



Parallele Programmierung mit C++ (SS 2017)

Abgabe bis zum 28. April 2017, 14:00 Uhr

Lernziele:

- Modellierung einfacher Prozesse mit CSP
- Implementierung und Benutzung einer Klasse in C++

Hinweise:

Melden Sie sich für diese Veranstaltung in SLC an. Ohne diese Anmeldung können Sie keine Lösungen elektronisch einreichen. Sobald Sie sich angemeldet haben, wird es bis zu einer Stunde brauchen, bis Sie anschließend Lösungen einreichen können.

Aufgabe 1: Getränkeautomat

Modellieren Sie mit Hilfe von CSP einen Getränkeautomat, der Tee zum Preis von 1 Euro und Kaffee zum Preis von 1,50 Euro anbietet. Zu unterstützende Münzen sind die für 50 Cent und 1 Euro, die in einer beliebigen Reihenfolge verwendet werden dürfen. Der Automat sollte beim Kaffee in der Lage sein, beim Bezahlen mit zwei einzelnen Euromünzen 50 Cent wieder herauszugeben. Die Rückgabe des Wechselgeldes erfolgt dabei vor der Ausgabe des Kaffee. Dabei sollte

$$\alpha \text{Automat} = \{muenze50, muenze100, rueckgabe_muenze50, tee, kaffee\}$$

gelten.

So könnte ein möglicher Verlauf aussehen:

```
thales$ trace -p automat.csp
Alphabet: {kaffee, muenze100, muenze50, rueckgabe_muenze50, tee}
Acceptable: {muenze100, muenze50}
muenze50
Acceptable: {muenze100, muenze50}
```

```

muenze100
Acceptable: {kaffee}
kaffee
Acceptable: {muenze100, muenze50}
muenze100
Acceptable: {muenze100, muenze50, tee}
muenze100
Acceptable: {rueckgabe_muenze50}
rueckgabe_muenze50
Acceptable: {kaffee}
kaffee
Acceptable: {muenze100, muenze50}
OK
thales$

```

Ergänzen Sie den Automaten mit einer Fehlfunktion, bei der vier 50-Cent-Münzen in einer Reihe akzeptiert werden, dann aber der Automat seine Funktion einstellt:

```

thales$ echo muenze50 muenze50 muenze50 muenze50 | trace -p fehlfunktion.csp
Alphabet: {kaffee, muenze100, muenze50, rueckgabe_muenze50, tee}
Acceptable: {muenze100, muenze50}
Acceptable: {muenze100, muenze50}
Acceptable: {muenze50, tee}
Acceptable: {kaffee, muenze50}
Acceptable: {}
OK
thales$

```

Testen Sie Ihre Lösung interaktiv mit unserer CSP-Implementierung und reichen Sie Ihre Lösungen auf der Thales ein mit folgendem Kommando:

```
submit pp 1 automat.csp fehlfunktion.csp
```

Hierbei sollte *automat.csp* der funktionierende Automat sein und *fehlfunktion.csp* derjenige mit Fehlfunktion.

Aufgabe 2: Nim-Spiel

Nim ist ein Spiel für zwei Personen, bei dem auf n Haufen jeweils s_i Stäbchen verteilt sind. Die beiden Spieler ziehen alternierend und ein Spielzug besteht darin, 1 bis m Stäbchen von genau einem Haufen zu entfernen. Findet ein Spieler keine Stäbchen mehr zum Entfernen vor, hat er verloren.

Laden Sie für diese Aufgabe *Nim.tar.gz* von der Vorlesungswebseite herunter und packen Sie das Archiv mit Hilfe von *tar* aus. So könnte das funktionieren:

```

thales$ mkdir nim
thales$ cd nim
thales$ wget -O- -q \
> http://www.mathematik.uni-ulm.de/numerik/pp/ss17/uebungen/01/Nim.tar.gz |
> gunzip | tar xf -
thales$ ls
Makefile Nim.cpp NimGame.cpp NimGame.hpp NimMove.cpp NimMove.hpp
thales$

```

Sie finden dann eine halb fertige Umsetzung des Spiels vor. Die Klasse *NimMove* repräsentiert einen Spielzug, *NimGame* enthält die Datenstruktur und alle Methoden, um ein Nim-Spiel durchzuführen, und in *Nim.cpp* finden Sie ein darauf basierendes Hauptprogramm, das eine Partie zwischen einem menschlichen und einem Computer-Spieler durchführt. Überall, wo Sie in den Quellen die Zeichenkette *FIXME* vorfinden, ist etwas zu ergänzen. Das betrifft die Klasse *NimGame*, bei der die Mehrheit der Methoden umzusetzen ist, und den Teil des Hauptprogramms, bei der der Computer einen Spielzug aussucht, wenn er die Chance hat, zu gewinnen.

