

CSCM10 Research Methodology Specification and Verification

Anton Setzer

[http://www.cs.swan.ac.uk/~csetzer/lectures/
computerScienceProjectResearchMethods/current/index.html](http://www.cs.swan.ac.uk/~csetzer/lectures/computerScienceProjectResearchMethods/current/index.html)

Monday 13 November 2017

- ① Critical Systems
- ② Specification
- ③ Verification
- ④ Dependent Type Theory
- ⑤ Security
- ⑥ Theoretical Topics

CSCM10

Specification and Verification

2 / 58

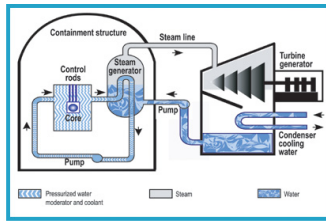
Definition

- ① Critical Systems
- ② Specification
- ③ Verification
- ④ Dependent Type Theory
- ⑤ Security
- ⑥ Theoretical Topics

Definition: A critical system is a

- computer, electronic or electromechanical system
- the failure of which may have serious consequences, such as
 - substantial financial losses,
 - substantial environmental damage,
 - injuries or death of human beings.

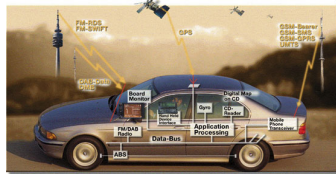
Example 1: Nuclear Power



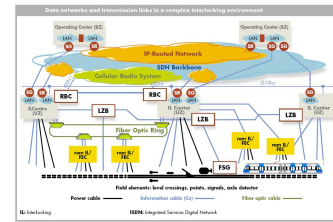
Example: Medical Devices



Example: Embedded Systems in Automobile Industry



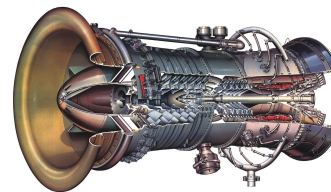
Example: Railways





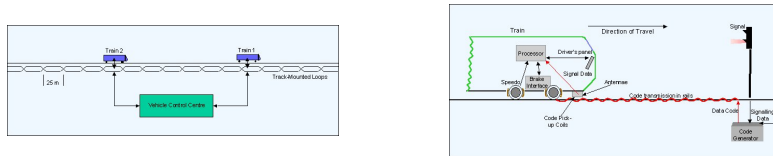
Industrial Partners of Swansea Group in Safe and Secure Systems

Swansea Safe and Secure Systems Group



- The department of Computer Science has a big group working on logic, theoretical computer science and applications to verification of software and hardware.
- Long experience in working with verification of software and hardware.
- Industrial connections with companies such as Rolls Royce, Developers of Electronic Payment Systems, Siemens.

- Well established collaboration with Siemens Rail Automation (Chippenham, formerly Invensys Railsystems) on modelling and verification of new generations of railway interlocking systems.
 - Currently working on radio controlled moving block systems (ERTMS).



- Verification in the Railway Domain
 - Ulrich Berger
 - Phil James
 - Faron Moller
 - Liam O'Reilly
 - Markus Roggenbach
 - Monika Seisenberger
 - Anton Setzer
- Embedded Systems and Testing
 - Arnold Beckmann,
 - Markus Roggenbach.

Why Formal Specification?

- Critical Systems
- Specification
- Verification
- Dependent Type Theory
- Security
- Theoretical Topics

- Natural language specification can be ambiguous.
 - “The output is a red light or a green light”.
 - Do you mean “either or” or “inclusive or”?

- Formal specification enforce precision.
 - Example: If the level of the water in the tank is above a certain level, the plug valve must be closed.
Do you mean
 - maximum level,
 - average,
 - medium,
 - or ... (lots of other possibilities)?

Challenges in Specification

- Finding a suitable language which is
 - expressive
 - and simple enough for the user to understand it.
- Describe the meaning of specification languages (semantics).
- For specifying a formal system, determine the right
 - notions,
 - level of abstraction

- Natural language specifications don't allow formal verification.

Example

- Distant signals and main signal in railways.
Is
 - the main signal a function of the distant signal,
 - or the distant signal a function of the main signal,
 - or are main signal and distant signal in a relation.
- During specification, often need to switch between different choices.
- General problem of modelling systems.

- Algebraic Specification.
 - Markus Roggenbach (CASL)
 - John Tucker (theory of algebraic specification)
- Process Algebras
 - Faron Moller (CCS),
 - Markus Roggenbach (CSP-CASL),
 - Anton Setzer (CSP-Agda).

