Ablaufvisualisierung für nebenläufige Java-Programme mit UML Interaktionsdiagrammen

Katharina Mehner, Annika Wagner

AG Datenbank- und Informationssysteme
Fachbereich Informatik
Universität Paderborn
D-33095 Paderborn

{mehner|awa}@upb.de

Nebenläufige Programmierung ist auch mit modernen Sprachen wie Java, die eigene Sprachkonstrukte für die Synchronisation bieten, komplex. Neben den aus der sequentiellen Programmierung bekannten Fehlern kommt es hier zu Fehlern bei der Sicherheit und der Lebendigkeit der ausgeführten Berechnungen. Diese Fehler sind besonders schwer zu finden, da nebenläufige Programme nichtdeterministisch sind. Während bisherige Testwerkzeuge und formale Analysen mit textuellen Repräsentationen von Programmabläufen arbeiten, denken wir, daß eine Visualisierung das Verständnis und das Auffinden von Fehlern erleichtern kann. Dazu verwenden wir UML, eine weitverbreitete visuelle objektorientierte Modellierungssprache. Wir zeigen wie man Programmabläufe mit UML Sequenz- und Kollaborationsdiagrammen visualisieren kann. Dabei konzentrieren wir uns auf die Darstellung der Synchronisation und erweitern UML, um damit die Laufzeitmechanismen von Java zur Synchronisation darzustellen. Abschließend stellen wir ein Konzept für ein Werkzeug vor.

1 Einleitung

Nebenläufige Programmierung gewinnt zunehmend an Bedeutung, nicht nur in verteilten Anwendungen, sondern auch bei Anwendungen für einzelne Rechner. Dem wird in modernen Programmiersprachen Rechnung getragen, indem sie die Nebenläufigkeit in der Sprache selber unterstützen, statt wie bisher nur durch entsprechende Bibliotheken. Ein wichtiges Beispiel dafür ist die Sprache Java [4], in die abstrakte Konstrukte für die Synchronisation eingegangen sind. Nebenläufige Programmierung bleibt trotzdem schwierig, weil zusätzlich zu den aus der sequentiellen Programmierung bekannten Fehlern, Fehler durch die Nebenläufigkeit hinzukommen. Im wesentlichen unterscheidet man *Sicherheitsprobleme* und *Lebendigkeitsprobleme*, wie zum Beispiel Deadlocks (Verklemmungen) oder Dormancy (d.h. schlafende oder wartende Threads) [8]. Diese Fehler sind besonders schwer zu finden, da nebenläufige Programme nichtdeterministisch sind.

Es existieren eine Reihe von Werkzeugen zum *Testen* [2] und zur *formalen Analyse* [16] nebenläufiger Programme. Hierbei spielen aus der realen Ausführung eines Programmes gewonnene aber auch fiktive Ausschnitte von Programmabläufen (auch

Programm-*Traces* genannt) eine zentrale Rolle. Sie werden oft in textueller Form repräsentiert. Dies ist aber nicht adäquat für komplexe Fehlersituation, wie man sie in nebenläufiger Programmierung findet [10]. Daher betrachten wir die Visualisierung von Programmabläufen. Um Fehler wie Deadlocks oder Dormancy zu verstehen, reicht es aber nicht, Programmabläufe auf Ebene der Sprachkonstrukte von Java zu visualisieren, da die Konstrukte zur Synchronisation sehr abstrakt sind. Um Fehler zu finden, muß man die dahinter verborgenen Laufzeitmechanismen verstehen. Daher ist es unser Ziel, zusammen mit den Programmabläufen die *Laufzeitmechanismen* von Java zur Synchronisation zu *visualisieren*.

Zur Visualisierung verwenden wir die objektorientierte Modellierungssprache *UML (Unified Modeling Language)* [3,14]. Der Vorteil von UML liegt neben der weiten Verbreitung darin, daß man eine Sprache wählt, die bereits im Softwareentwicklungsprozeß eingesetzt wird. Dies erlaubt es, die bei der Visualisierung entstandenen Diagramme mit denen aus der Modellierung zu vergleichen. Darüber hinaus vermeidet man, daß eine weitere neue Sprache erlernt werden muß.

In Abschnitt 2 gehen wir auf Programm-Traces ein. In Abschnitt 3 stellen wir kurz die Synchronisationsmechanismen in Java vor. In Abschnitt 4 stellen wir die Visualisierung mit UML für den wechselseitigen Ausschluß vor. In Abschnitt 5 skizzieren wir eine Werkzeugunterstützung. In Abschnitt 6 betrachten wir verwandte Arbeiten und in Abschnitt 7 geben wir eine Zusammenfassung.

2 Fehlersuche mit Programm-Traces

Man kann auf zwei verschiedene Weisen das Verhalten nebenläufiger Programme untersuchen. *Statische* Analyseverfahren untersuchen den Quelltext, während *dynamische* Analyseverfahren zusätzlich Information aus der Ausführung von Programmen gewinnen. Dazu werden bei der Programmausführung ausgeführte Anweisungen, Methodenaufrufe sowie Änderungen an Variablen und Objekten mitprotokolliert. Ein solches Protokoll nennt man auch einen *Programm-Trace*.

Die Fehlersuche in nebenläufigen Programmen unterscheidet sich wesentlich von sequentiellen Programmen. Zur Fehlersuche in nebenläufigen Programmen muß man nicht nur die Korrektheit von Berechnungsvorschriften prüfen, sondern auch noch verschiedene Abläufe, da Threads (nebenläufige Kontrollflüsse) nichtdeterministisch abgearbeitet werden, wovon aber nicht das Resultat abhängig sein sollte. Zunächst muß man für die Fehlersuche solche verschiedenen Abläufe aus den Programmen gewinnen. Während man bei *nichtdeterministischen* Testen durch mehrfaches Ausführen versucht, voneinander unterschiedliche Programm-Traces zu gewinnen, erzwingt man beim *deterministischen* Testen eine gewünschte Ablaufreihenfolge von Threads [1]. Die so gewonnenen Traces müssen dann auf Fehler untersucht werden. Dazu gibt es verschiedene Hilfmittel wie z.B. *Zeitstempel*[1], um den kausalen Zusammenhang zwischen verschiedenen Threads zu erfassen oder *Race Analysis*, um herauszufinden, wo es zwischen Threads möglicherweise *Race Conditions* gibt [13].