Für Nim-Spiele gibt es eine auf Patrick M. Grundy und Roland P. Sprague zurückgehende Theorie, die bei Nim-Spielen jeder Spielsituation eine nicht-negative sogenannte Nim-Zahl zuordnet. Wenn der Wert 0 ist, dann hat der am Zug befindliche Spieler keine sichere Gewinnstrategie mehr. Bei einem positiven Nim-Wert gibt es mindestens einen Spielzug, der zum Gewinn führt. Bei dem vorgestellten Spiel werden zunächst die Nim-Werte für die einzelnen Haufen bestimmt und dann per Nim-Addition zu einem Wert aggregiert. Der Nim-Wert für einen einzelnen Haufen ist die Zahl der verbliebenen Stäbchen modulo der um eins erhöhten maximalen Zahl von Stäbchen, die in einem Zug entfernt werden dürfen (also $m + 1$). Die Nim-Addition entspricht dem bitweisen Exklusiv-Oder (also dem Operator „ \wedge “ in C++).

Beispiel: Angenommen, wir haben drei Haufen mit 9, 11 und 7 Stäbchen und es können maximal drei Stäbchen entfernt werden. Dann sind die Nim-Werte für die einzelnen Haufen 1, 3 und 3 und „ $1 \wedge 3 \wedge 3$ “ ergibt 1. In dieser Situation ist die Entfernung eines Stäbchens vom ersten Haufen ein guter Zug, da sich dann bei 8, 11 und 7 ein Nim-Wert von 0 ergibt.

Es ist empfehlenswert, mit der Fertigstellung von *NimGame.cpp* zu beginnen und dabei jeweils auf die Kommentare und die genannten Vorbedingungen in der zugehörigen Header-Datei *NimGame.hpp* Rücksicht zu nehmen. Mit `#include <cassert>` steht bereits *assert* zur Verfügung und davon sollte auch entsprechend Gebrauch gemacht werden. Bei der bereits implementierten Methode *set_heap_size* sehen Sie, wie auch ein übergebener Index mit *assert* überprüft werden kann. Bei der Methode *get_heap_size* können Sie das analog umsetzen.

Zur Datenstruktur eines *NimGame*-Objekts gehört die Spielkonfiguration *number_of_heaps* (Zahl der Haufen) und *maxtake* (maximale Zahl der Stäbchen, die in einem Zug entfernt werden dürfen. 0 bedeutet hier, dass beliebig viele von einem Haufen genommen werden dürfen). In dem Vektor *heap_size* wird die Zahl der verbliebenen Stäbchen in jedem der Haufen verwaltet. Es wird hierzu die STL-Container-Klasse *vector* benutzt, die ähnlich wie ein Array benutzbar ist. Beim bereits vorhandenen Konstruktor wird der Vektor passend dimensioniert. Die Variable *next_player* gibt jeweils an, wer als nächster Spieler am Zug ist. Am Ende eines Spieles handelt es sich dabei um den Verlierer.

Abgesehen von *NimGame.cpp* ist nur noch beim Hauptprogramm *Nim.cpp* der Teil zu ergänzen, der einen gewinnenden Zug ermittelt, wenn es einen solchen gibt. Hierzu können Sie systematisch alle zulässigen Züge auswerten, bis sie einen finden, der zu einem Nim-Wert von 0 führt. Den Test können Sie durchführen, indem Sie einfach das Spielobjekt kopieren und dann den zu testenden Zug auf der Kopie durchführen:

```
NimGame test = game; // make a copy of the current game
NimMove testmove(heap_index, count); // create a move
test.execute_move(testmove); // execute it
if (test.nim_value() == 0) {
    // winning move found ...
}
```

Wenn Sie aus mehreren zulässigen Zügen einen pseudo-zufällig aussuchen wollen, kommt Ihnen die Klasse *NimMoveSelector* aus *NimMoveSelector.hpp* entgegen.

So könnte eine erfolgreiche Partie aussehen:

```
thales$ Nim
*** Game of Nim ***
Number of heaps: 3
Maximal number of sticks that can be taken in one move: 3
Heaps: 10 11 11
Your move: 0 2
Heaps: 8 11 11
Taking 1 from heap 0
Heaps: 7 11 11
Your move: 0 3
Heaps: 4 11 11
Taking 1 from heap 0
Heaps: 3 11 11
Your move: 0 3
Heaps: 0 11 11
Taking 1 from heap 1
Heaps: 0 10 11
Your move: 1 3
Heaps: 0 7 11
Taking 1 from heap 1
Heaps: 0 6 11
Your move: 1 3
Heaps: 0 3 11
Taking 1 from heap 1
Heaps: 0 2 11
Your move: 2 1
Heaps: 0 2 10
Taking 1 from heap 1
Heaps: 0 1 10
```

```
Your move: 2 1
Heaps: 0 1 9
Taking 1 from heap 1
Heaps: 0 0 9
Your move: 2 1
Heaps: 0 0 8
Taking 1 from heap 2
Heaps: 0 0 7
Your move: 2 3
Heaps: 0 0 4
Taking 1 from heap 2
Heaps: 0 0 3
Your move: 2 3
Congratulations!
thales$
```

Sie können Ihre Lösung wieder mit *submit* einreichen:

```
thales$ submit pp 2 Nim.cpp NimGame.cpp
```

Es steht Ihnen aber auch frei, die anderen Teile zu verändern und in aktualisierter Form mit einzureichen.

Viel Erfolg!