen teilhaben“, das heißt, in beiden Prozessen findet dasselbe Ereignis zum genau gleichen Zeitpunkt statt. Das gemeinsame Ereignis hat dann den Charakter einer Nachricht oder Kommunikation. Die herkömmliche Programmsemantik mit ihren blockierenden Funktionsaufrufen und Variablenzugriffen ist ein wenig anders. Die Lösung, die wir in QualiCore gefunden haben, führt für jeden Methoden- bzw. Funktionsaufruf ein eigenes Ereignis im Modell ein, das dann den beteiligten Prozessen gemeinsam ist. Dasselbe gilt auch für die Rückkehr von Funktionsaufrufen und das Lesen oder Schreiben von gemeinsam genutzten Attributen bzw. Variablen.

Die *konzeptionelle Schwierigkeit* besteht darin, von dem einen, im Sequenzdiagramm gezeigten Programmablauf auch auf andere mögliche Abläufe zu schließen, die der Entwickler – dem Diagramm nach zu urteilen – vermutlich in seinem Code realisieren wird. Schon im sequenziellen Fall wird ja nicht jedes mögliche Programmverhalten im Entwurf spezifiziert; die Implementierung schließt solche „Lücken“ dann durch die konkrete Wahl von Code. Der parallele Fall ist erheblich komplexer, weil sich das Verhalten der Programmfäden durch ihre gleichzeitige Ausführung überlappt (*Interleaving*). Dadurch werden in einem parallelen Programm viele Abläufe implizit möglich, die im Entwurf nicht gezeigt und häufig auch nicht angedacht sind. Genau in diesen, oft unbewusst mit-codierten Abläufen liegen mögliche Fehlerquellen für das parallele Programm.

Für unseren Ansatz zur Fehlererkennung ist es daher wichtig, nicht nur die Angaben im Sequenzdiagramm zu modellieren, sondern im CSP-Modell darüber hinaus zu gehen und zu versuchen, die Intention des Entwicklers und die implizit mit enthaltenen Abläufe ebenfalls einzufangen. Das gelingt in QualiCore durch bestimmte Verallgemeinerungen bei der Modellierung einzelner Objekte, wie sie im Sequenzdiagramm beschrieben sind – etwa durch das Zulassen alternativer Reihenfolgen von einzelnen Methodenaufrufen oder durch das Zulassen unterschiedlicher zeitlicher Überlappungen von Methodenaufrufen, die von den Kommunikationspartnern eines bestimmten Objekts ausgehen.

Beispiel

Wir möchten unser systematisches Vorgehen bei der Modellerzeugung an einem Beispiel für das parallele Kompressionsprogramm „Bzip2“ (vgl. [Gil]) erläutern.

Das Sequenzdiagramm in **Abbildung 1** zeigt das Kernstück der Kommunikation in diesem parallelen Programm. Der Hauptfaden (Main) reiht zu komprimierende Datenblöcke in eine Warteschlange (Fifo) ein (enqueue) und reserviert in der Ausgabeliste (Output) entsprechende Slots (allocatechunk). Parallel dazu liest ein zweiter Programmfaden (Consumer) Blöcke aus der Warteschlange (dequeue), komprimiert sie (compress) und schreibt das Ergebnis in die Ausgabeliste (writechunk). Gleichzeitig liest ein dritter Programmfaden (Filewriter) komprimierte Blöcke aus der Ausgabeliste (getchunk) und speichert sie auf der Festplatte (in der Abbildung nicht gezeigt). Die Prozesse Main, Consumer und Filewriter arbeiten also nebenläufig. Zur Synchronisierung benutzen sie ein Flag, das von Main gesetzt wird, um anzuzeigen, dass die Verarbeitung beendet werden soll. Das Flag wird von Consumer und Filewriter regelmäßig geprüft.