Interessanterweise spielen Programm-Traces nicht nur in der dynamischen Analyse eine Rolle, sondern auch in der statischen. Zur formalen Analyse von Quellcode werden unter anderem *Modelchecker* eingesetzt [16]. Für nebenläufige Programme kann

ein solcher Modelchecker z.B. Programm-Traces berechnen, die zu einem Deadlock führen.

Die Darstellung zeigt, daß es ausgereifte Techniken gibt, um Programm-Traces zu erzeugen. Aufgrund der Komplexität der aufzufindenden Fehler stellt sich die Frage nach geeigneten Visualisierungtechniken. Für die Fehlersuche werden unter Umständen auch verschiedene Sichtweisen auf die erzeugten Programm-Traces gebraucht. In [2] wird z.B. mit Programm-Traces gearbeitet, welche die für die Synchronisation relevanten Ereignisse herausfiltern, allerdings auch nur in textueller Form. Wir wollen für Programm-Traces insbesondere die Laufzeitmechanismen für Sprachen mit abstrakten Synchronisationmechansimen wie Java visualisieren. Bevor wir darauf näher eingehen, geben wir zunächst einen Überblick über das Modell der Nebenläufigkeit in Java.

3 Nebenläufige Programmierung in Java

Die Sprache Java unterstützt nebenläufige Programmierung mit *Threads* in der Sprache selber [4,8]. Ein Thread ist ein sequentieller Kontrollfluß innerhalb eines Programmes. In Java sind Threads zugleich Objekte und die Klassenbibliothek bietet entsprechende Methoden darauf an.

Verschiedene Threads innerhalb eines Programmes können die gleichen Objekte verwenden, um ihre Funktionalität zu implementieren. Aber auch die Kommunikation zwischen Threads geschieht über gemeinsame Objekte (sogenannte *shared objects*). Um Inkonsistenzen gemeinsam genutzter Objekte zu vermeiden, muß der Zugriff verschiedener Threads synchronisiert werden. Darüberhinaus müssen Threads koordiniert werden, um Informationen austauschen zu können, d.h. sie müssen manchmal aufeinander warten.

Der Schutz von Daten vor simultanen Zugriffen und die Koordination von Zugriffen geschieht klassischer Weise mit einem sogenannten *Monitor*. Ein Monitor verwaltet zum einen den Zugriff auf zu schützende Daten, indem er nur maximal einen Thread auf diesen Daten arbeiten läßt, auch bekannt als Konzept des *Wechselseitigen Ausschlusses*. Der Monitor unterstützt zum anderen die Koordination zwischen Threads, indem er Mechanismen anbietet, mit denen Threads auf das Eintreffen bestimmter Bedingungen warten können, bzw. Mechanismen, mit denen Threads signalisieren können, daß sich Bedingungen verändert haben. Wartende Threads werden in einer Warteschlange verwalten.

Monitore sind als Sprachkonzept in Java eingegangen. Dabei ist jedem Objekt ein Monitor zugeordnet. Mit dem Schlüsselwort `synchronized` kann ein Anweisungsblock oder eine gesamte Methode gekennzeichnet werden, die nie von mehr als einem Thread gleichzeitig ausgeführt werden kann. Mit den Methoden `wait`, `notify` und `notifyAll` kann die Warteschlange des Monitors manipuliert werden.

Mit `synchronized` wird immer ein gesamtes Objekt für alle anderen Methoden oder Blöcke, die ebenfalls als `synchronized` deklariert sind, gesperrt. Bei einer als `synchronized` deklarierten Methode ist dies implizit das Objekt auf dem die Methode aufgerufen wird. Im Falle eines Anweisungsblockes muß das Objekt bei der Deklaration von `synchronized` angeben werden.

Nur ein Aufruf einer dieser Methoden oder Anweisungblöcke erhält den *Lock* (dt. Sperre) für das Objekt, um damit arbeiten zu können. Aufrufe andere Threads werden solange blockiert. Ist der erste Aufruf abgearbeitet, erhält einer der blockierten Aufrufe in nichtdeterministischer Auswahl als nächster den Lock und kann abgearbeitet werden.

Durch die Blockierung beim Warten auf einen Lock ergibt sich die Gefahr für einen *Deadlock*. Im einfachsten Fall kann es zu einem Deadlock kommen, wenn zwei Threads je ein Objekt gelockt haben und auf die Freigabe des jeweils anderen Locks warten. Die notwendige Voraussetzung für einen Deadlock ist somit, daß es einen Zyklus in der Beziehung zwischen Objekten gibt, die von verschiedenen Threads genutzt werden. Deadlocks sind schwer zu finden, da sie von einer konkreten Aufrufreihenfolge in verschiedenen Threads abhängen, die an den Stellen, an denen die Threads dann nicht mehr weiterarbeiten können, nicht nachvollziehbar ist. Daher ist der Kontext und die Aufrufgeschichte nötig, um einen Deadlock überhaupt beschreiben zu können. Deadlocks treten aufgrund des Nichtdeterminismus bei der Abarbeitung von Threads nicht zwingend auf, selbst wenn das Programm die Möglichkeit eines Deadlocks birgt.

Die Methoden `wait` und `notify(All)` können nur innerhalb eines `synchronized`-Bereiches verwendet werden. Bei Aufruf von `wait` wird der Thread, der das Objekt gerade benutzt in die Warteschlange eingereiht, dabei geht er in den Zustand wartend über. Aus dieser Warteschlange können Threads nur entfernt werden, wenn andere Threads auf demselben Objekt `notify(All)` aufrufen. Mit `notify` wird nur ein Thread aus der Warteschlange entfernt, während mit `notifyAll` alle Threads entfernt werden. Entfernte Threads werden wieder aktiviert. `notify` kann nur von aktiven Threads aufgerufen werden. Gibt es in einem Programm nur noch wartende Threads und keine aktiven mehr, können die wartenden nie wieder aktiviert werden. Dieses Problem ist auch bekannt als *Dormancy*, d.h. für immer schlafende Threads. Ein lebendiges Programm setzt also eine Aufeinanderabstimmung von `wait`- und `notify`-Methoden voraus.

