ulm university universität
uulm

Institut für Numerische Mathematik

Dr. Andreas F. Borchert

24. April 2015
Blatt 2

Parallele Programmierung mit C++ (SS 2015)

Abgabe bis zum 8. Mai 2015, 14:00 Uhr

Lernziele:

- Nichtdeterministische Prozesse mit CSP
- Dynamischer Polymorphismus in C++

Aufgabe 4: Unzuverlässiges Netzwerk

Anfragen an einen DNS-Server und dessen Antworten erfolgen normalerweise mit Hilfe von UDP-Paketen, die nicht unbedingt zuverlässig ankommen, d.h. sie können auch verloren gehen oder auch doppelt ankommen. Man greift hier normalerweise auf mehrere Versuche zurück, wenn es zu Timeouts kommt. Versuchen Sie dies in CSP zu modellieren mit

- Einen Server mit dem Alphabet { *receive_request*, *process_request*, *send_response* }, der jede eingehende Anfrage beantwortet und dem egal ist, ob seine Antwort ankommt oder nicht,
- einen Klienten mit dem Alphabet { *send_request*, *receive_response*, *process_response*, *timeout* }, wobei *timeout* das private Ereignis ist, das als nicht zeitgemäßes Eintreffen einer Antwort interpretiert wird, und
- einem Netzwerk mit dem Alphabet { *send_request*, *receive_request*, *send_response*, *receive_response* }.

Hierbei kann der Klient nach dem erfolgen Empfang der Antwort oder dem endgültigen Fehlschlagen (dreifacher Timeout) mit *STOP* beendet werden. Natürlich lässt sich auch ein schöneres Ende mit *SKIP* modellieren, aber dann müssen alle Prozesse sich daran beteiligen.

Versuchen Sie zunächst ein zuverlässiges Netzwerk zu implementieren. Aber setzen Sie dennoch bereits die Behandlung von *timeout* um. Das entspricht der realen Situation, dass auch nach einem Timeout durchaus noch die (reichlich späte) Antwort ankommen kann. Da Pakete unabhängig voneinander in beide Richtungen übertragen werden können, sollte

Ihr Netzwerk eine gewisse Kapazität besitzen, die die parallele Übertragung von mehr als einem Paket zulässt. Dies lässt sich am leichtesten mit dem |||-Operator erreichen.

Wenn es soweit klappt, können Sie den ⊓-Operator verwenden, um die unzuverlässige Übertragung ins Spiel zu bringen, so dass Pakete entweder ankommen, doppelt ankommen oder überhaupt nicht ankommen.

Wenn Sie das auf der Vorlesungsseite zur Verfügung gestellte CSP-Werkzeug verwenden, ist der ⊓-Operator in ASCII als „|~|“ darzustellen.

Ihre Lösung können Sie einreichen mit

```
submit pp 4 dnsreq.csp
```

Aufgabe 5: Abstrakte Spielereien

In der im ersten Übungsblatt vorgestellten und in der Aufgabe 3 umgesetzten Implementierung eines Spiels hängt der Spielablauf von der Art der Spieler ab. Bei menschlichen Spielern muss ein Zug interaktiv eingelesen und überprüft werden, bei einem rechnerseits implementierten Spieler muss anhand irgendwelcher Kriterien ein Zug gefunden werden. Wenn ein Computer gegen einen menschlichen Spieler zu spielen hat, führt dies zu Konstruktionen wie die folgende:

```
while (!game.finished()) {
   cout << "Noch sind es " << game.size_of_heap() <<
      " Staebchen." << endl;
   int move;
   if (game.next_player() == human) {
      move = fetch_move_from_human_player(game);
   } else {
      move = fetch_move_from_computer(game);
      cout << "Der Computer nimmt " << move <<
         " Staebchen." << endl;
   }
   game.perform_move(move);
}
```

Es wäre jedoch reizvoll, wenn die Organisation des Spielablaufs von der Art der Spieler unabhängig sein könnte.

Sie sollten deswegen eine abstrakte Basisklasse für Spieler definieren, deren Schnittstelle die Verwaltung eines Namens und die Bestimmung eines Spielzugs ermöglicht. (Der Name darf aber später nicht mehr verändert werden.) Zu dieser Basisklasse sollte es mindestens zwei abgeleitete Klassen geben, wovon eine Variante als Schnittstelle zu einem menschlichen Spieler dient und die andere vom Computer bestimmte Züge generiert.

Ferner ist eine allgemeine Spielprozedur zu implementieren, die als Parameter zwei Spieler und ein Spielobjekt erhält und dann eine Partie durchzieht. Diese Prozedur sollte auch jeweils den aktuellen Spielverlauf ausgeben und am Ende feststellen, wer gewonnen hat. Abschließend ist ein Hauptprogramm zu schreiben, dass zwei Spieler und eine Partie anlegt,

um diese dann der Spielprozedur zu übergeben. Sie können das in verschiedenen Varianten testen, etwa menschlicher Spieler gegen menschlichen Spieler, menschlicher Spieler gegen Computer oder auch zwei Computer-Lösungen gegeneinander antreten lassen.

Wenn Ihnen das Nim-Spiel als zu langweilig erscheint, steht es Ihnen in diesem Blatt auch ausdrücklich frei, ein beliebiges anderes Zwei-Personen-Spiel auszusuchen.

Einzureichen sind alle Quellen:

```
submit pp 5 Player.hpp Computer.cpp Computer.hpp \
   Human.cpp Human.hpp Nim.hpp Nim.cpp Game.hpp Game.cpp \
   PlayerVsComputer.cpp
```

Oder, falls Sie mit anderen zusammengearbeitet haben, zusätzlich mit Angabe einer *team*-Datei, in der die Loginnamen der Team-Mitglieder genannt werden (für jeden Loginnamen eine eigene Zeile):

```
submit pp 5 Player.hpp Computer.cpp Computer.hpp \
   Human.cpp Human.hpp Nim.hpp Nim.cpp Game.hpp Game.cpp \
   PlayerVsComputer.cpp team
```

Hierbei ist *Player.hpp* die abstrakte Klasse für einen Spieler, die ohne eine zugehörige Implementierung auskommt, *Computer.hpp* und *Computer.cpp* die davon abgeleitete Klasse für einen spielenden Computer mitsamt Implementierung, *Human.hpp* und *Human.cpp* die ebenfalls von *Player* abgeleitete Klasse, die als Schnittstelle für einen menschlichen Spieler operiert, *Nim.hpp* und *Nim.cpp* die Klasse und die Implementierung für das Spiel (kann von der Aufgabe 3 übernommen werden), *Game.hpp* und *Game.cpp* die Schnittstelle und die Organisation eines Spielverlaufs (muss nicht unbedingt als Klasse realisiert werden) und abschließend *PlayerVsComputer.cpp* als Beispiel für ein Hauptprogramm, das einen menschlichen Spieler gegen den Computer antreten lässt.

Falls Sie ein anderes Spiel auswählen, dann ist statt *Nim.hpp* der allgemeinere Name *GameStatus.hpp* und entsprechend statt *Nim.cpp* dann *GameStatus.cpp* zulässig.

Viel Erfolg!