Die Applikation wird nun schrittweise und systematisch mit CSP als ein Prozess BZIP modelliert. Prozessnamen in CSP werden per Konvention großgeschrieben. Jedes Objekt im Sequenzdiagramm – gleich, ob es aktiv ist oder passiv – wird im Modell als ein eigener CSP-Prozess erfasst. Da die Objekte nebenläufig arbeiten, ist eine Parallelschaltung (Operator ||) BZIP der Prozesse für die einzelnen Objekte:

```
BZIP = MAIN || FIFO || OUTPUT || FLAG || CONSUMER ||
FILEWRITER
```

Jeder der Prozesse wird nun separat modelliert. Methodenaufrufe zwischen Objekten werden als Ereignisse in CSP modelliert. Ereignisnamen in CSP werden per Konvention kleingeschrieben. Die Ereignisse erhalten ein entsprechendes Präfix mc für method call im Namen, wobei die am Methodenaufruf beteiligten Objekte ebenfalls genannt werden. Zum Beispiel wird der Aufruf der Methode enqueue im Objekt Fifo durch das Objekt Main als Ereignis mc_MAIN_FIFO.enqueue modelliert. Entsprechendes gilt für die Rückkehr von Methodenaufrufen (mr für method return), das Erzeugen von Objekten (oc für object creation) und das Warten auf die Beendigung eines anderen aktiven Objekts bzw. Programmfadens (oj für object join). Methodenaufrufe innerhalb desselben Objekts werden etwas anders notiert, z. B. der Aufruf der compress-Methode innerhalb von Consumer als Ereignis compress_CONSUMER. Für das aktive Objekt Main ergibt sich so insgesamt die CSP-Formel:

```
MAIN = (mc_MAIN_FLAG.inittounset –>
mc_MAIN_FIFO.init –> oc_CONSUMER –> oc_FILEWRI-
TER –> producer_MAIN –> MX)

MX = (mc_MAIN_OUTPUT.allocatechunk –>
mc_MAIN_FIFO.enqueue –> MX
Π
mc_MAIN_OUTPUT.allocatechunk –>
mc_MAIN_FLAG.setflag –> oj_MAIN_FILEWRITER –>
mc_MAIN_FIFO.destroy –> SKIP)
```

In Main finden also zuerst nacheinander die Methodenaufrufe inittounset auf dem Flag und init auf der Warteschlange statt. Dann werden der Consumer-Faden und der Filewriter-Faden erzeugt und schließlich geht Main in seine Verarbeitungsschleife. Die Entscheidung, ob bzw. wann die Schleife verlassen wird, ist aus dem Sequenzdiagramm nicht ersichtlich – die Situation wird deshalb als eine nicht-deterministische Auswahl modelliert (Operator Π). Der MX-Prozess wird aus rein technischen Gründen eingeführt, um den Schleifenteil syntaktisch