4 Visualisierung von wechselseitigem Ausschluß

In diesem Abschnitt betrachten wir die Visualisierung des wechselseitigen Ausschlusses, der in Java mittels `synchronized` programmiert wird. Wir verwenden UML-Interaktionsdiagramme (Sequenz- und Kollaborationsdiagramme), mit denen Abläufe auf Instanzebene modelliert werden können [3,14]. In diesem Kontext verwenden wir keine Klassendiagramme, da wir annehmen, daß diese bereits aus der Modellierungsphase vorhanden sind. Wir werden sehen, daß beim Verständnis von Synchronisation verschiedene Aspekte eine Rolle spielen, die sich mit verschiedenen UML-Interaktionsdiagrammen darstellen lassen. Wir setzen voraus, daß dem Leser die wesentlichen Bestandteile von Sequenz- und Kollaborationsdiagrammen bekannt sind.

Wir verwenden den von UML vorgesehenen *Erweiterungmechanismus* des *Stereotyps*, um die UML-Diagramme an Java anzupassen. Mittels Stereotypen lassen sich neue Sprachelemente von gegebenen UML-Modellierungselementen ableiten,

um UML an einen spezifischen Anwendungsbereich anzupassen. Der Stereotyp wird dabei entweder in doppelten spitzen Klammern <<...>> in Verbindung mit dem ursprünglichen Sprachelement angegeben oder durch ein neues graphisches Element dargestellt. So kann z.B. das Modellierungselement Klasse zu einer Javaklasse spezialisiert werden durch Kennzeichnung mit dem Stereotyp <<Java>>.

Sequenzdiagramme

Um die Synchronisation von Threads zu verstehen, spielt die genaue zeitliche Reihenfolge von Aufrufen eine besondere Rolle, weil davon abhängt, zu welchem Ergebnis der Ablauf eines nebenläufigen Programmes führt. In UML gibt es nur eine Diagrammart, mit der man Abläufe in ihrer *zeitlichen* Reihenfolge visualisieren kann: Sequenzdiagramme.

Als Beispiel betrachten wir eine einfache Klasse `cell` (vgl. Abbildung 1). Die Klasse `cell` hat einen Inhalt `value` und besitzt eine Methode `swap()`, um den Inhalt mit einer anderen Zelle zu tauschen. Der Tauschvorgang muß natürlich atomar sein, d.h. es dürfen keine anderen Objekte in dieser Zeit die Zelleninhalte lesen oder ändern. Daher werden wie in einem 2-Phasen-Protokoll zuerst via `synchronized` die benötigten Locks angefordert, nämlich auf beide beteiligten Zellobjekte. Danach werden die Inhalte getauscht.

```
class cell {
      private int value;
      public void swap(cell other){
           synchronized(this){
            synchronized(other){
               int newValue = other.value;
               other.value = value;
               value = newValue;}  }
      }
...}
```

Abbildung 1 Java-Klasse `cell`

Die einfachste denkbare nebenläufige Situation mit Objekten dieser Klasse ist, daß zwei Zellen unabhängig von einander von zwei verschiedenen Threads aufgefordert werden, ihre Inhalte zu tauschen. Zwei verschiedene Abläufe, die sich aus dieser Situation ergeben können, sind in Abbildung 2 und 3 als Sequenzdiagramme dargestellt. Beide Abläufe beruhen aber auf dem gleichen Javaprogramm, daß wir hier aus Platzgründen nicht zeigen. In den Sequenzdiagrammen sind die beiden Threads als aktive Objekte `client1` und `client2` dargestellt.

Problematisch bei dieser Visualisierung ist die Repräsentation von `synchronized`. Hierbei handelt es sich um ein Sprachkonstrukt von Java, was im eigentlichen Sinne keinem Methodenaufruf entspricht. Ohne Repräsentation von `synchronized` verliert das Sequenzdiagramm jedoch jegliche Aussagekraft. Die Frage, wie ein `synchronized`–Block am besten zu repräsentieren ist, läßt sich beantworten,

indem man sich mit seiner Semantik befaßt: zu Beginn des Blocks soll ein Lock auf das Parameterobjekt angefordert werden. Wir repräsentieren das Schlüsselwort `synchronized` daher als expliziten synchronen Methodenaufruf auf dem Parameterobjekt. Der Aufruf muß synchron erfolgen, weil das aufrufende Objekt auf den Erhalt des Locks warten muß. Am Ende des `synchronized`–Block wird der Lock durch einen synchronen Aufruf wieder zurückgegeben.