- ① Critical Systems
- ② Specification
- ③ Verification
- ④ Dependent Type Theory
- ⑤ Security
- ⑥ Theoretical Topics

Verification

- Verification is the process of determining whether a software product coincides with its specification.
- Many methods.
- Main method is testing.
- Testing usually not complete.
- In order to guarantee that a program is guaranteed to be correct, one needs prove that the output of software coincides with the specification.
 - Necessary especially for critical systems.
 - Increasingly used for general systems, e.g. by Microsoft, to guarantee security of its software.
- Done using theorem proving techniques.

4 Ways of Proving Theorems

1. Theorem proving by hand.

- What mathematicians do all the time.
- Will remain in the near future the main way for proving theorems.
- Problem: Errors.
 - As in programs after a certain amount of lines there is a bug, after a certain amount of lines a proof has a bug.
 - The problem can only be reduced by careful proof checking, but not eliminated completely.
- Unsuitable for verifying large software and hardware systems.
 - Data usually too large.
 - Likely that one makes the same mistakes as in the software.

2. Theorem proving with some machine support.

- Machine checks the syntax of the statements, creates a good layout, translates it into different languages.
- Theorem proving still to be done by hand.
- **Example:** most systems for specification of software.
- **Advantages:**
 - Less errors.
 - User is forced to obey a certain syntax.
 - Specifications can be exchanged more easily.
- **Disadvantage:** Similar to 1.

3. Interactive Theorem Proving.

- Proofs are fully checked by the system.
- Proof steps have to be carried out by the user.
- **Advantages:**
 - Correctness guaranteed (provided the theorem prover is correct).
 - Everything which can be proved by hand, should be possible to be proved in such systems.

- (Interactive theorem proving)
 - **Disadvantages:**
 - It takes much longer than proving by hand.
 - Similar to programming:
 - To say in words what a program should do, doesn't take long.
 - To write the actual program, can take a long time, since much more details are involved than expected.
 - Requires experts in theorem proving.

4. Automated Theorem Proving.

- The theorem is shown by the machine.
- It is the task of the user to
 - state the theorem,
 - bring it into a form so that it can be solved,
 - usually adapt certain parameters so that the theorem proving solves the problem within reasonable amount of time.

- (Automated theorem proving)
 - **Advantages**
 - Less complicated to “feed the theorem into the machine” rather than actually proving it.
Might be done by non-specialists.
 - Sometimes faster than interactive theorem proving.

- (Automated theorem proving)
 - **Disadvantages**
 - Many problems cannot be proved automatically.
 - Can often deal only with finite problems.
 - We can show the correctness of one particular processor.
 - But we cannot show a theorem, stating the correctness of a parametric unit (like a generic n -bit adder for arbitrary n).
 - In some cases this can be overcome.
 - Limits on what can be done (some hardware problems can be verified as 32 bit versions, but not as 64 bit versions).

Verification in Industry

- Most verification done using testing.
- Some theorem proving by hand and with some machine support done.
- Increasingly theorem proving using automated theorem proving done.
 - Investment of Microsoft in various automated theorem provers.
 - Package management in Linux became much faster due to use of SAT solvers (Automated Theorem Provers).

Verification in Industry

- Interactive theorem proving on its way into industry.
 - Typical scenario:
 - General properties of a system proved using interactive theorem proving
 - E.g. signalling principles formally expressed safety.
 - That a concrete installation is in accordance with those general principles done using automated theorem proving.
 - E.g. show that a railway interlocking system fulfils signalling principles.

- Verification using automated theorem provers (ATP).
 - Oliver Kullmann (SAT solvers, e.g. OK-Solver)
- Verification using interactive theorem provers (ITP).
 - Markus Roggenbach (Isabelle),
 - Ulrich Berger (Minlog, Coq),
 - Monika Seisenberger (Minlog),
 - Anton Setzer (Agda).

- ① Critical Systems
- ② Specification
- ③ Verification
- ④ **Dependent Type Theory**
- ⑤ Security
- ⑥ Theoretical Topics

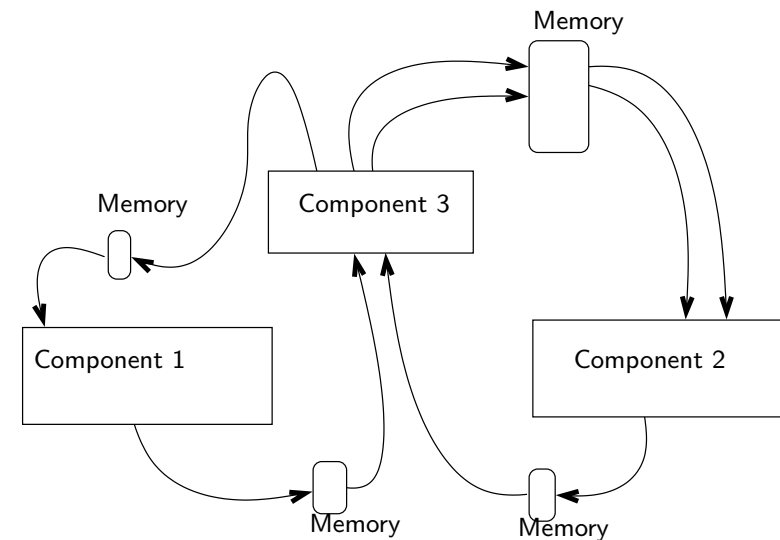
Agda

- Agda is a theorem prover which is as well a prototype of a dependently typed programming language.
- In Agda proofs and programs are the same.
- A proof of a theorem A is a program p of type A written as