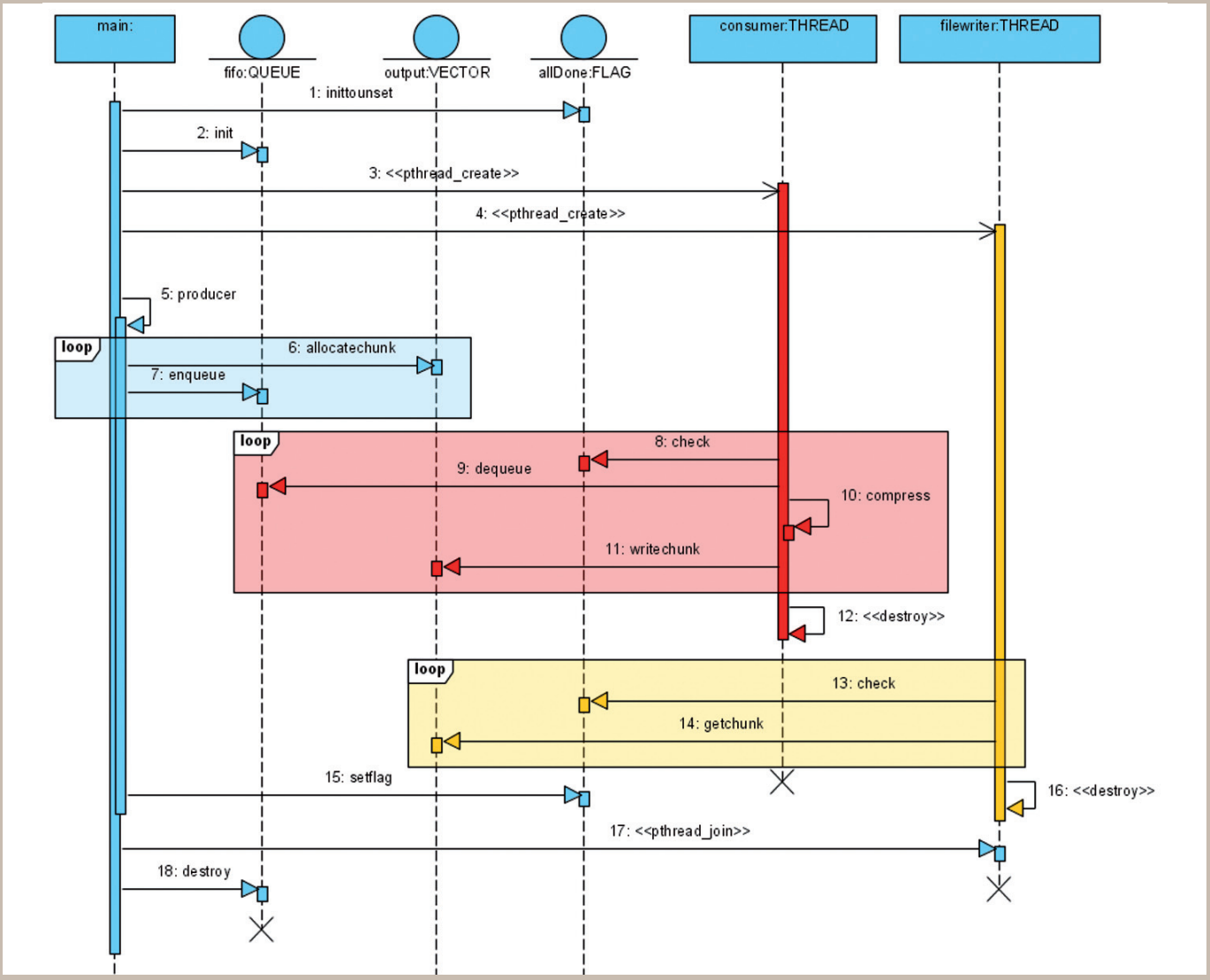

Abb. 1: Sequenzdiagramm zum parallelen Kompressionsprogramm Bzip2.

anbinden zu können. SKIP ist der CSP-Prozess, der nichts mehr tut, aber auch nicht blockiert. Er schließt das Modell für Main syntaktisch korrekt ab.

Passive Objekte, die nur dann etwas tun, wenn von einem anderen Objekt eine ihrer Methoden aufgerufen wird, werden in unserem Ansatz etwas anders modelliert. Ein Beispiel ist das passive Fifo-Objekt. Sowohl Main als auch Consumer rufen Methoden von Fifo auf. Die Methodenaufrufe an sich werden, wie gehabt, als eigene CSP-Ereignisse modelliert. Da die Aufrufe durch Main und Consumer aber nebenläufig erfolgen können, wird der CSP-Prozess für die Warteschlange (FIFO) als eine Parallelkomposition zweier Teilprozesse modelliert (Operator ||), von denen der eine (FX1) die Kommunikation mit Main, der andere (FX2) die mit Consumer abdeckt:

```
FIFO = FX1 || FX2
```

Aus Sicht von Fifo kann Main die Methoden init, enqueue oder destroy aufrufen. Nach Aufruf der destroy-Methode terminiert Fifo. Das Fifo-Objekt selbst hat keinen Einfluss darauf, wann welche Methode aufgerufen wird. Sein Verhalten bei Aufruf durch Main (FX1) wird daher als deterministische Auswahl der drei möglichen Methodenaufrufe modelliert (Operator |):

```
FX1 = (mc_MAIN_FIFO.init -> FX1 |
mc_MAIN_FIFO.enqueue -> FX1 |
mc_MAIN_FIFO.destroy -> od_FIFO -> SKIP)
```