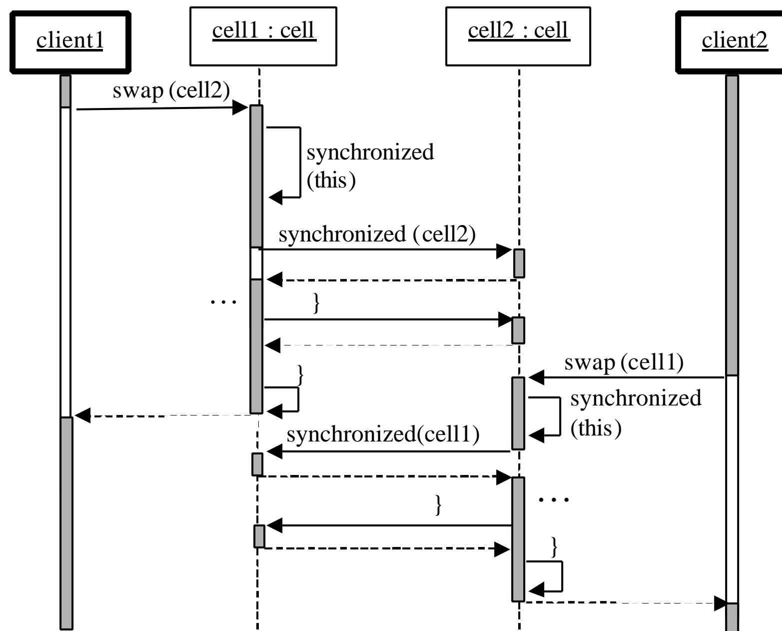

Abbildung 2 Sequenzdiagramm für `swap`

Während das erste Sequenzdiagramm einen Ablauf zeigt, bei dem beide Zellen ihre Inhalte erfolgreich austauschen, kommt es im zweiten Sequenzdiagramm zu einem Deadlock. Betrachten wir zunächst das Diagramm aus Abbildung 2. Hier tauscht `cell1` zuerst ihren Inhalt mit `cell2`. Dazu fordert sie die nötigen Locks an, die sie auch erhält. Der eigentliche Austausch der Inhalte ist an der durch `...` markierten Stelle ausgelassen. Anschließend gibt `cell1` ihre Locks wieder frei. Dann wird `cell2` zum Austausch der Inhalte aufgefordert. Dieser Austausch verläuft genauso problemlos. Man beachte, daß die Schraffur einer Aktivierung angibt, an welcher Stelle ein aktiviertes Objekt tatsächlich etwas berechnet. Eine nicht schraffierte Aktivierung gibt an, daß die Aktivierung warten muß. (Diese Notation wurde in UML 1.3 neu eingeführt.)

Hierin liegt auch schon der Unterschied zum zweiten Sequenzdiagramm in Abbildung 3. An der Tatsache, daß irgendwann alle Aktivierungen nicht mehr schraffiert sind, erkennt man, daß ein Deadlock eingetreten ist. Weiterhin zeigt das Diagramm auch noch die Aufrufreihenfolge, die zu diesem Deadlock geführt hat.

Während `cell1` versucht, ihren Inhalt mit `cell2` zu tauschen, versucht `cell2` den Tausch bereits in umgekehrter Richtung. Was jedoch nach wie vor nicht ersicht-

lich ist, ist der Grund, *warum* diese Aufrufreihenfolge zu einem Deadlock führt. Dieser Grund liegt in der Semantik von `synchronized`, das einen Lock auf das Empfängerobjekt anfordert. Hat bereits ein anderes Objekt diesen Lock, muß der Sender von `synchronized` warten. Die Tatsache, daß ein anderes Objekt diesen Lock besitzt, ist eine besondere Art von Beziehung zwischen dem Objekt, welches den Lock hält, und dem gelockten Objekt.

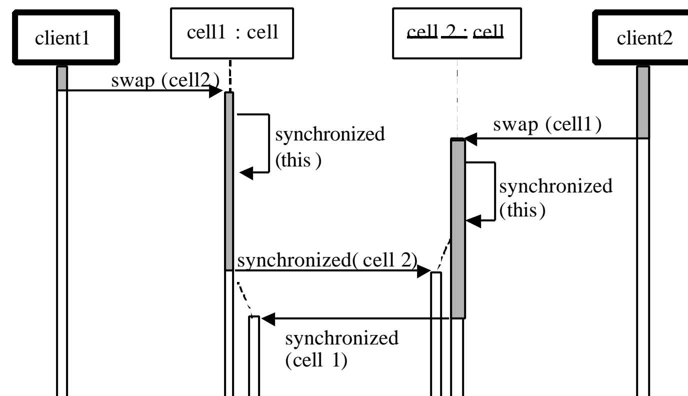

Abbildung 3 Sequenzdiagramm mit Deadlock

Beziehungen jeglicher Art sind im Sequenzdiagramm jedoch nicht sichtbar. Da Beziehungen jedoch in Kollaborationsdiagrammen dargestellt werden können, liegt es nahe, diese zu verwenden, um in Ergänzung zum Sequenzdiagramm auch noch den Grund des Deadlocks analysieren zu können.

Kollaborationsdiagramme

In diesem Abschnitt zeigen wir, wie man mit Hilfe geeigneter Erweiterungen von Kollaborationsdiagrammen die Gründe näher analysieren kann, die zu einer Deadlock-Situation geführt haben.

Dazu betrachten wir zunächst wieder den Ablauf, wie er im Sequenzdiagramm in Abbildung 3 dargestellt ist, und übersetzen ihn in ein Kollaborationsdiagramm (vgl. Abbildung 4). Wie schon im Sequenzdiagramm werden die beiden Threads als aktive Objekte dargestellt. Wir zeigen die gleichen Methodenaufrufe und `synchronized`-Anweisungen wie zuvor. Die zeitliche Anordnung des Sequenzdiagramms wird in Sequenznummern übersetzt. Nebenläufige Threads werden durch unterschiedliche Buchstaben gekennzeichnet, hier mit `A` und `B`. Unteraufrufe werden durch geschachtelte Numerierungen gekennzeichnet. Mit den Sequenznummern kann man jedoch nur die zeitliche Anordnung innerhalb eines einzigen Threads zeigen, nicht jedoch die globale zeitliche Anordnung wie vorher. Diese Information geht bei der Übersetzung verloren, weil Kollaborationsdiagramme nicht zeitliche Bezüge zwischen verschiedenen Threads darstellen können.

Durch die Parameterübergabe entstehen zwei neue Links vom Stereotyp `<<parameter>>`. Im Aufruf `swap(cell2)` von `client1` auf `cell1` werden die Aufru-

fe `synchronized(this)` und `synchronized(cell2)` abgearbeitet (analog für `client2` und `cell2`). Bei der Repräsentation von `synchronized` verfahren wir genauso wie im Sequenzdiagramm. Man beachte, daß somit dem Kollaborationsdiagramm aus Abbildung 4 beide Sequenzdiagramme aus Abbildung 2 und 3 zugrunde liegen könnten. Daher kann es auch nicht den Deadlock zeigen, was eigentlich unsere Absicht war.

