Outline

General

- All information on the Web page
  `http://www.voronkov.com/lics.cgi`.
- Assessment: exam (80%), exercises (20%).
- Exercises: at the end of (almost) every week with the deadline in one week.

Computer Systems and Correctness

Suppose we design a (complex) system, which may contain various components, for example, sensors, networks, computers. All of these components are using software.

Computer Systems and Correctness

Suppose we design a (complex) system, which may contain various components, for example, sensors, networks, computers. All of these components are using software.

We have requirements on how the system should function, for example safety, reliability, security, availability, absence of deadlocks etc.

Computer Systems and Correctness

Suppose we design a (complex) system, which may contain various components, for example, sensors, networks, computers. All of these components are using software.

We have requirements on how the system should function, for example safety, reliability, security, availability, absence of deadlocks etc.

How can one ensure that the system satisfies these requirements?

Computer Systems and Correctness

Suppose we design a (complex) system, which may contain various components, for example, sensors, networks, computers. All of these components are using software.

We have requirements on how the system should function, for example safety, reliability, security, availability, absence of deadlocks etc.

How can one ensure that the system satisfies these requirements?

Modern computer systems are unreliable.

Small Example: Software

Consider the following fragment of a C program:

```c
int* allocateArray(int length)
{
  int i;
  int* array;
  array = malloc(sizeof(int)*length);

  for (i = 0;i <= length;i++)
    array[i] = 0;
  return array;
}
```

Is this program correct?

Small Example: Software

Consider the following fragment of a C program:

```
int* allocateArray(int length)
{
  int i;
  int* array;
  array = malloc(sizeof(int)*length);

  for (i = 0;i <= length;i++)
    array[i] = 0;
  return array;
}
```

Is this program correct?

Hardly: it writes into memory that has not been allocated.

Small Example: Software

Consider the following fragment of a C program:

```
int* allocateArray(int length)
{
  int i;
  int* array;
  array = malloc(sizeof(int)*length);

  for (i = 0;i < length;i++)
    array[i] = 0;
  return array;
}
```

Is this program correct?

Small Example: Software

Consider the following fragment of a C program:

```
int* allocateArray(int length)
{
  int i;
  int* array;
  array = malloc(sizeof(int)*length); // may return 0!

  for (i = 0;i < length;i++)
    array[i] = 0;
  return array;
}
```

Is this program correct?
No: it may write to the null address.

Small Example: Software

Consider the following fragment of a C program:

```
int* allocateArray(int length)
{
  int i;
  int* array;
  array = malloc(sizeof(int)*length);
  if (!array) return 0;
  for (i = 0;i < length;i++)
    array[i] = 0;
  return array;
}
```

Is this program correct?

Small Example: Software

Consider the following fragment of a C program:

```
/* Returns a new array of integers of a given
 length initialised by a non-zero value */
int* allocateArray(int length)
{
  int i;
  int* array;
  array = malloc(sizeof(int)*length);
  if (!array) return 0;
  for (i = 0;i < length;i++)
    array[i] = 0;
  return array;
}
```

Is this program correct?
No: it initialises the array by zeros

Small Example: Software

Consider the following fragment of a C program:

```
/* Returns a new array of integers of a given
 length initialised by a non-zero value */
int* allocateArray(int length)
{
  int i;
  int* array;
  array = malloc(sizeof(int)*length);
  if (!array) return 0;
  for (i = 0;i < length;i++)
    array[i] = 0;
  return array;
}
```

Is this program correct?
We discussed correctness of a program without ever defining what it means.

Small Example: Software

Consider the following fragment of a C program:

```
/* Returns a new array of integers of a given
 length initialised by a non-zero value */
int* allocateArray(int length)
{
  int i;
  int* array;
  array = malloc(sizeof(int)*length);
  if (!array) return 0;
  for (i = 0;i < length;i++)
    array[i] = 0;
  return array;
}
```

Is this program correct?
We discussed correctness of a program without ever defining what it means.
So what is correctness?

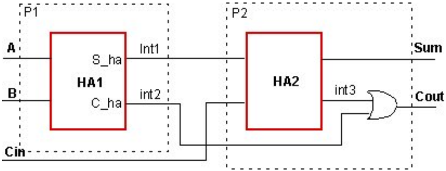
Notes

- We could spot the first two errors without knowing anything about the intended meaning of the program. But we had to understand the meaning of C programs in general and some specific properties of programming in C.

Notes

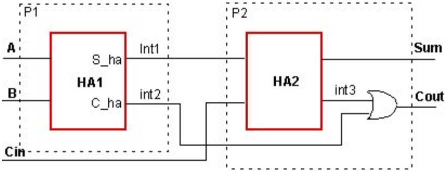
- We could spot the first two errors without knowing anything about the intended meaning of the program. But we had to understand the meaning of C programs in general and some specific properties of programming in C.
- To understand the last “error” we had to know something about the the intended behaviour of the program.