Die Schleife um enqueue wird hier nicht mit modelliert, weil sie sich in der Ablauflogik von Main befindet, nicht in der Ablauflogik von Fifo. Das Verhalten von Fifo bezüglich Consumer (FX2) ist einfacher, nur die Methode dequeue kann aufgerufen werden:

```
FX2 = (mc_CONSUMER_FIFO.dequeue -> FX2)
```

Durch die Modellierung der Warteschlange als Parallelkomposition können sich im Modell die Ereignisse der beiden Teilprozesse (FX1 und FX2) überlappen und es entspricht ja auch dem real möglichen Verhalten des Programms, dass zeitgleich zwei Methoden desselben Objekts (hier: Fifo) von unterschiedlichen Fäden (hier: Main und Consumer) abgearbeitet werden können – mit allen zugehörigen Fehlerrisiken.

```
CONSUMER = (oc_CONSUMER –> CX)
CX = (mc_CONSUMER_FLAG.check –> mc_CONSUMER_FIFO.dequeue –> compress_CONSUMER –> mc_CONSUMER_OUTPUT.writechunk –> CX
□
mc_CONSUMER_FLAG.check –> destroy_CONSUMER –> od_CONSUMER –> SKIP)

FILEWRITER = (oc_FILEWRITER –> FX)
FX = (mc_FILEWRITER_FLAG.check –> mc_FILEWRITER_OUTPUT.getchunk –> FX
□
mc_FILEWRITER_FLAG.check –> destroy_FILEWRITER –> od_FILEWRITER –> oj_MAIN_FILEWRITER –> SKIP)
```

***Listing 1:** CSP-Modell für die aktiven Objekte* Consumer *und* Filewriter.

Die anderen Objekte, Consumer und Filewriter (aktiv) bzw. Output und Flag (passiv), werden ganz entsprechend und genauso systematisch mit CSP modelliert (**siehe Listing 1 und Listing 2**). Insgesamt ergibt sich ein überschaubares, kompaktes CSP-Modell für den im Sequenzdiagramm gezeigten Entwurfsausschnitt von Bzip2.

Fehlerfall

Die Synchronisierung, die im Sequenzdiagramm gezeigt wird, ist lückenhaft. Nach dem Setzen des Flags wartet Main auf die Beendigung von Filewriter (pthread_join), in der impliziten Annahme, dass vor Filewriter auch Consumer beendet sein wird. Das wird zwar im Diagramm mit der Reihenfolge der beiden Schleifenblöcke suggeriert, ist tatsächlich aber nicht gewährleistet. Es kann durchaus sein, dass das Flag von Main gesetzt wird, kurz nachdem Consumer es überprüft hat, aber kurz bevor Filewriter es prüft. Consumer wird dann versuchen, einen weiteren Block aus der Warteschlange zu lesen, obwohl der zugehörige Speicher bereits von Main freigegeben wurde (destroy). Dieser Nebenläufigkeitsfehler war tatsächlich in einer frühen parallelen Version von Bzip2 enthalten (vgl. [Yu09]).

Im CSP-Modell von Bzip2, das wir ausgehend von nur dem Sequenzdiagramm konstruiert haben, ist auch der fehlerhafte Ablauf möglich. Das CSP-Modell enthält die Ereignisfolge

<mc_CONSUMER_FLAG.check, mc_MAIN_FLAG.setflag, mc_MAIN_FIFO.destroy, mc_CONSUMER_FIFO.dequeue>

Diese Ereignisfolge wird von CSP-Modellprüfern wie FDR2 (vgl. [For]) im Modell BZIP gefunden und kann in Simulationen beobachtet werden. Unser schematisch gewonnenes CSP-Modell verallgemeinert offenbar das Sequenzdiagramm, das nur den vom Entwickler erwarteten Ablauf zeigt, und eignet sich dadurch zur Fehlersuche schon auf Entwurfsebene. Genau das wollen wir im QualiCore-Projekt erreichen.

