CS_313 High Integrity Systems/ CS_M13 Critical Systems

Course Notes

Chapter 1: Programming Languages for Writing Safety-Critical Software

Anton Setzer
Dept. of Computer Science, Swansea University
http://www.cs.swan.ac.uk/~csetzer/lectures/critsys/14/index.html

October 10, 2014

Remark

This section is based heavily on Neil Storey [St96], *Safety-critical computer systems*, Addison-Wesley, 1996, pp. 218 - 227.

Main Criteria for Choice of Programming Languages for Critical Systems

- **Logical soundness.**
  - Is there a sound, unambiguous definition of the language?
- **Complexity of definition.**
  - Are there simple, formal definitions of the language features?
  - Too high complexity results in high complexity and therefore in errors in compilers and support tools.
- **Expressive power.**
  - Can program features be expressed easily and efficiently?
  - The easier the program one has written, the easier it is to verify it.

Main Criteria for Choice of Programming Languages for Critical Systems

Security.

- Can violations of the language definitions be detected before execution?
  - Some interpreted languages detect errors only when running it.
  - Various languages like Eiffel and even Java allow to define programs, which
    - the compiler regards as type correct,
    - although they aren't,
    - run time errors are caused by this.

Problem in Java

The problem in Java is:

- Assume a class Person with subtype Student.
- Assume a method which takes as element an array of elements of Person

  void init(person[] myarray){⋯}.
- Assume this method replaces one element of this array by a new element of Person.

  myarray[0] = new Person();
- Since Student is a subtype of Person, an array of Student is a subtype of an array of Person.
- So this method can be called with an array of Student.

  Student[] studentarray = ...
  init(studentarray)

Problem in Java

Student is subtype of Person

void init(person[] myarray){ myarray[0] = new Person(); }.

Student[] studentarray = ...

init(studentarray)

- This is accepted by javac.
- When this is executed, we get a run time error, because at run time the call of init will make the assignment

  studentarray[0] = new Person();

  But studentarray is an array of Student, and new Person() is not a Student.

Example Code

```
public class arrayProblem{
  Student[] studentArray = new Student[10];
  void init(Person[] myarray){ myarray[0] = new Person(); };

  arrayProblem(){ init(studentArray); };

  public static void main(String[] args){
      Student[] studentArray = new arrayProblem().studentArray;};
};

class Person{}
class Student extends Person {}
```


Main Criteria for Choice of Programming Languages for Critical Systems

- **Verifiability.**
  - Is there support for verifying that program code meets the specification?
- **Bounded space and time requirements.**
  - Can it be shown that time and memory constraints are not exceeded?

Common Reasons for Program Errors

- **Subprogram side effects.**
  - Variables in the calling environment are unexpectedly changed.

Example Problem with Side Effects

Consider a function (here using Java syntax):

```
int f(int x){ y = x; return x + 1;}
```

where y is an instance variable.
Consider the following code:

```
z = f(x)
```

f is used as a function, and one might overlook the fact that using f changes y.
Then change of y in f is called a **side effect**.

Side Effects

In general a **side effect** is when evaluating an expression (such as f(x) above) has the result of **changes in the environment**, e.g.

- carrying out some external procedure such as printing out some text, like in

  ```
  int f(int x){ System.out.println(x); return x + 1;}
  ```

- changes of some other variables.

Order of Evaluation

- Side effects cause problems when an expressions makes calls to functions.
- Example:

  ```
  int y = 0;

  int f(x){ y = y + 1; return x;};

  System.out.println(f(0) + y);
  ```

- Consider expression f(0) + y:
- If f(0) is evaluated before y, then y is incremented first by 1, so the result printed is 0 + 1 = 1
- If y is evaluated first, it has still value 0, the result printed is 0 + 0 = 0.

Order of Evaluation in Java

- From the Java language specification, 15.7 http://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.7

  "The Java programming language guarantees that the operands of operators appear to be evaluated in a specific evaluation order, namely, from left to right."

  "The left-hand operand of a binary operator appears to be fully evaluated before any part of the right-hand operand is evaluated."

Common Reasons for Program Errors

- **Failure to initialise.**
  - Variable is used before it is initialised.
- **Aliasing.**
  - Two or more distinct names refer to the same storage location. Changing one variable changes a seemingly different one.

Example Aliasing Problem

- We write ff and tt for the Boolean values false and true.
- Let xor be the binary operation on Booleans with the following truth table:

x	y	x xor y
ff	ff	ff
ff	tt	tt
tt	ff	tt
tt	tt	ff

- One can see easily the following (try out all choices for the variables and check that both sides of the equation give the same result):
  - xor is commutative, i.e. $\mathrm{x} \operatorname{xor} \mathrm{y} = \mathrm{y} \operatorname{xor} \mathrm{x}$.
  - xor is associative, i.e. $\mathrm{x} \operatorname{xor} (\mathrm{y} \operatorname{xor} \mathrm{z}) = (\mathrm{x} \operatorname{xor} \mathrm{y}) \operatorname{xor} \mathrm{z}$.
  - $\mathrm{x} \operatorname{xor} \mathrm{x} = \mathrm{ff}$.
  - $\mathrm{x} \operatorname{xor} \mathrm{ff} = \mathrm{x}$.