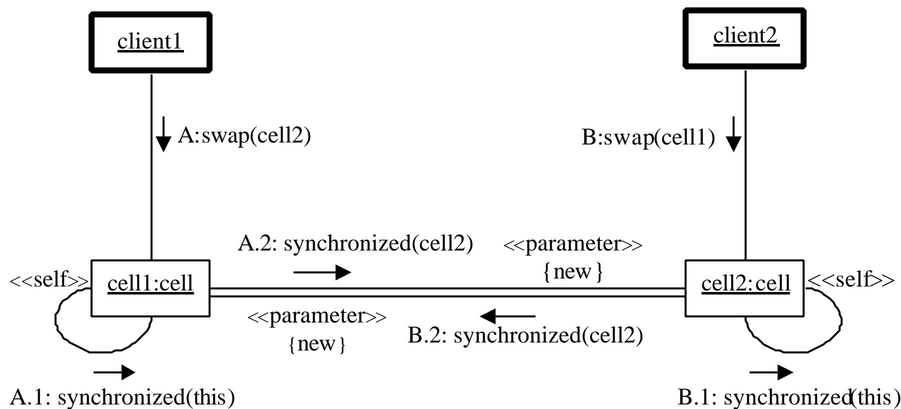

Abbildung 4 Kollaborationsdiagramm

Stereotypisierte Beziehungen im Kollaborationsdiagramm

Der eigentliche Zweck unseres Überganges vom Sequenzdiagramm zum Kollaborationsdiagramm war die Repräsentation der internen Vorgänge bei einer `synchronized`-Anweisung. Der Besitz eines Locks stellt eine gerichtete Beziehung zwischen dem Objekt, das den Lock hält; und dem gelockten Objekt dar. Dabei handelt es sich jedoch um keinen normalen Link. Unserer Beziehung liegt keine Assoziation im Klassendiagramm zugrunde und sie ist auch nicht navigierbar. Daher führen wir einen *Stereotyp* `<<locks>>` ein. Analog verhält es sich mit der Situation, in der ein Objekt einen Lock anfordert, ihn jedoch nicht erhält und deshalb warten muß. Hierfür sehen wir den *Stereotyp* `<<acquires>>` vor.

Repräsentieren wir nun die Situationen aus den Sequenzdiagrammen in Abbildung 2 und 3 als Kollaborationsdiagramm mit diesen Erweiterungen, so unterscheiden sich die entstehenden Kollaborationsdiagramme. Im Falle des Deadlocks sieht das Kollaborationsdiagramm wie in Abbildung 5 dargestellt aus. Sowohl die beiden Instanzen des Stereotyps `<<locks>>` als auch die des Stereotyps `<<acquires>>` sind neu zugefügt und mit dem Schlüsselwort `{new}` markiert. Im Falle eines Kollaborationsdiagrammes für das linke Sequenzdiagramm aus Abbildung 2 wären nur die beiden Locks vorhanden und diese wären `{transient}`, weil sie auch wieder freigegeben werden.

Mit der Einführung der Stereotypen `<<locks>>` und `<<acquires>>` sind wir unserem Ziel, die Gründe eines Deadlocks zu visualisieren, näher gekommen. Erkennbar wird, daß alle Threads blockiert sind. Beide Threads haben als letzte Anwei-

sung mittels synchronized einen Lock angefordert, auf den sie warten müssen. Verfolgt man die Links der Stereotypen <<locks>> und <<acquires>>, so kann man sehen, daß die Threads sich sogar gegenseitig blockieren, da sie jeweils einen Lock anfordern, welchen der andere hält. Genau das macht den Deadlock aus.

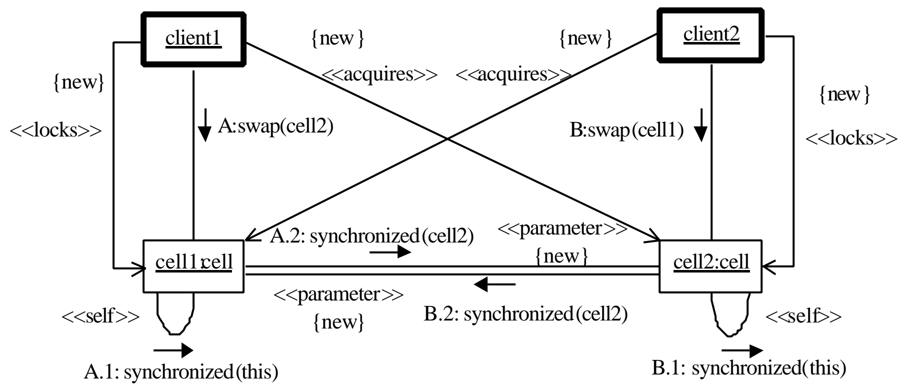

Abbildung 5 Kollaborationsdiagramm mit stereotypisierten Links

Unsere bisherigen Erweiterungen haben also dazu beigetragen, daß man die für den Deadlock verantwortlichen Locks bestimmen kann. Um den Deadlock beheben zu können, muß man jedoch die angeforderten Locks und das Warten auf Locks auch den entsprechenden Anweisungen zuordnen können. Bei diesem Beispiel ist das sehr einfach, weil sich die synchronized Anweisungen direkt eine Ebene unter dem swap() Aufruf befinden. Somit ist die Zuordnung von den <<locks>> und <<acquires>> Links problemlos durchzuführen. Wenn sich im Rumpf einer Methode zwei synchronized-Blöcke hintereinander befinden, ist die Zuordnung nicht mehr möglich.

Die Situation wird noch wesentlich komplizierter, wenn eine solche synchronized-Anweisung sich irgendwo in einer Folge von geschachtelten Methodenaufrufen (alle auf dem gleichen Objekt) befindet. Dann wird man zur Beschreibung von Synchronisation nicht unbedingt alle diese Methodenaufrufe ins Kollaborationsdiagramm aufnehmen, sondern vielleicht nur die synchronized-Anweisung oder die umgebenden Methode darstellen. D.h. man will nicht nur wie im Beispiel die Semantik einer einzelnen synchronized-Anweisung visualisieren, sondern auch die Möglichkeit nutzen, von einem ganz Block von Anweisungen und ineinander geschachtelten Methodenaufrufen nur die Semantik hinsichtlich der Sychronisation zu abstrahieren. Wie man auch dann noch die Entstehung des Locks der synchronized-Anweisung bzw. der umgebenden Methode zuordnen kann, erläutern wir im nächsten Abschnitt.