Automatisierung

Derzeit arbeiten wir daran, alle Schritte in unserem Vorgehen zu automatisieren und prototypische Werkzeuge dafür bereitzustellen.

Das beginnt bei der Festlegung eines einheitlichen Formats, in dem die Sequenzdiagramme abgelegt werden. Wir haben mit unseren KMU-Partnern ein XML-Schema für QualiCore definiert, das mit verschiedenen Programmiersprachen nutzbar ist, insbesondere C/C++ und Java. Dabei berücksichtigen wir, dass manche Informationen auf verschiedene Weise in einem Sequenzdiagramm angegeben sein können. So stellen wir eine einheitliche Eingabe für die Modellerzeugung und Fehlererkennung her.

Für die Fehlererkennung legen wir die bekannten Fehlersituationen als generische CSP-Formeln in einer Datenbank ab. Die Fehlersituation, die im Bzip2-Beispiel enthalten ist, könnte man mit „Datenzugriff nach Objektlöschen“ umschreiben. Mit CSP lässt sich dieses Muster wie folgt generisch modellieren: ▶

impressum

Verlag: SIGS DATACOM GmbH
Lindlaustr. 2c, D-53842 Troisdorf
Tel.: +49 (0) 2241/2341-100, Fax: +49 (0) 2241/2341-199
www.sigs-datacom.de · E-Mail: info@sigs-datacom.de

Chefredakteur: Thorsten Janning
Lindlaustr. 2c, D-53842 Troisdorf
Tel.: +49 (0) 2241/2341-100, Fax: +49 (0) 2241/2341-199
E-Mail: fachartikel@objektspektrum.de

Mitglieder der Redaktion:
Matthias Bohlen, Consultant
Wolfgang Keller, object architects
Dr. Johannes Mainusch, Otto GmbH & Co. KG
Prof. Dr. Andreas Rausch, Technische Universität Clausthal
Martin Schimak, Martin Schimak GmbH
Rainer Singvogel, msg systems ag
Torsten Weber, GROSSWEBER, Groß, Weber & Partner

Honorary Editorial Member: Dr. Frances Paulisch

Aus der Szene: Ulrich Schmitz
E-Mail: Ulrich.Schmitz@sigs-datacom.de

Schlussredakteurin: Kirsten Waldheim,
E-Mail: schlussredaktion@objektspektrum.de

Freie Mitarbeiterin: Elke Niedermair, E-Mail: e.a.n@gmx.de

Manuskripteinsendungen: Manuskripte werden von der Redaktion – ausschließlich in elektronischer Form – gerne angenommen. Mit der Einsendung von Manuskripten gibt der Verfasser die Zustimmung zum Abdruck, zur Vervielfältigung auf Datenträgern und zur Übersetzung. Weitere Informationen und Honorarfestlegungen finden Sie in unseren Autorenrichtlinien: www.sigs-datacom.de/fachzeitschriften/objektspektrum/hinweise-fuer-autoren/richtlinien.html.

Verlagsleitung: Günter Fuhrmeister

Redaktions- und Herstellungsleitung Zeitschriften:
Susanne Herl, Tel.: +49 (0) 2241/2341-550,
E-Mail: Susanne.Herl@sigs-datacom.de

Leserservice: objektspektrum@sigs-datacom.de

Vertriebs- und Marketingleitung: Beate Friedrichs
Tel.: +49(0)2241 / 2341 - 560, Email: Beate.Friedrichs@sigs-datacom.de

Anzeigen: Brigitta und Karl Reinhart
Ostring 5h, D-85630 Grasbrunn,
Tel.: +49 (0) 89/464729, Fax: +49 (0) 89/463815,
E-Mail: Karl.Reinhart@sigs-datacom.de

Druck: F&W Mediencenter GmbH
Holzhauser Feld 2 · D-83361 Kienberg · Tel.: +49 (0) 8628/9884-52
E-Mail: anzeigendaten@objektspektrum.de

