Technische Universität München
Institut für Informatik
Prof. Dr. Helmut Veith
Andreas Holzer, M. Sc.

SS 2007
Übungsblatt 8
Sonntag, 8. Juli 2007

Übung zur Vorlesung Model Checking

BLAST 2.0

Use BLAST 2.0 to solve the following exercises. BLAST 2.0 can be obtained from `http://mtc.epfl.ch/software-tools/blast/`. Use the documentation provided on the website to solve the following exercises.

(a) *Using Assumptions in Blast.* Use Blast for exercise a) ii) from sheet 7 (20. Juni 2007).

(b) *Checking correct API usage.* Add to `struct train` and `struct ticket` observer variables and insert into the code assignments, checks and assertions of these variables (maybe with some enclosing `if`) to ensure correct usage of the Train API as specified in the given file `train.h`. Show that the code in `main.c` violates the API usage rules.

```
// train.h ------------------------------------
#ifndef TRAIN_H
#define TRAIN_H

// dummy structure definitions
struct train {
};

typedef struct train* Train;

struct ticket {
};

typedef struct ticket* Ticket;

// A train object is allocated
Train allocateTrain(char* target);

// A train object gets deallocated,
// but only if it was allocated before.
void deallocateTrain(Train t);

// A ticket object is returned corresponding
// to the given train object. The train object
// has to be allocated before.
// The ticket is then valid for one travel.
Ticket purchaseTicket(Train train);

// Invalidates a given ticket. This function
// has to be called with corresponding ticket
// and train objects.
void travel(Ticket ticket, Train train);

#endif

// train.c --------------------------------------
#include <stdlib.h>
#include <assert.h>
#include "train.h"

Train allocateTrain(char* target) {
  // stub that simulates creation of a train
  Train train;

  return train;
}

void deallocateTrain(Train train) {
  // stub that simulates the deallocation
  // of a train
  free(train);
}

Ticket purchaseTicket(Train train) {
  // stub that simulates creation of a new
  // ticket corresponding to the train
  Ticket ticket;

  return ticket;
}

void travel(Ticket ticket, Train train) {
  // here the ticket gets invalidated
}


// main.c ---------------------------------------
#include "train.h"

void main() {
  Train trainToBerlin = allocateTrain("Berlin");
  Train trainToMunich = allocateTrain("Munich");
  Ticket ticket = purchaseTicket(trainToBerlin);

  travel(ticket, trainToBerlin);
  travel(ticket, trainToBerlin);
  travel(ticket, trainToMunich);

  deallocateTrain(trainToBerlin);
}
```

(c) *Using the specification language of Blast.* Create a specification in the Blast Specification language to falsify the following code in `main.c`. You have to implement some global variable in the specification that checks, whether index bounds get violated. Modify you variable corresponding to the `buffIndex` variable in the given program.

```
// main.h -----------------------------------------
#ifndef MAIN_H
#define MAIN_H

void write(int c);
int read();
void init();

#define MAX_BUFF_SIZE 100

#endif

// main.c -----------------------------------------
#include "main.h"

int __BLAST_NONDET;

int buffIndex = 0;
int buffer[MAX_BUFF_SIZE];

void init() {
  // some init stuff ...
}

void write(int c) {
  buffer[buffIndex++] = c;
}

int read() {
  buffIndex--;
}

void server() {
  write(__BLAST_NONDET);
}

void client() {
  read();
}

void main() {
  while (1) {
    // server and client work "in parallel"
    if (__BLAST_NONDET) {
      server();
    }
    else {
      client();
    }
  }
}
```

(d) *Proving Loop Termination with Blast.* In this exercise you should show that the loop in the given program terminates. It is not sufficient to show that there is a program execution that leaves the loop (what could be shown by placing an `assert(0)` after the loop). Instead, you have to show that every program execution eventually leaves the loop (a liveness property!). In order to solve this exercise you have to add some code to the given program. One way to prove a loop is terminating is to show that the transition relation of the program is well-founded, i.e., there is now infinitely descending chain. There are automated approaches to do this (of course, the problem remains undecideable in general). In the following simple example we learn how safety model checkers can be used for such purposes. The following steps are necessary:

i) Find a ranking function for the loop in the program. A ranking function assigns a value to a program state. This value decreases with every successing program state. Furthermore, there is a least value. So, every chain of program states is well-founded with resprect to this ranking function. Afterwards, turn this function into a ranking relation. This means, define a relation that relates two different program states. In this example it suffices to consider only the value of the variable x at different program states.

ii) Introduce an additional variable that records the value of x at some point in the program run.

iii) Inside the loop you have to record the value of x at some point in the program run (therefor the introduction of a nondeterministic `if` is helpful).

iv) After the old value of x is recorded you have to check if the old value of x and the current value of x are always contained in your ranking relation. If this is the case and your ranking relation is well-founded, then the loop will always terminate.

If you are interested in learning more about techniques to show automatically program termination (especially the one presented here) you may be interested in looking at the paper of Cook et al. [1].

```
// main.c
#include <assert.h>

unsigned int __BLAST_NONDET;

void main() {
  unsigned int x = __BLAST_NONDET;

  while (x > 1) {
    x = x - 10;
  }
}
```

References

[1] B. Cook, A. Podelski and A. Rybalchenko. *Termination Proofs for Systems Code*. ACM SIGPLAN 2006.