Numerierung neu erzeugter stereotypisierter Links

Im vorherigen Abschnitt haben wir motiviert, daß die Zuordnung der <<locks>>- und <<acquires>>-Links zu den sie erzeugenden Anweisungen bzw. Methoden wichtig ist, um die Stellen im Code ausfindig zu machen, die ursäch-

lich für den Deadlock sind. Diese Zuordnung können wir vornehmen, indem wir entstehende Links analog zu Unteraufrufen mit in die Numerierung einbeziehen. In Abbildung 6 haben wir die Links der Stereotypen `<<locks>>` und `<<acquires>>` numeriert.

Normalerweise wird in Kollaborationsdiagrammen bewußt auf die Numerierung dieser Links verzichtet. Der Grund dafür ist, daß von der konkreten Reihenfolge ihrer Erzeugung abstrahiert werden soll, was beim Einsatz von Kollaborationsdiagrammen in der Modellierungsphase Sinn macht. Diese Abstraktion ist aber in unserem Zusammenhang unerwünscht, da wir konkrete Programmabläufe visualisieren wollen.

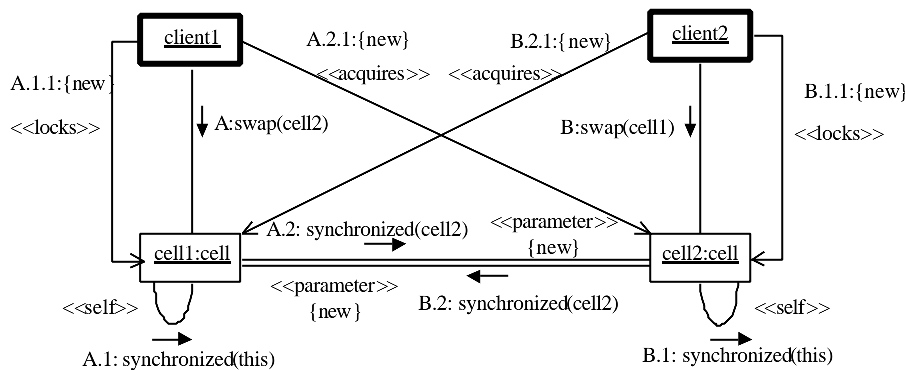

Abbildung 6 Kollaborationsdiagramm mit numerierten Links

Zeitweilig aktive und zeitweilig mehrfach aktive Objekte

Bisher haben bei der Visualisierung der Synchronisation die aktiven Objekte eine herausragende Stellung eingenommen. Ihnen wurden die Locks zugeordnet und sie sind es, die auf Locks warten. Dies resultiert aus der Abbildung der entsprechenden Laufzeitmechanismen von Java. In einer konkreten Deadlocksituation müssen die aktiven Objekte aber nicht immer von Interesse sein. Im Gegenteil, ihre Berücksichtigung kann die zu visualisierende Situation unnötig komplex machen. In diesem Abschnitt zeigen wir, wie man diese Komplexität reduzieren kann, indem man das Kollaborationsdiagramm auf den eigentlich relevanten Ausschnitt beschränkt. Dazu ersetzen wir das Vorkommen aktiver Objekte durch den neu zu definierenden *Stereotyp* der *zeitweilig (mehrfach) aktiven Objekte*.

Was aktive von passiven Objekten unterscheidet, ist der eigene *Kontrollfokus*, über den ein aktives Objekt verfügt. Im Laufe der Ausführung eines Programmes bewegt sich dieser Kontrollfokus bei jedem Methodenaufruf vom aufrufenden Objekt zum aufgerufenen. Das aufgerufene Objekt wird also zu einem *zeitweilig aktiven Objekt*, da es ja zu diesem Zeitpunkt über einen Kontrollfokus verfügt. Der kleinste relevante Ausschnitt eines Kollaborations-diagrammes zur Visualisierung synchronisierter Abläufe wird stets das aufgerufene Objekt enthalten. Zur Repräsentation des Threads genügt es daher, dieses aufgerufene Objekt mit Hilfe eines *Stereotyps* zu einem zeitweilig aktiven Objekt zu machen. Als graphische Darstellung dieses Stereotyps wäh-

len wir die dicke Umrandung des aktiven Objektes, jedoch in gestrichelter Form als Zeichen der Zeitweiligkeit.

Damit man sieht, welche Objekte von dem Thread gelockt sind, biegen wir die `<<locks>>` Links so um, daß sie jetzt von dem zeitweilig aktiven Objekt auf die gelockten Objekte zeigen. Ebenso hängen wir jetzt die `<<acquires>>` Links an die zeitweilig aktiven Objekte (siehe Abbildung 7).

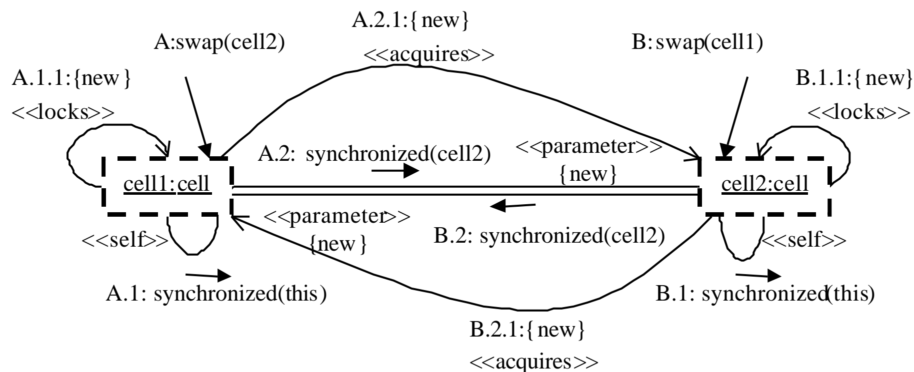

Abbildung 7 Kollaborationsdiagramme mit zeitweilig aktiven Objekten

Da alle passiven Objekte in einem Kollaborationsdiagramm zeitweilig aktiv sind, markieren wir nur Objekt als zeitweilig aktiv, in dem sich der Kontrollfokus zuletzt befindet. An dieses Objekt werden dann auch die `<<locks>>` Links gehängt.