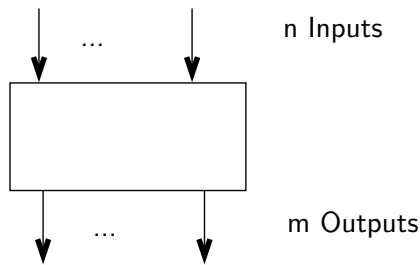
$$p : A$$

- Relatively easy for programmers, since they don't need to learn a different activity.
- Agda uses the novel concept of dependent types.
- In Swansea Anton Setzer is expert in Agda.

Example: Boolean Circuits



What is a Component?



A Boolean Component can be represented by a

$$f : \text{Bool}^n \rightarrow \text{Bool}^m$$

What is the type $\text{Bool}^n \rightarrow \text{Bool}^m$?

- $\text{Bool}^n \rightarrow \text{Bool}^m$ is a type depending on $n, m : \mathbb{N}$.
- In most languages you don't have any dependent type. You need to replace this by $\text{List}(\text{Bool}) \rightarrow \text{List}(\text{Bool})$.
- In C++ you can define

$$\text{Bool}^n \rightarrow \text{Bool}^m$$

but only, if n, m are known at compile time.

- Disallows dynamic dependencies, e.g. depending on user input.
- In Agda we can directly use $\text{Bool}^n \rightarrow \text{Bool}^m$ as a dependent type.

Example 2: Grammars

- Assume you want to write programs which manipulate Java programs.
 - E.g. change a variable not using brute query replace.
- One way of doing this:
 - Define a data type of Java programs.
 - Translate strings into this data type and back again.
 - Write programs which work on this data type of Java programs.

Example 2: Grammars

- An oversimplified grammar for Java might start as follows:

```
JavaProg      → "class" identifier "{ " JavaProgBody " } "  
JavaProgBody → ( VariableDecl ) * ( MethodDecl ) *  
VariableDecl → TypeDecl VariableName ";"  
...
```

- Let Grammarsymbol be the set of terminals and non-terminals (JavaProg, JavaProgbody, ...).
- For each Grammarsymbol S we define the type $\llbracket S \rrbracket$ of entities of this type, e.g.
 - $\llbracket \text{TypeDecl} \rrbracket = \text{String}$.
 - $\llbracket \text{VariableName} \rrbracket = \text{String}$.
 - $\llbracket \text{VariableDecl} \rrbracket = \text{String} \times \text{String}$.
- $\llbracket S \rrbracket$ is a **dependent type** depending on $S : \text{GrammarSymbol}$.

Parser : $(\text{GrammarSymbol} \times \text{String}) \rightarrow \text{Bool}$

Transformer : $(S : \text{GrammarSymbol})$
 $\rightarrow (s : \text{String})$
 $\rightarrow \text{Parser}(S, s) == \text{true}$
 $\rightarrow \llbracket S \rrbracket$

- Makes heavy use of the dependent type $\llbracket S \rrbracket$.
- Parser Libraries in C++, Haskell, Agda have been built based on this idea.

Generative Programming

- These are examples of generative programming.
- In generative programming you want to build highly generic programs, which generate and manipulate programs from elements of data types.

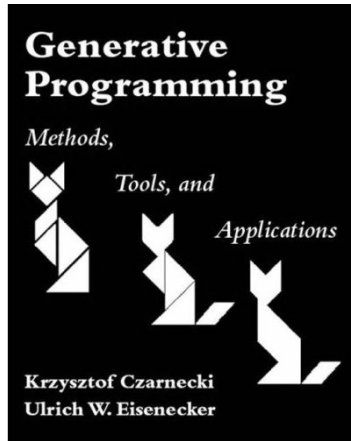
Generative Programming

- So we have
 - a base data type BaseType (like GrammarSymbol before),
 - a type of programs $\text{Program}(S)$ based on $S : \text{BaseType}$ (like $\llbracket S \rrbracket$ before),
 - operations which manipulate $\text{Program}(S)$, e.g.

transform1 : $((S : \text{BaseType1}) \times \text{Program1}(S))$
 $\rightarrow \text{BaseType2}$

transform2 : $((S : \text{BaseType1}) \times \text{Program1}(S))$
 $\rightarrow \text{Program2}(\text{transform1}(S, s))$

- Now we can create factories for generating programs.
- Replace handcrafted programs by generated programs.
- Similar to step from pre-industrial to industrial age.