Example Aliasing Problem

- The following is a way of exchanging two Boolean values without the use of a temporary variable:

$$\begin{aligned} \mathrm{x} &:= \mathrm{x} \operatorname{xor} \mathrm{y}; \\ \mathrm{y} &:= \mathrm{x} \operatorname{xor} \mathrm{y}; \\ \mathrm{x} &:= \mathrm{x} \operatorname{xor} \mathrm{y}; \end{aligned}$$

Exchange Procedure

- The exchange program exchanges the arguments because if we give different names to the instances of variables

$$\begin{aligned} x1 &= x \text{ xor } y; \\ y1 &= x1 \text{ xor } y; \\ x2 &= x1 \text{ xor } y1; \end{aligned}$$

we get (using the laws above)

$$\begin{aligned} y1 &= x1 \text{ xor } y &&= (x \text{ xor } y) \text{ xor } y &&= x \text{ xor } (y \text{ xor } y) \\ &= x \text{ xor ff} &&= x \\ x2 &= x1 \text{ xor } y1 &&= (x \text{ xor } y) \text{ xor } x &&= y \text{ xor } (x \text{ xor } x) \\ &= y \text{ xor ff} &&= y \end{aligned}$$

Exchange Procedure

- Doing the above bitwise we can exchange as well integers.

Example Aliasing Problem in Java

- In order to write a procedure for exchanging Booleans in Java we need to use a small wrapper class:

```
class MyBool {
   public boolean theBool;
   MyBool (boolean x) { theBool = x;};
}
```

- Now write the exchange function as follows (ˆ = xor)

```
void exchange(MyBool x,MyBool y){
  x.theBool = x.theBool ^ y.theBool;
  y.theBool = x.theBool ^ y.theBool;
  x.theBool = x.theBool ^ y.theBool;
};
```


Example Aliasing Problem

```
void exchange(MyBool x,MyBool y){
  x.theBool = x.theBool ^ y.theBool;
  y.theBool = x.theBool ^ y.theBool;
  x.theBool = x.theBool ^ y.theBool;
};
```

- If `x` and `y` are the same object the above sets x.theBool to false: The last line then reads `x.theBool = x.theBool ^ x.theBool;` which sets (using $x \text{ xor } x = \text{ff}$) `x.theBool = false`
- So if `x.theBool` was true, and `x` and `y` happen to be the same object, the above method is not an exchange function.

Repaired Exchange Procedure

```
void exchange(MyBool x,MyBool y){
  if (x != y) {
     x.theBool = x.theBool ^ y.theBool;
     y.theBool = x.theBool ^ y.theBool;
     x.theBool = x.theBool ^ y.theBool;
   }
};
```


Exchange Program in SPARK Ada

SPARK Ada (introduced in the next Section) will not allow to instantiate exchange function (both the "wrong" and "correct" version) by the same parameter.

Reasons for Program Errors

- **Expression evaluation errors.**
  - E.g. out-of-range array subscript, division by zero, arithmetic overflow.
  - Different behaviour of compilers of the same language in case of arithmetic errors.

Comparison of Languages

Cullyer, Goodenough, Wichman have compared suitability of programming languages for high integrity software by using the following criteria:

Wild jumps.

- Can it be guaranteed that a program cannot jump to an arbitrary memory location?
  - By use of gotos.

Overwrites.

- Can a language overwrite an arbitrary memory location?
  - C, C++ can do so.

Semantics.

- Is semantics defined sufficiently so that the correctness of the code can be analysed?

Comparison of Languages

Model of mathematics.

- Is there a rigorous definition of integer and floating point arithmetic (overflow, errors)?
  - E.g. in Java, floating point arithmetic is defined as following the IEEE floating point arithmetic.
    - States precisely when we get an overflow etc. and what to do if we have an overflow.
  - If this is not precisely defined, a program might
    - run perfectly on the machine used for testing it (which ignores an error)
    - and might crash on the machine, it is actually running.

Comparison of Languages

Operational arithmetic.

- Are there procedures for checking that the operational program obeys the model of arithmetic when running on the target processor?
  - E.g. programs which determine, whether the processor follows the IEEE floating point standard.

Data typing.

- Are there means of data typing that prevent misuse of variables?

Exception handling.

- Is there an exception handling mechanism in order to facilitate recovery if malfunction occurs?

