Comparison of the Concurrency Concepts of Ada, CHILL, Erlang, and Java - Case Studies 2

Frank Ecke, Diplomarbeit Friedrich-Schiller-Universität Jena, Juli 1999
www1.informatik.uni-jena.de/themen/pap-talk/da-fe.pdf

Ziel der Arbeit, die eine Fortsetzung und Erweiterung von [BE 98] darstellt, ist es, durch Fallstudien den Vergleich der Nebenläufigkeitsaspekte von vier Programmiersprachen - Ada, CHILL, Erlang und Java - zu erweitern und zu vertiefen. Die Beispiele sind so gewählt, dass sie ein möglichst breites Feld abdecken. Es zeigt sich, dass - trotz der Vielfalt der von den Sprachen gebotenen Ausdrucksmittel - keine Sprache allmächtig ist. In den Beispielen, bei denen Kommunikation eine wichtige Rolle spielt, tritt die Asynchronität, die in CHILL und Erlang zu finden ist, negativ auf. Sie muss unterbunden werden, was natürlich zusätzlichen Entwicklungs- und Programmieraufwand bedeutet. Das Nebenläufigkeitsmodel von Java beispielsweise erlaubt keine direkte Kommunikation zwischen Akteuren. Ada, eigentlich für Echtzeitsysteme entwickelt, ist gerade in diesem Bereich nicht überzeugend.

Eine Standardaufgabe beim nebenläufigen Programmieren ist es, abzusichern, dass gegenseitiger Ausschluss beim Zugriff auf kritische Ressourcen gewährleistet ist. Um dies zu erreichen, gibt es im wesentlichen zwei Möglichkeiten: einen Ressourcenmanager, der Anrufe von der Aussenwelt entgegenimmt und sicherstellt, dass immer nur maximal ein Anruf bearbeitet wird; oder ein passives Konstrukt, für das, um es zu verwenden, vorher eine exklusive und nur einmal vorhandene Berechtigung erworben werden muss. Beide Methoden haben ihre Anwendungsgebiete, weswegen eine Programmiersprache, die Nebenläufigkeit unterstützt, auch beide anbieten sollte. Im Falle von Erlang finden wir aber nur die aktive Lösung. Abstraktionsinversion ist die Folge - Dinge, die an sich passiv sind, müssen mit aktiven Komponenten realisiert werden. Dies ist sehr schön sichtbar beim Problem der speisenden Philosophen, bei denen die Gabeln eine passive kritische Ressource darstellen. Eine Auswirkung in diesem Fall ist, dass möglicherweise erhöhter Verwaltungsaufwand zu Laufzeit entsteht. Bereits erwähnt wurde das Fehlen von direkter Kommunikation in Java, was die Entwicklung eines Ressourcenmanagers hier erschwert.

Kommunikation ist oftmals ein zentrales Thema einer Anwendung. Tritt die Kommunikation mit mehreren Partnern auf, so ist es häufig wünschenswert (ja sogar notwendig), dass sie in einer bestimmten Reihenfolge abläuft; z.B. sollte bei der Ressourcenverteilung so verfahren werden, dass sich die zeitliche Vergabe der Ressourcen in der zeitliche Ordnung der Anforderungen widerspiegelt oder dass Prioritäten beachtet werden usw. Dies setzt jedoch voraus, dass die Menge der Wartenden geordnet ist. Leider finden wir das nur in Ada und Erlang. Wird ein wartender Akteur in CHILL oder Java aufgeweckt, so heisst dies nicht notwendigerweise, dass die Ressource, auf die er gewartet hat, auch tatsächlich verfügbar ist. Falls nicht, so wurde der falsche Akteur geweckt, und er muss wieder in die Wartemenge eingebracht werden. All dies ist händisch auszuführen, d.h., erfordert zusätzlichen Aufwand. Niemand verbietet dem Laufzeitsystem von CHILL oder Java jedoch, beim nächsten Male wieder den (oder einen) falschen Akteur aufzuwecken usw. Wir nennen das eine abgeschwächte Form von busy waiting und wissen, dass es nicht gut ist. Eine Lösung des Problems ist nicht einfach; es ist offenbar stets Aufwand zu betreiben, der zum Prüfen der Bedingung nämlich. Verringert werden kann dieser Aufwand allerdings, wenn wir uns überlegen, wer ihn wann am besten betreibt und das Laufzeitsystem entsprechend bauen. Wenn alle Akteure für sich selbst prüfen, und dann notwendigerweise - nachdem sie aufgeweckt wurden, so bedeutet das zusätzliche Prozesswechsel, die vermieden werden könnten, würden wir dem Laufzeitsystem die Aufgabe des Prüfens übertragen. In Ada und Erlang funktioniert die Sache so - es wird zuerst geprüft, ob die gewünschte Ressource verfügbar ist, bevor (und nur dann) der Akteur geweckt wird. Die Passivkonstruktion im vorangegangenen Satz soll ausdrücken, dass dies vom Laufzeitsystem erledigt wird, wir geben nur die gewünschte Ressource/Bedingung an.

