A formal security policy for Xenon

John McDermott and Leo Freitas
Naval Research Laboratory and University of York
October 2008 @ FMSE
Hoare's Verification Grand Challenge

A mature scientific discipline should set its own agenda and pursue ideals of purity, generality, and accuracy far beyond current needs

- **science** explains why things work in full generality by means of calculation and experiment
- **an engineering discipline** exploits scientific principles in the study of the specification, design, construction, and production of working artifacts, and improvements to both process and design
- **the verification challenge** is to achieve a significant body of verified programs that have precise external specifications, complete internal specifications, machine-checked proofs of correctness with respect to a sound theory of programming
Deliverables

1. **a comprehensive theory of programming**
   - covering the features needed to build practical and reliable programs

2. **a coherent toolset**
   - automating the theory and scaling up to large codes

3. **a collection of verified programs**
   - replacing existing unverified ones
   - continuing to evolve as verified code
   - a repository

You can’t say any more it can’t be done!
Here, we’ve done it!
Circus tools architecture overview

- Circus AST, XML interchange format (CZT project)
- parsers, pretty printing and IDE integration: jEdit, Eclipse
- typechecking (under development) (MSc thesis)
- complete operational semantics specification (PhD thesis)
- compiler (v2): new operational semantics (FACJ article)
- plugable theorem proving architecture (PhD thesis)
  - Z/Eves—Java integration
  - various Z toolkit laws and MC theories
  - studies for PVS and SAT solvers integration
  - trivial symbolic evaluator: arithmetic and propositional calculus
**Circus**

- **Circus** = $Z + \text{CSP} + \text{ZRC}$
- associated refinement theory and calculus based on UTP
- correctness by construction
- **Circus** programs
  - sequence of paragraphs: $Z + \text{channels} + \text{processes}$
  - channels
    * strongly-typed
    * allows generic actuals
  - processes
    * encapsulated state: $Z$
    * behaviour: $Z + \text{CSP} + \cdots$
Circus by example

Accumulator process

```
channel out : \mathbb{N}
process Accumulator1 \cong
begin
    state
    \text{State} \cong [v : \mathbb{N}]
    AddOp \cong [\Delta \text{State} \mid v' = v + 1]
    \bullet (\mu X \bullet \text{out}!v \rightarrow AddOp ; X)
end
```

Notice the nondeterminism on the initial value of \( v \) in \( \text{out}!v \)
**Circus Model checker top-level design**

- **LaTeX (and other) source(s) input → witnesses**
- **symbolic (on-the-fly) refinement model checking**
- **integrated theorem proving with extensible interfaces**
  - stacked pipelining with external tools
  - application oriented theories as (source) modules
  - elegant OO-design: modular, flexible, configurable, etc.
Circus tools summary

Textual UI

- **SPM**: parsing, pretty-printing, typechecking
- compiler for process behaviour (ProBE-like) exploration
- model checker for refinement (FDR-like) checking
- **TPM**: Expr, Pred transformation (and VC elimination)

Graphical UI

- initial integration with jEdit and Eclipse via CZT
- stand-alone GUI in Java or Web-based (envisioned)
- networked version of TUI through sockets
Representing *Circus*: motivations and rationale

Why \( \LaTeX \) and XML?

- CZT support for standard Z-\( \LaTeX \), as well as other Z tools
- interoperability with existing tools that use \( \LaTeX \)
- seemingly description between reports/papers and actual code

Why not like \( \text{CSP}_M \) ?

- Z standard is \( \LaTeX \) oriented
- extending available \( \text{CSP}_M \) parser for Z is difficult
- extending available CZT Z tools for CSP/Circus is easier
Representing *Circus*

**Circus XML format**

- standard format enabling interoperability between Circus tools
- allows different programming languages to handle Circus
- useful in case the language is extended

**General Circus AST**

- enables transformation between (3) different formats.
- Java (AST) code generation from XML (metadata) schemas
- some Circus tools based on this AST:
  1. typechecker
  2. model checker
  3. refinement calculator
  4. *Circus* → Java
Basic tools around AST for *Circus*
Circus tools front end

Specifications processing module (SPM)

Parser

Printer

Typechecker

processing from/to various sources: \LaTeX{}, UNICODE, XML
Compiler architecture overview
Plugable TP architecture

- **symbolic set utilities**: various operations over $\text{Expr} \& \text{Pred}$
- **symbolic evaluators**: predicate, expression, and sch-text transformers
- **theorem proving stacked solvers**: external tools integration
- **integration with SPM for interactive (AskUser) mode**

**Trivial symbolic evaluator (i.e., automatic TP)**

- **simple arithmetic**: $+, -, \times, \div, \mod, >, \geq, <, \leq$, etc.
- **simple predicates**: $\land, \lor, \Rightarrow, \Leftrightarrow, \neg, x, c, \text{num}$, etc.
- **finite predicate calculus**: $\forall, \exists$, etc.
- **most Z tookit laws for sets and relations**
- **MC (regions theory) laws for normalisation**
Theorem proving architecture overview