Comparison of Languages

Exhaustion of memory.

- Are there facilities to guard against running out of memory?
  - **Object-oriented** and **functional** programming languages have a problem here, since memory is allocated on the fly.
  - Potential problem of **garbage collection**, if it is executed in a time-critical situation (e.g. the autopilot might carry out garbage collection, while landing).
  - **Recursion** is as well problematic, since the depth of recursion cannot be controlled, and each recursion step requires usually the allocation of new memory.

Safe subsets.

- Is there a safe subset of the language that satisfies requirements more adequately than the full language?

Comparison of Languages

Separate compilation.

- Is it possible to compile modules separately, with type checking against module boundaries?
  - It should be possible to split the program into units (packages, classes), which are located in different files, with separate interface definitions.
  - This allows to verify the correctness of each unit individually, and avoids the danger that exchanging one unit destroys the correctness of already verified units.

Well-understood.

- Will designers and programmers understand the language sufficiently to write safety critical software?

Comparison of Languages

- The next slide contains a comparison of programming languages.
  - The languages are a bit old.
  - Unfortunately I couldn't find any newer comparison of programming languages, only individual comparison of pairs of programming languages.
  - The principles are state of the art – use of safe subsets instead of new programming languages.
- Legend for next slide:
  - + means protection available,
  - ? means partial protection,
  - - means no protection.

Comparison of Languages

	Structured assembler	C	CORAL 66	ISO PASCAL	Modu-la 2	Ada
Wild jumps	+	?	?	?	?	+
Overwrites	?	-	-	?	?	?
Semantics	?	-	?	?	+	?
Model of mathematics	?	-	?	+	+	?
Operational arithmetic	?	-	-	?	?	?
Data typing	?	-	?	?	?	+
Exception handling	-	?	-	-	?	+
Safe subsets	?	-	+	+	?	+
Exhaustion of mem.	+	?	?	?	?	-
Separate compil.	-	-	?	?	+	+
Well understood	+	?	?	+	+	?

Remarks on CORAL 66

- CORAL 66 = compiled structured programming language related to Algol.
- Developed at the Royal Radar Establishment RRE, Malvern, UK.
- Used for real-time systems.
- Allowed inline assembly code.
- No free CORAL 66 compilers seem to be available today.

Analysis

- C most unsuitable language.
- Modula-2 most suitable.
  - Problem of Modula-2: **limited industrial use**.
  - Therefore lack of tools, compilers.
  - Industrial use contributes to reliability of compilers.
- Case study revealed:
  Compiler faults are equivalent to one undetected fault in 50 000 lines of code.
  - Especially problem of optimisation.
  - By using compilers heavily compilers are tested and compiler errors are detected and removed.

Analysis (Cont.)

- One solution: **development of new languages** for high integrity software.
  - Same problem as for Modula-2: limited industrial use.
- Better solution: introduction of **safe subsets**.
  - Rely on standard compilers and support tools.
  - Only additional checker, which verifies that the program is in the subset.
  - Add annotations to the language.

Safe Subsets

	CORAL subset	SPADE-Pascal	Modula2 subset	Ada subset
Wild jumps	+	+	+	+
Overwrites	+	+	+	+
Semantics	+	+	+	?
Model of mathematics	?	+	+	+
Operational arithmetic	?	+	?	+
Data typing	?	+	+	+
Exception handling	-	-	?	+
Safe subsets	?	+	+	?
Exhaustion of mem.	+	+	?	?
Separate compil.	?	?	+	+
Well understood	+	+	+	+

Programming Languages Used

- **Aerospace.**
  - Trend towards Ada.
  - Use of languages like FORTRAN, Jovial, C, C++.
  - 140 languages used in the development of the Boeing 757/767.
    75 languages used in development of the Boeing 747-400.
    E.g. C++ for the seat entertainment system of Boeing 777.
  - Northrup B2 bomber control system: C++

Programming Languages Used

- **Aerospace (Related).**
  - Air traffic control systems in US, Canada, France: Ada.
  - Denver Airport baggage system written in C++, but initial problems probably not directly related to the use of C++.
    - Problems with the software for the Denver Airport baggage system delayed the opening of this airport by one year.
    - The economic damage caused by these problems -shows that this software has some aspects of a **business critical system**.
    - But that's a degree of critically which applies to almost all business software.

Programming Languages Used

- **Spacecraft.**
  - European Space Agency: use of Ada in mission-critical systems.
  - NASA: Assembler, Ada.
  - Space shuttle: Hal/s and Ada plus other languages.
- **Automotive systems:**
  - Much assembler. Also C, C++, Modula-2
- **Railway industry:**
  - Ada as de-facto standard.
- **In general:**
  - Trend towards **Ada** for the high-integrity parts of the software.
  - Use of assembler, where necessary.