Abonnenten-Service: IPS Datenservice GmbH, Postfach 13 31,
D-53335 Meckenheim, Tel.: +49 (0) 22 25/70 85-374
Fax: +49 (0) 22 25/70 85-376, Patrick König, Markus Preis
E-Mail: aboservice@sigs-datacom.de

Erscheinungsweise: zweimonatlich

Bezugspreise:
Einzelverkaufspreis: € 9,90 (D), € 10,90 (A), sfr 18,20 (CH)
Jahresabonnement Deutschland: € 56,40 inkl. Versandkosten
Jahresabonnement Europa: € 62,40 inkl. Versandkosten
Jahresabonnement außerhalb Europas: € 72,30 inkl. Versandkosten
Studentenabo: € 50,80 inkl. Versandkosten

Bankverbindung:
Postbank München, Konto-Nr. 560 737 801, BLZ 700 100 80

Lieferung an Handel:
Verlagsunion KG, Postfach 5707,
D-65047 Wiesbaden, Tel.: +49 (0) 61 23/620-0

Layout und Produktion: leuchtfeuer-agentur · Sven Lesemann
Büro in Köln: Schanzenstraße 31 · D-51063 Köln,
www.leuchtfeuer-agentur.de · E-Mail: Lesemann@leuchtfeuer-agentur.de

ISSN 0945-0491

Die Zeitschrift wird mit chlorfreiem Papier hergestellt

```
OUTPUT = OX1 || OX2 || OX3
OX1 = (mc_MAIN_OUTPUT.allocatechunk –> OX1)
OX2 = (mc_FILEWRITER_OUTPUT.getchunk –> OX2)
OX3 = (mc_CONSUMER_OUTPUT.writechunk –> OX3)

FLAG = FLAGX1 || FLAGX2 || FLAGX3
FLAGX1 = (mc_MAIN_FLAG.inittounset –> FLAGX1 | mc_MAIN_FLAG.setflag –> FLAGX1)
FLAGX2 = (mc_FILEWRITER_FLAG.check –> FLAGX2)
FLAGX3 = (mc_CONSUMER_FLAG.check –> FLAGX3)
```

***Listing 2:** CSP-Modell für die passiven Objekte* Output *und* Flag.

```
ACCESSAFTERDELETION = (od_SD –>
mc_P_SD.method –> STOP)
```

Dabei steht SD für ein gemeinsames Datenobjekt *(Shared Data)*, method für eine beliebige Methode dieses Objekts und P für einen weiteren Prozess, der diese Methode aufrufen will.

Um nach potenziellen Fehlern in einem konkret vorliegenden CSP-Modell einer Applikation suchen zu können, muss jedes Fehlermuster zunächst für die vorgelegte Applikation instanziiert werden, d. h. für die generischen Prozess- und Ereignisnamen des Musters müssen die konkreten Prozess- und Ereignisnamen der Applikation eingesetzt werden. Wir erzeugen dazu systematisch alle vom Fehlermuster her sinnvollen Belegungen. Im Beispiel Bzip2 werden für SD und P aus dem Fehlermuster ACCESSAFTERDELETION alle sechs Objektnamen aus BZIP eingesetzt; von den 30 möglichen Belegungen ergeben aber nur zwei einen Sinn. Für diese Belegungen werden alle möglichen Methodennamen eingesetzt. Das ergibt dann vier sinnvolle Instanzen des Fehlermusters.

Danach übergeben wir das CSP-Modell der Applikation und jedes der instanziierten Fehlermuster einem Modellprüfer, der automatisch überprüft, ob die Ereignisfolge eines Fehlermusters in den Ereignisfolgen der Applikation auftreten kann oder nicht. Falls ja, ist ein potenzieller Nebenläufigkeitsfehler auf Modellebene gefunden. Derzeit benutzen wir FDR2, der für akademische Zwecke frei nutzbar und leistungsfähig ist. Die Rechenzeit für die Suche nach dem ACCESSAFTERDELETION-Muster beträgt im Bzip2-Beispiel mit FDR2 weniger als eine halbe Sekunde.