In einem nebenläufigen Kontext können sich die Kontrollfoki mehrerer Threads in einem Objekt befinden, sofern nicht alle Methoden als `sychronized` deklariert sind. Diese können dann gleichzeitig verschiedene oder sogar dieselbe Methode ausführen. Ein Objekt, das über mehrere Kontrollfoki verfügt, nennen wir *zeitweilig mehrfach aktives Objekt.* Dafür führen wir einen Stereotyp mit einer graphischen Darstellung ein. Ein solches Objekt erhält dieselbe gestrichelte Umrandung wie das zeitweilig aktive Objekt, mit dem Unterschied, daß für jeden weiteren Kontrollfokus, über den das Objekt verfügt, ein weiterer gestrichelter Rahmen versetzt hinter dem Objekt gezeichnet wird. Bei mehrfach aktiven Objekten hängen wir die während der Synchronisation entstehenden Links wie `<<locks>>` und `<<acquires>>` nicht alle an die Umrandung des Objektes, sondern nur die für den ersten Kontrollfokus. Für alle weiteren Kontrollfoki werden die jeweils unter einem Fokus bei der Synchronisation entstandenen Links an den jeweiligen versetzten Rahmen gehängt. Aus Platzgründen verzichten wir auf ein Beispiel und verweisen den Leser auf [9].

Zur Demonstration unserer Erweiterungen haben wir hier ein sehr einfaches Beispiel verwendet. Die Gefahr für einen Deadlock bei der Klasse `cell` läßt sich für einen erfahrenen Programmierer leicht erkennen. Der Deadlock kann vermieden werden, indem man eine Reihenfolge für angeforderte Ressourcen festlegt [8]. In realistischeren Beispielen sind Deadlocks jedoch nicht auf eine einzige Klasse oder sogar eine einzige Methode beschränkt, sondern entstehen im Zusammenspiel verschiedener Methoden und Klassen. Gerade dann ist es wichtig, eine visuelle Repäsentation der beteiligten Objekte und Aufrufe zu haben.

5 Visualisierung von Thread-Koordination

Zur Visualisierung von Programmabläufen, in denen `wait`- und `notify`-Methoden aufgerufen werden, können ebenfalls Sequenzdiagramme eingesetzt werden. Fehler wie schlafende Threads erkennt man ähnlich wie bei Deadlocksituationen daran, daß irgendwann kein Thread mehr im Sequenzdiagramm aktiv ist. Aus Platzgründen können wir hier keine Beispiele angeben.

Allerdings unterscheidet sich die Analyse von Dormancy-Probleme von Deadlocks. In Deadlocksituationen benötigen wir exakte Informationen über die Beziehungen, da zyklische Beziehungen der Grund für einen Deadlock sein können. Im Falle fehlender `notify`- Aufrufe läßt aber nicht eindeutig sagen, warum diese Aufrufe fehlen oder wo sie hingehören. Diese Probleme erfordern eine umfassendere Analyse und können auch nicht basierend auf einem einzeln Programm-Trace erfolgen. Die Analyse muß sich damit befassen, welche Bedingungen zum Warten von Threads führen, wann sich Bedingungen ändern, und wer über die Änderungen von Bedingungen benachrichtigen kann und wer benachrichtigt werden muß. Dazu benötigt man die vorher erwähnte *statische Analyse*, wie z.B. das Programm Slicing (siehe Abschnitt 7). Zur Visualisierung reichen dann ebenfalls nicht instanzbezogene UML-Diagramme aus wie die Interaktionsdiagramme. Daher beabsichtigen wir, zur Beschreibung des Verhaltens auf einer höheren Abstraktionsebene UML-Statecharts einzusetzen.

6 Werkzeugunterstützung

Während für einfache Beispiele, wie hier vorgestellt, die UML-Diagramme von Hand erstellt werden können, ist dies für komplexere Beispiel nicht möglich, sondern muß automatisiert werden. Im Zentrum der von uns angedachten Werkzeugunterstützung soll jedoch nicht die Erzeugung von Programm-Traces liegen, sondern eine *Visualisierungskomponente*, deren Hauptfunktionalität darin bestehen soll, aus Programm-Traces UML-Diagramme zu erzeugen und diese um die Darstellung der Synchronisation anzureichern.

Diese Komponente sollte daher eine offene Schnittstelle für die Weiterverarbeitung von Traces aus existierenden Tools haben. Denkbar wäre, hier XML als Datenaustauschformat zugrunde zu legen. Existierende Tools erzeugen bisher aber noch keine XML-Formate, so daß zusätzlich Transformatoren erstellt werden müssen, um proprietäre Formate nach XML zu übersetzen.

Die Darstellung der entstehenden UML-Diagramme sollte mit einem existierenden UML-Casetool erzeugt werden wie z.B. Rational Rose [11]. Die darin enthaltenen Editoren unterstützen auch selbstdefinierte Stereotypen. UML-Diagramme können in Rational Rose über eine XML-Schnittstelle eingelesen werden. Demnach sollen die von der Visualisierungkomponente erzeugten UML-Diagramme auf dieser Schnittstellendefinition basieren.

7 Verwandte Arbeiten

Unser Ziel, Programmabläufe zu visualisieren, um Fehler finden zu können, steht in Bezug zum sogenannten *Debugging*. Während gewöhnliche Debugger dazu dienen, den Zustand eines Programms anzuzeigen oder fortgeschrittenere Werkzeuge auch Programm-Traces in textueller Form erzeugen können, liegt der Vorteil von unserem Ansatz darin, daß wir eine graphische Sprache zur Visualisierung von Traces verwenden. Darüber hinaus setzen wir für die verschiedenen Aspekte, unter denen man Traces bei der Fehlersuche betrachten will, verschiedene Diagramme ein. Während Sequenzdiagramme besser geeignet sind zeitliche Zusammenhänge darzustellen, können Kollaborationsdiagramme besser strukturelle Beziehungen darstellen. Außerdem gehen wir über gewöhnliche Ansätze des Debuggings hinaus, da wir die Laufzeitmechanismen darstellen. [2] beschäftigt sich speziell mit der Darstellung der Synchronisation in Programm-Traces, allerdings auch nur in textueller Form. Hier werden die für die Synchronisations relevanten Ereignisse aus Programm-Traces herausgefiltert. In [7] wird das Debugging paralleler (und damit nebenläufiger) Programme durch Eventgraphen unterstützt. Dieser Ansatz stellt gegenüber unserem schon einen weiteren Abstraktionsschritt dar, da er die beteiligten Objekte nicht mehr visualisiert.

