

Context-Aware Code Certification (C^3)

Jutta Eusterbrock

SEAMLESS Solutions



<http://www.seamless-solutions.de>

Research & Consulting Towards Smart Software

Automated Software Engineering Conference, September 2004, Linz

Safety Risk: Omission of Context

- Spectacularly Costly Software Failures
 - Ariane 5 space launcher *exploded* as converting a floating point number to a signed 16 bit integer was executed with an input data value outside the 'range'
 - Mars Climate Orbit was *lost* due to the usage of both metric and English units, e.g., inches
- More common problems
 - Break-down of client browsers when downloading Server Software
 - EBay-Fake exploiting Java-script features

Javascript Code Example

```
function demo(syzy,actual,data)
{if (syzy < 100) {year = syzy+1900;}
  else {year = syzy;}
  i = 1; value = data[year-i-1900];
  while (!(value>=actual || i==10))
  {i=i+1;
    value=data[year-i-1900];}
  if ((value >=actual)
    {record=false;}
  else {record=true;}
  return record;}
```

 Netscape

?



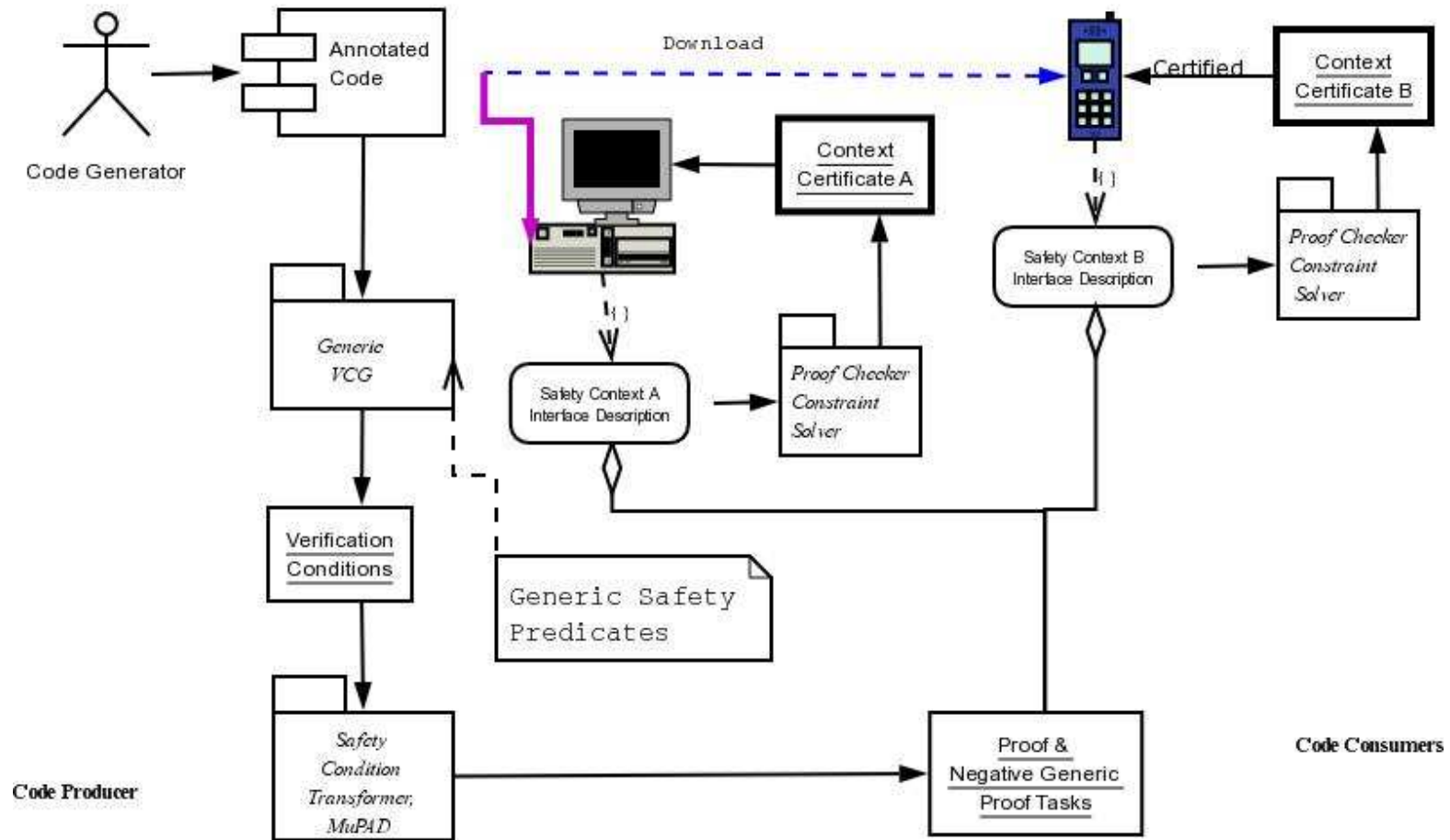
Code Properties

- Behaviour depends on browser-specific computation of current year (*sysy*)

Browser Year	1800	1900	1970	1999	2000	2038	2050	3000
Konqueror	70	70	70	99	100	138	70	70
Netscape 7.0	-100	0	70	99	100	138	150	1100
Explorer	1800	0	70	99	2000	2038	2050	3000

- Program is functionally correct with respect to pre-postconditions and assuming the Explorer year representation

C³ Architecture



MuPAD

Code Certification (Necula)

Code Consumer define safety policy \leftrightarrow Code Producer generate proofs (certificates)