Example: Circuit Design



We used a circuit C_1 in a processor and would like to replace it by another circuit C_2. For example, we may believe that the use of C_2 results in a lower energy consumption.

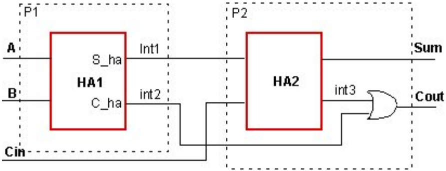
Example: Circuit Design



We used a circuit C_1 in a processor and would like to replace it by another circuit C_2. For example, we may believe that the use of C_2 results in a lower energy consumption.

We want to be sure that C_2 is correct, that is, it will behave according to some specification.

Example: Circuit Design



We used a circuit C_1 in a processor and would like to replace it by another circuit C_2. For example, we may believe that the use of C_2 results in a lower energy consumption.

We want to be sure that C_2 is correct, that is, it will behave according to some specification.

If we know that C_1 is correct, it is sufficient to prove that C_2 is functionally equivalent to C_1.

Another Example: Vending Machine

1. The vending machine contains a drink storage, a coin slot, and a drink dispenser. The drink storage stores drinks of two kinds: beer and coffee. We are only interested in whether a particular kind of drink is currently being stored or not, but not interested in the amount of it.
2. The coin slot can accommodate up to three coins.
3. The drink dispenser can store at most one drink. If it contains a drink, this drink should be removed before the next one can be dispensed.
4. A can of beer costs two coins. A cup of coffee costs one coin.
5. There are two kinds of customers: students and professors. Students drink only beer, professors drink only coffee.
6. From time to time the drink storage can be recharged.

Another Example: Vending Machine

1. The vending machine contains a drink storage, a coin slot, and a drink dispenser. The drink storage stores drinks of two kinds: beer and coffee. We are only interested in whether a particular kind of drink is currently being stored or not, but not interested in the amount of it.
2. The coin slot can accommodate up to three coins.
3. The drink dispenser can store at most one drink. If it contains a drink, this drink should be removed before the next one can be dispensed.
4. A can of beer costs two coins. A cup of coffee costs one coin.
5. There are two kinds of customers: students and professors. Students drink only beer, professors drink only coffee.
6. From time to time the drink storage can be recharged.

We would like to prove some properties of this model, for example that a student will never leave money in the coin slot.

How to Establish Correctness

- Consider the system (or a computer program) as a mathematical object. To do this, we will have to build a formal model of the system (or the program).

How to Establish Correctness

- Consider the system (or a computer program) as a mathematical object. To do this, we will have to build a formal model of the system (or the program).
- Use a formal language for expressing intended properties.

How to Establish Correctness

- Consider the system (or a computer program) as a mathematical object. To do this, we will have to build a formal model of the system (or the program).
- Use a formal language for expressing intended properties.
- The language must have a semantics that explains what are possible interpretations of the sentences of the formal language. The semantics is normally based on the notions is true, is false, satisfies.

How to Establish Correctness

- Consider the system (or a computer program) as a mathematical object. To do this, we will have to build a formal model of the system (or the program).
- Use a formal language for expressing intended properties.
- The language must have a semantics that explains what are possible interpretations of the sentences of the formal language. The semantics is normally based on the notions is true, is false, satisfies.
- Write a specification, that is, intended properties of the system in this language.

How to Establish Correctness

- Consider the system (or a computer program) as a mathematical object. To do this, we will have to build a formal model of the system (or the program).
- Use a formal language for expressing intended properties.
- The language must have a semantics that explains what are possible interpretations of the sentences of the formal language. The semantics is normally based on the notions is true, is false, satisfies.
- Write a specification, that is, intended properties of the system in this language.
- Prove formally that the model satisfies the specification.

What is Logic?

Mathematical logic is a branch of science that deals with notions such as

- syntax and semantics;
- proof theory and model theory;
- reasoning.

How to Prove Properties of Programs or Systems?

How to Prove Properties of Programs or Systems?

- Hire all people with PhD in mathematical logic in the world;

How to Prove Properties of Programs or Systems?

- Hire all people with PhD in mathematical logic in the world;
- Delegate the problem of proving to a computer program.

Computational Logic

Computational logic deals with applications of logic in computer science and computer engineerings, including

- software and hardware verification;
- circuit design;
- constraint satisfaction;
- knowledge representation and reasoning;
- semantic Web;
- planning;
- databases (semantics and query optimisation);
- theorem proving in mathematics;
- ...

This Course

- propositional logic;
- satisfiability checking in propositional logic;
- semantic tableaux;
- binary decision diagrams (BDDs);
- quantified boolean formulas;
- propositional logic of finite domains;
- state-changing systems and transition systems;
- temporal logic;
- model checking.

End of Lecture 1

Slides for lecture 1 end here . . .