Verification Condition Generators (VCG)

- A verification condition generator computes verification conditions for a given program extended with some annotations (directly or indirectly).
- Proving the verification condition(s) yields higher reliability of the program.
- Using tools like ESC/Java helps to find common bugs in software otherwise unnoticed.
Drawbacks of VCGs

- Program correctness has been transferred to proof correctness
- An automated theorem prover can be used, but is itself a complicated piece of software
- Whether or not a first-order theorem holds is only semi-decidable
- A VCG does not help you to obtain a correct program. It merely proves your program is correct.
Stepwise refinement

- When a statement is inserted between a pre-postcondition pair, new pre- and postconditions can be computed directly for the remainder of the programming problem.
- A tool can administer all proof obligations and solve simple proofs automatically.
- One can build a library of programs that are proved to be correct.
Tool requirements

Correctness should be ensured:

- the tool must be based on a well-founded theory
- even if the tool becomes large a bug in the tool should never lead to undetected bugs in the result. (Using the De Bruijn criterion)

In order to be used by programmers:

- the notations and concepts used should be as close to the pen and paper counterparts as possible.
- the tool should be easy to use. (e.g. it needs an advanced graphical user interface).
Intermezzo: The De Bruijn criterion

- There should be a representation of the proof, that can be checked by a small reliable program.

- Hence, if I do not trust my tool, I can write a validator for the results myself.

- This ensures that even if the tool becomes large and complex, errors will be detected in the end.

- Size and complexity of the tool no longer influence reliability.
Tool requirements

- The tool may not enforce a specific order of proving and programming.
- It must be able to serve as a framework to be used for experiments towards larger scale programs.
Creating a theoretical framework

1) To construct proofs, we use a typed lambda calculus for first-order logic.

First-order logic is very intuitive and widely known amongst all potential users. By using a typed lambda calculus, we achieve a high degree of reliability of the implementation of the tool (due to the De Bruijn criterion)
We can define datastructures in FOL, like natural numbers:

\[
\begin{align*}
\text{nat:}&:*s & \quad \text{// *s is the type of all sets} \\
0: & \text{nat} \\
\text{s}(x: & \text{nat}): & \text{nat} \\
\text{AllNat}: & \text{(forall } x: \text{nat. } x=0 \text{ or (exists } y: \text{nat. } x=s(y)))
\end{align*}
\]

- induction cannot be defined in this way, but has to be added to the logic axiomatically (\text{AllNat} can then be proved by induction)
- the definitions requires equality
Defining equality

Defining equalities:

\[\text{equ}(x, y : \text{nat}) : \star p\]

(\forall x : \text{nat}. \text{equ}(x, x))

(\forall x, y : \text{nat}. \text{equ}(x, y) \Rightarrow \text{equ}(y, x))

(\forall x, y, z : \text{nat}. \text{equ}(x, y) \text{ and } \text{equ}(y, z) \Rightarrow \text{equ}(x, z))

With this, we can prove by induction that if \( P(x) \) and \( \text{equ}(x, y) \) hold, so does \( P(y) \) (Leibniz), but this is a meta-theoretical proof.

(i.e. a proof about the derivation system). Hence, we cannot use Leibniz as a rule!
Creating a theoretical framework

2) In order to support program derivation, a Hoare logic is used. Such a logic is intuitive to the students and directly connects a program to its specification.

Another alternative would be a dynamic logic, but these are a lot less commonly known by potential users. Moreover, they connect programs with an operational semantics.
Hoare Logic in Cocktail

\[ \begin{align*}  
\{P\}S\{Q\} & \quad \{Q\}T\{R\} \\
\{P\}S;T\{R\} 
\end{align*} \]

\[ \begin{align*}  
\Gamma \vdash p: P' \Rightarrow P & \quad \{P\}S\{Q\} \\
\Gamma \vdash q: Q \Rightarrow Q' & \quad \{P'\}S\{Q'\} 
\end{align*} \]
Regard S as a proof (program) that proves that \{P\}\{Q\} is satisfiable. Denote this as S:P\Rightarrow Q.

If P\Rightarrow Q holds, then \{P\}\{Q\} is trivial, since nothing needs to be done! (that is, we need a proof \(p\) of P\Rightarrow Q).

For this step from proof to program, we introduce a special “programming” construct.
The fake statement

\[ \Gamma \vdash p : P \Rightarrow Q \]

fake \( p : P \Rightarrow Q \)

Now use:

\[ \Gamma \vdash p : P' \Rightarrow P \]

fake \( p : P' \Rightarrow P \)

\[ \Gamma \vdash q : Q \Rightarrow Q' \]

fake \( q : Q \Rightarrow Q' \)

\[ S : P \Rightarrow Q \]

fake \( p ; S ; q : P' \Rightarrow Q' \)

The entire logic is now syntax-directed!
Creating a theoretical framework

3) To automatically construct proofs, we implemented a tableau-based automated theorem prover as a proof of concept.

This theorem prover constructs a tableau to prove a theorem, which is then translated into a lambda-term to ensure reliability of the system.
Mixing it all up: Cocktail

- The logic, automated theorem prover and Hoare logic were moulded in order to fit together.
- A software architecture was designed to enable the simultaneous editing of several programs, proofs and theories through a set of coupled structure editors.
- Editors for our framework were implemented, employing a context sensitive graphical user interface.
Cocktail’s theorem prover

Forward and backward reasoning:

- Reasoning through combining known information, yielding new information
- Reasoning through decomposition of the current (sub)goal into smaller goals until the goal is trivial.

Rewriting:

- Either by using a single rule containing an equation
- Or by exhaustively using a set of rules in a specified order and direction (computing normal-forms)
Results

- The formal basis of the tool is a single coherent formalism, which ensures safety of the system in both theory and practice (25 Axiomatic Rules).
- The tool supports interactive derivation of programs from specifications. The program and its correctness proof can be developed simultaneously.
- Cocktail’s theorem prover is fairly intuitive for all users, using usual notations and tactics.
Why this doesn't really work:

- Programmers do not start by writing the specification.
- Since the proofs need to fit perfectly in the program's holes, changing the program is tedious.
- Most proofs need to be constructed manually. Programmers will not do this.
- One wants to use libraries that are not formally verified.
- The language is way too simplified: no arrays, records, pointers/references, etc.
Future work:

- Almost finished: build an extended automated theorem prover that supports equational reasoning.
- Ongoing: support a much more elaborate programming language.
- Use a VCG for post-verification of programs.
- Integrate other approaches and tools within the same framework.
- Support external automated theorem provers
- Provide more theory "off the shelve"
- Support termination (optionally)
Vragen?

Questions?

Fragen?