Zwei der Fallstudien befassen sich mit Spezialanwendungen der nebenläufigen Programmierung: Interruptbehandlung und zeitliches Einplanen von Prozessen. Interruptbehandlung tritt oftmals im Zusammenhang mit eingebetteten Systemen auf und stellt die Programmierer vor einige Probleme. Nicht nur sind diese hardwarespezifisch (es müssen im allgemeinen die Besonderheiten der zugrundeliegenden Hardware berücksichtigt werden), sondern auch die Natur von Interruptbehandlungsroutinen unterscheidet sich bei genauerem Hinsehen von der normaler Unterprogramme. Die Aufrufmodelle sind unter Umständen verschieden, und nicht jede Anweisung ist geeignet innerhalb einer solchen Routine. Von den vier untersuchten Sprachen ist nur eine, Ada, hilfreich auf diesem Gebiet. Dies ist zwar wahr, aber nicht ganz gerecht Java gegenüber. Das Anwendungsgebiet dieser Sprache ist nicht im Bereich der Echtzeit- oder eingebetteten Systeme zu suchen. Bei der Beliebtheit, der sich Java momentan erfreut, war es jedoch angebracht, sich auch einmal jenseits des Tellerrandes umzusehen. Und in der Tat, der Autor ist fündig geworden. PERC, ein Dialekt der

Sprache Java, bietet Mittel, um Interruptbehandlung in den Griff zu bekommen. Mehr noch, PERC ist die einzige Sprache in dem Quartett/Quintett, die uns beim zeitlichen Einplanen von Prozessen wirklich zur Hand geht. Normalerweise ist es nicht so wichtig, wann ein bestimmter Prozess/eine Menge von Prozessen ausgeführt wird. Die funktionale Korrektheit eines Programms darf davon nicht abhängig sein. Unglücklicherweise gibt es aber auch noch eine weitere Form der Korrektheit - die zeitliche. Wie gesagt, oftmals kann diese ignoriert werden, aber es gibt natürlich auch Situationen, in denen wir uns diesen Luxus nicht erlauben können: in Echtzeitsystemen beispielsweise. Hier ist ein zu spät abgeliefertes (genaues) Ergebnis schlimmer als ein rechtzeitig erbrachtes, das ggf. weniger genau ist. Beim Entwickeln von Echtzeitsystemen wird viel Zeit und Mühe darauf verwandt, sicherzustellen, dass die zeitliche Einplanung funktioniert. Für Prozesse werden sogenannte temporal scopes festgelegt, die angeben, wann ein Prozess zu starten ist, wie lange er laufen darf oder wie oft, wann er beendet sein muss, usw. Auf dem Papier sieht das alles sehr gut aus, irgendwann jedoch müssen die Prozesse aber einmal implementiert werden. Was hat die gewählte Programmiersprache zu bieten bezüglich der Spezifikation von temporal scopes? Fällt die Wahl auf Ada, CHILL, Erlang oder Java, so ist die Antwort ernüchternd: nichts! Das, was in der Theorie so schön funktioniert, muss jetzt mehr oder weniger mühsam in die Praxis umgesetzt werden. Die temporal scopes müssen simuliert werden. Niemand sieht den Prozessen mehr an, dass sie bestimmte zeitliche Rahmenbedingungen einzuhalten haben. Am wenigsten das Laufzeitsystem, was ja eigentlich diese Bedingungen überwachen soll. Licht am Ende des Tunnels verspricht PERC, das die Angabe von temporal scopes erlaubt und über Mechanismen verfügt, die auch deren Überwachung gewährleisten.

1 Betreuer: Prof. Dr. Jürgen F.H. Winkler, FSU Jena

2. Betreuer: Dr. Stefan Kauer, FSU Jena

BE 99 Brömel, Peter; Ecke, Frank: Description and Comparison of the Concurrency Concepts of Ada, Java, and CHILL. Studienarbeit, Friedrich-Schiller-Universität Jena, 13. Jan. 1999
www1.informatik.uni-jena.de/themen/pap-talk/studarb-pb-fe.htm