Theorem proving module (TPM)

Symbolic set utilities
- Rel Op
- Pred Op
- MC Op
- Prop Op
- Inter Op
- ?? Op

Evaluators
- AST-Q
- AST-A
- AST+ AST-Q
- AST-A
- AST+ AST-Q

TP Manager
- TP Bridge
- TP Registry
- Ask user solver
- TP Solver 1
- TP Solver ...
- TP Solver n

LaTeX-A
LaTeX-Q

AST+
Plugable TP architecture (cont.)

**TP stacked solvers: pre**

- integration with at least one available format
- Java interface to call external tools APIs
- recognition of set theoretical Expr and Pred

**TP stacked solvers: post**

- become part of the VC discharging chain
- undischarged VC’s are passed along the chain
- default evaluation of tagged VC’s for batch execution
- rationale of use among solvers rather than simply stacked
Integrated stacked solvers

**internal tools**

- trivial symbolic evaluator *(done)*
- CZT Z schema unfoldler/normaliser *(partially)*
- CZT theory-parameterised rewriting engine *(partially)*

**external tools**

- CZT ZLive: Z animator *(done)*
- Z theorem provers: Z/Eves *(done)*, ProofPower-Z *(TODO)*
- preliminary feasibility study: PVS *(done)*, SAT solvers *(TODO)*
Refinement engine

- “skilful” driver of the compiler
- VC generation for MC algorithm (i.e., regions)
- searching algorithm code derived through refinement calculus

Interacting with the model checker

- composition of SPM, Compiler, and TPM
- various stacked solvers to discharge VCs
- external tools plumbed through XML or Java interfaces
Compiler demo: GCD Euclidian algorithm examples

Versions from wikipedia GCD

- GCDA — abstract recursive version (AbsGCD)
- GCDB — assignment base loop implementation (ImplGCD)
- GCDC — iterative Euclidian algorithm (EucledianGCD)
- GCDD — efficient version of Euclidian algorithm (EfficientGCD)

Let’s see them in Circus

- while loops as tail recursive actions
- i.e., \((\text{while } b \text{ do } P)\) encoded as:

\[
(\mu X \bullet (\text{if } b \rightarrow (P ; X)\mid \neg b \rightarrow \text{Skip } \text{fi}))
\]
Recursive GCD Euclidian algorithm in *Circus*

channel $in, out : \mathbb{N}$

process $AbsGCD \triangleq$

begin

$GCDA \triangleq in?p : (p > 0) \rightarrow in?q : (p > q) \rightarrow GCDA1(p, q)$

$GCDA1 \triangleq (a, b : \mathbb{N}) \bullet$

( if $(b = 0) \rightarrow out!a \rightarrow Skip$

\[ \| \neg b = 0 \rightarrow GCDA1(b, a \mod b) \]

fi)$

\bullet $GCDA$

end
Imperative GCD Euclidian algorithm in *Circus*

```
process ImplGCD ≜
begin
  GCDB ≜ in?p : (p > 0) → in?q : (p > q) → GCDB1(p, q)

  GCDB1 ≜ (a, b : N) •
    (μ X •
      (if (b = 0) → Skip
       || (¬ b = 0) →
          (var t : N • t := b ; b := a mod b ; a := t) ; X
          fi)
      ) ; out!a → Skip
    • GCDB
  end
```
Iterative GCD Euclidian algorithm in *Circus*

```plaintext
process EuclidGCD ⊑
begin
    GCDC ⊑ in?p : (p > 0) → in?q : (p > q) → GCDC1(p, q)
    GCDC1 ⊑ (a, b : N) •
        (µ X •
            (if (a = b) → Skip
             | (¬ a = b) →
                (if (a > b) → a := a − b
                 | (a < b) → b := b − a
            fi) ; X
        fi)
    ) ; out!a → Skip
    • GCDC
end
```
Efficient/factored GCD Euclidian algorithm in *Circus*

```plaintext
process EfficientGCD ≜
begin
  GCDD ≜ in?p : (p > 0) → in?q : (p > q) → GCDD1(p, q)
  GCDD1 ≜ (a, b : N) • (var x, y, x0, y0, t, q : N •
    x, y, x0, y0 := 0, 1, 1, 0 ;
    (μ X •
      (if (b = 0) → Skip
       \ (¬ b = 0) →
       t := b ;
       q := a div b ;
       b := a mod b ;
       a := t ; t := x ;
       x := x0 − (q * x) ;
       x0 := t ; t := y ;
       y := y0 − (q * y) ;
       y0 := t ; X
     fi) ) ; out!a → Skip
    • GCDD
  )
end
```