Unser Ansatz ist auch verwandt mit dem Gebiet des *Programmverstehens. Statisches* Programmverstehen arbeitet auf dem Quelltext. Eine Technik dafür ist beispielweise das *Program Slicing* [5]. Die Grundidee besteht darin, in einem Programm alle Anweisungen zu bestimmen, die den Wert einer zuvor bestimmten Variablen ändern. In [15] wird eine Erweiterung bestehender Slicing-Techniken unter besonderer Berücksichtigung der Nebenläufigkeit in Java-Programmen vorgestellt. *Dynamisches* Programmverstehen verwendet zusätzlich Informationen aus Programm-Traces. Darunter gibt es bereits Ansätze, die graphische Sprachen verwenden, wie z.B. Message Sequence Charts in dem Werkzeug Jinsight [5]. Hier werden jedoch keine Laufzeitmechanismen visualisiert.

Beim dynamischen *Reverse Engineering* verwendet man die Information aus verschiedenen Programm-Traces dazu, um generelle Verhaltensweisen des Programmes zu inferieren, z. B. in Form von State Charts [12]. Die bisherigen Ansätze beschäftigen sich jedoch noch nicht explizit mit Synchronisation. Außerdem liegt der Fokus darauf, lauffähige Programme zu verstehen, nicht jedoch darauf, sich mit Fehlersituationen zu beschäftigen.

8 Zusammenfassung und Ausblick

Wir haben einen Ansatz vorgestellt, der Entwickler von nebenläufigen Programmen bei der Fehlersuche durch visuelle Repräsentation von Programmabläufen unterstützen soll. Dazu verwenden wir UML, eine weitverbreitete Sprache zur objektorientierten Modellierung. Zusätzlich zur normalen Verwendung von Programm-Traces visualisieren wir die Laufzeitmechanismen von Java bei der Synchronisation. Wir setzen dafür verschiedene Diagramme ein, um verschiedene Aspekte zu unterstützen. Während Sequenzdiagramme zur Visualisierung des zeitlichen Ablaufes geeignet sind, ergänzen Kollaborationsdiagramme die Dimension struktureller Beziehungen.

Zur Abbildung der Laufzeitbeziehungen wurden neue Stereotypen eingeführt. Während man mit diesen zwei Diagrammarten Probleme wie Deadlocks gut erkennen kann, muß noch eine geeignete Unterstützung für die Darstellung von Dormancy - Problemen gefunden werden.

Wir haben skizziert, wie wir uns eine Werkzeugunterstützung vorstellen. Dabei ist es wichtig, die Visualisierungkomponente sowohl von einem konkreten UML-Tool unabhängig zu machen, wie auch von Tools zur Generierung der Programm-Traces. Hierzu existieren bereits ausgereifte Werkzeuge, die auf das Testen oder das Model-checking von nebenläufigen Programmen spezialisiert sind. Unser Schwerpunkt soll auch in Zukunft darauf liegen, zu untersuchen, wie diese Werkzeuge an UML anzubinden sind und inwieweit die Visualisierung mit UML die Analyse von Programm-Traces verbessern kann.

Literatur

[1] A. Bechini, K. Tai, Timestamps for Programs Using Messages and Shared Variables, *Proc. of the 18th International Conference on Distributed Computing Systems*, 1998.

[2] A. Bechini, K.Tai, Design of a Toolset for Dynamic Analysis of Concurrent Java Programs, *Proc. of the 6th International Workshop on Program Comprehension IWPC'98*. 1998.

[3] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*, Reading, MA. Addison-Wesley, 1998.

[4] J. Gosling, B. Joy, G. Steel, *The Java Language Specification*, Addison-Wesley, 1996.

[5] S. Horwitz, T. Reps, D. Binkley, Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems 12, 1 (January 1990), 26-60.

[6] Jinsight (http://www.research.ibm.com/jinsight), IBM Research 1998.

[7] D. Kranzlmüller, S. Grabner, J. Volkert, Debugging with the MAD Environment, in J. Dongarra, B. Tourancheau (Eds.), *Environments and Tools for Parallel Scientific Computing*, Vol23, Nos. 1-2, April 1997.

[8] D. Lea, *Concurrent Programming in Java*, Addison-Wesley, 1997.

[9] K. Mehner, A. Wagner, Visualisierung der Synchronisation von Java-Threads mit UML, *Workshop Modellierung*, 2000.

[10] M. Pancake, Visualization Techniques for Parallel Debugging and Performance-Tuning Tools, in A.Y. Zomaya, *Parallel Computing: Paradigms and Applications*, Intl. Thomson Computer Press, 1996.

[11] Rational Rose 98, (http://www.rational.com), Rational Inc. 1998.

[12] T. Systa, On the Relationship between Static and Dynamic Models in Reverse Engineering Java Software, *Proc. of the 6th Working Conference on Reverse Engineering WCRE'99*, Atlanta, 1999.

[13] K. Tai, Race Analysis of Traces of Asynchronous Message-Passing Programs, *Proc. of the 17th International Conference on Distributed Computing Systems*, 1997.

[14] UML Specification, Version 1.3. The Object Management Group, Doc. Ad/99-06-08. June 1999.

[15] J. Zhao, Slicing Concurrent Java Programs, *Proc. of the 7th International Workshop on Program Comprehension IWPC'99*, 1999.

[16] J. Zhao, K. Tai, Deadlock Analysis of Synchronous Message-Passing Programs, *International Symposium on Software Engineering for Parallel and Distributed Systems PDSE'99*, Los Angeles 1999.