SortedList

- What is SortedList?
 - An element of SortedList is a list which is sorted.
 - It is a pair $\langle l, p \rangle$ s.t.
 - l is a NatList.
 - p is a proof or verification that l is sorted:
 - $p : \text{Sorted}(l)$.

- Assume we want to assign a type to a sorting function `sort` on lists of natural numbers.
- In most programming language, the type of it is essentially

$\text{sort} : \text{NatList} \rightarrow \text{NatList}$

for the type of lists of natural numbers `NatList`.

- In dependent type theory, we can demand more correctness, namely that its type is

$\text{sort} : \text{NatList} \rightarrow \text{SortedList}$.

- We assume some notion of `NatList` (list of natural numbers).

SortedList

- For the moment, ignore what is meant by $\text{Sorted}(l)$ as a type.
- Only important: $\text{Sorted}(l)$ depends on l .
 - $\text{Sorted}(l)$ is a predicate expressed as a type.
- Elements of SortedList are pairs $\langle l, p \rangle$ s.t.
 - $l : \text{NatList}$.
 - $p : \text{Sorted}(l)$.
- $\text{Sorted}(l)$ is a dependent type.

- An element of $\text{Sorted}(l)$ will be a **proof** that l is sorted.
- If l is **sorted**, then $\text{Sorted}(l)$ will be provable, and therefore **will have an element**.
 - It is possible to write a program which computes an element of $\text{Sorted}(l)$.
- If l is **not sorted**, then $\text{Sorted}(l)$ will have no proof and it will therefore **no element**.
 - Then it is not possible to write a program which computes an element of $\text{Sorted}(l)$.

- Then the pair $\langle l, p \rangle$ will be an element of

$$\text{SortedList} := (l : \text{NatList}) \times \text{Sorted}(l) .$$

- SortedList is the type of pairs $\langle l, p \rangle$ s.t.
 - $l : \text{NatList}$,
 - $p : \text{Sorted}(l)$.
 called the **dependent product**
- $\text{sort} : \text{NatList} \rightarrow ((l : \text{NatList}) \times \text{Sorted}(l))$ expresses:
 - sort converts lists into sorted lists.

The Dependent Function Type

- From a sorting function we know more:
 - It takes a list and converts it into a sorted list **with the same elements**.
- Assume a type (or predicate) $\text{EqElements}(l, l')$ standing for
 - l and l' have the same elements.

The Dependent Function Type

- A refined version of sort has type

$$(l : \text{NatList}) \rightarrow ((l' : \text{NatList}) \times \text{Sorted}(l') \times \text{EqElements}(l, l'))$$

- “ $\text{sort}(l)$ is a list, which is sorted and has the same elements”.
- “ sort is a program, which takes a list and returns a sorted list with the same elements.”
- The type of sort is an instance of the **dependent function type**:
 - The result type depends on the arguments.

- 1 Critical Systems
- 2 Specification
- 3 Verification
- 4 Dependent Type Theory
- 5 Security**
- 6 Theoretical Topics

- Cyberterrorism, General Security
 - Monika Seisenberger, Anton Setzer.
- Cryptocurrencies (Bitcoins, Blockchain).
 - Anton Setzer

Theoretical Topics

- 1 Critical Systems
- 2 Specification
- 3 Verification
- 4 Dependent Type Theory
- 5 Security
- 6 Theoretical Topics**

- Computability Theory and Limits of Computation
 - Ulrich Berger, Jens Blanck, Monika Seisenberger, John Tucker,
- Exact Real Number Computation
 - Ulrich Berger, Jens Blanck, Monika Seisenberger.
- Program Extraction
 - Ulrich Berger, Monika Seisenberger
- Proof Theory
 - Arnold Beckmann, Ulrich Berger, Monika Seisenberger, Anton Setzer, Jean Razafindrakoto.

- Complexity Theory
 - Arnold Beckmann, Oliver Kullmann, Faron Moller, Jean Razafindrakoto.
- Formal Argumentation
 - X. Fan.

- Critical Systems require more formal specification and verification.
- Expertise in Swansea in specification and verification.
- Problems of natural language specification can be overcome by formal specification.
- Verification techniques – from proving by hand to interactive and automated theorem proving.
- Agda as an example of a programming language based on dependent types.
- Use of dependent types for generative programming.
- Research related to Security.
- Wide range of theoretical topics covered in Swansea.