Wir arbeiten auch an einem Werkzeug, das das CSP-Modell automatisch aus einem oder mehreren Sequenzdiagrammen erzeugt. Dabei ist eine Reihe von Fallunterscheidungen und Sonderfällen zu berücksichtigen, etwa zwischen aktiven und passiven Objekten. Wir entwickeln außerdem Verfahren, um ein CSP-Modell auch aus anderen Entwurfsdiagrammen zu gewinnen, etwa aus Zustandsautomaten. Entscheidend ist, dass das CSP-Modell die in den Diagrammen gezeigten Abläufe geeignet verallgemeinert.

Erweiterungen

Mit unseren KMU-Partnern GPP Communications, Acellere und petaFuel arbeiten wir daran, die automatisierte Suche nach Nebenläufigkeitsfehlern auch für solche Projekte zu unterstützen, bei denen bereits lauffähiger Code in Java oder C/C++ vorliegt. Dazu wird der Code instrumentiert und es werden Laufzeitspuren erstellt, die in etwa den Gehalt von Sequenzdiagrammen haben. Ein solches Laufzeitprofil zeigt unter anderem die Methodenaufrufe, Objektinstanziierungen und Synchronisierungen. Das nachbearbeitete Profil wird dann in demselben Format abgelegt wie die Sequenzdiagramme und kann später von dem Modellerzeuger und dem Fehlererkenner weiterverarbeitet werden. Wir arbeiten außerdem daran, den Rückbezug von einem potenziellen Fehler auf die entsprechende Stelle im Code herzustellen.

Ausblick

In unserem Forschungsprojekt QualiCore sind die Konzepte und das schematische Vorgehen für die automatische Erzeugung von formalen CSP-Modellen aus Sequenzdiagrammen und die automatische Suche nach Nebenläufigkeitsfehlern in diesen Modellen ausgearbeitet. Erste Prototypen der Werkzeuge zur Automatisierung des Verfahrens liegen vor. Mit der Erweiterung unseres Ansatzes auf Laufzeitprofile von Anwendungen wollen wir die automatische Fehlersuche nicht nur für traditionelle Entwicklungsprozesse unterstützen, bei denen schon Code vorliegt, sondern auch für agile Prozesse, bei denen es keine vorgelagerte Entwurfsphase gibt und schnell zum Codieren übergegangen wird.

Unser Katalog an Fehlermustern wächst beständig an. Mit nur einer Ausnahme war in allen Fallbeispielen, an denen wir unser Verfahren bisher erprobt haben, der Nebenläufigkeitsfehler in dem entsprechenden CSP-Modell enthalten. Das Modell wurde immer ausgehend vom Entwurfsdiagramm, d. h. nur von dem erwarteten Szenario des Programmverhaltens aus, konstruiert. Es scheint also tatsächlich möglich zu sein, schon mit fragmentarischer Entwurfsinformation potenzielle Nebenläufigkeitsfehler in parallelen Programmen aufzudecken und zu vermeiden – auch solche Fehler, die von impliziten Reihenfolge-Annahmen des Entwicklers verursacht sind und die mit herkömmlichem Testen oder gängigen Wettlauf-Erkennern schwer zu finden sind. ■

Literatur & Links

[For] Formal Sytems Software, siehe: fsel.com/software.html

[FZI11] FZI Forschungszentrum Informatik, Das QualiCore-Projekt, 2011, siehe: qualicore-projekt.de/

[Gil] J. Gilchrist, Parallel BZIP2 (PBZIP2) – Data Compression Software, siehe: compression.ca/pbzip2/

[Hoa85] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall 1985

[Lu08] S. Lu, S. Park, E. Seo, Y. Zhou, Learning from Mistakes – A Comprehensive Study on RealWorld Concurrency Bug Characteristics, in: Proc. of 13th Int. Conf. on Architectural Support for Programming Languages and Operating Systems ASPLOS 2008

[Yu09] J. Yu, S. Narayanasamy, A Case for an Interleaving Constrained Shared-Memory Multi-Processor, in: Proc. of 36th Int. Symp. on Computer Architecture ISCA 2009