C^3 Program Specifications

Hoare triple: $\{pre-condition\}, program, \{post-condition\}$

- *program*: synthesised generic pseudo code: *assignment, sequence, if then else, while, for*
- *pre-,postcondition*: *Boolean, arithmetic expressions*

Safety Context Γ_{safe}

Specifies the interactions between the foreign software and its execution context and states requirements for safe behavior in terms of *generic (open) safety predicates*:

- $safe(Vars)$ characterizes safe states, eg., through (sub-)types + constraints
- $safe_expr(Expr)$: input operator arguments are valid and the result term is safe
- $safe_assign(Lhs, Rhs)$: the state after assignment $Lhs := Rhs$ is safe, provided that the state before is safe

Context-Aware Code Certification C^3

1. **Extended Floyd-Hoare VCG** for computation of re-usable generic safety pre-condition
2. **Safety pre-condition transformer**
 - (a) Algebraic simplification (MuPAD)
 - (b) Translation into negated clausal form
3. **Context-aware meta-level safety checker**
 - (a) Identifies *program inputs* and *program statements* which cause unsafe behaviour using constraint-solving
 - (b) Loosely connected to safety context, program specifications through viewpoints

Verification Condition Generator (VCG)

Computes a global re-usable safety pre-condition $SafeExpr$ in terms of generic safety predicates, given a Hoare triple $Pre, Prog, Post$, such that $Prog$ is **functionally correct** and **safe to execute in Γ** , iff

1. $\Gamma_{safe} \vdash (\forall x : SafeExpr(x))$
2. the safety conditions are satisfied prior to program execution
3. the program terminates

Verification Condition Generator

= Extended Floyd-Hoare Verification Rules:

Additional **generic preconditions** in terms of the generic safety predicates constrain admissible intermediate program states.

$$Post \vdash x := expr \rightarrow Post[x/expr] \wedge$$

$$safe_expr(expr) \wedge safe_assign(x, expr)$$

Verification Condition Generator

Part of the generated safety expression for sample code

```
safe_expr(sysy < 100) and
  (not sysy < 100 =>
    (safe_assign(year,sysy) and
      safe_assign(value,data(sysy-1901))
      and safe_assign(i,1)) and
    (sysy < 100 =>
      (safe_assign(year,sysy+1900) and
        safe_assign(value,data(sysy-1))
        and safe_assign(i, 1))))
```

Safety Condition Transformer

1. Simplifies the generated expression and reduces the length drastically
2. Translates the expression into an easy to handle negated clausal form:

$$\begin{aligned} &(\forall x(C \Rightarrow (Safe_E1 \wedge Safe_E2))) \Leftrightarrow \\ &\neg(\exists x(C \wedge \neg Safe_E1) \vee \exists x(C \wedge \neg Safe_E2)) \end{aligned}$$

Context-Aware Safety Checker

- Generic proof tasks have the general form

$$\Gamma_{safe} \vdash \neg(\exists x(C \wedge \neg Safe_{E1}) \vee \dots \vee \exists x(C \wedge \neg Safe_{Er})),$$

where Γ_{safe} is a safety context

- Safety Checker tries to identify situations **(counter example generation)**, such that $\exists x : (C \wedge \neg Safe_{E_i})$ is valid

Context-Aware Meta-Level Safety Checker

- Devising a **generic meta-level safety theory**
- Describing the safety context and safety rules in terms of the generic predicates as **constraint-logic program**
- **Lifting** proof tasks to the meta-level into a constraint-satisfaction problem
- Solving the **constraint-satisfaction problems**
- Returning results back by **reflection**

Safety Context: Program Interface

- Relation $vars$ binding members of a list of object-level variables OVL list to their corresponding meta-level variables MVL , ie., $vars(OVL, MVL)$
- Function $\sigma^{-1} : E \times [OVL, MVL] \rightarrow \sigma^{-1}(E)$ maps object-level expressions into meta-level expressions by substituting object-level with meta-level variables
- Type declarations
- Initialisations prior to program execution

Safety Context: Environment

- Type definitions `type(year, integer)`
- Finite domain subtype parameter ranges
 - Example “Netscape7.0” context

```
type_dom(syzy, Dom) :- maxint(MaxInt), Dom#=11..MaxInt.
```

- Admissible typestates
 - Example “Netscape7.0” context

```
type_state([syzy, year], [S, Year]) :-  
    dom(syzy, S), dom(int, Year), Year==(S +1900).
```

Safety Context: Rules

- **Safety properties**: “Uninitialised Variables”, “Out of Bounds Array Access”, “Overflow”, “Division by zero”, “Semantic Constraints”
- Safety properties are defined by instantiating **generic predicates**

```
not_safe_assign(Var, Expr, Vars) :-
```

```
    not_safe_expr(Expr, Vars).
```

```
not_safe_assign(assign(A, I), Expr, Vars) :-
```

```
    type(A, array(Type, Index)), dom(Type, Dom),
```

```
    I notin 0..Index-1.
```

Safety Context: Object \leftrightarrow Meta-level

- **Lifting** (mapping and naming object level proof tasks to the meta-level)

$lift(\exists(C \wedge \neg safe_expr(E)),$

$vars(OVL, MVL) \wedge \sigma^{-1}(C, [OVL, MVL]) \wedge type_state(MVL) \wedge$

$not_safe_expr(\sigma^{-1}(E), [OVL, MVL]) \wedge labeling(MVL))$

- **Reflection** (bring the reasoning results back to the object-level)

$not_safe_expr(\sigma^{-1}(E), [OVL, MVL]) \vdash \neg safe_expr(E)$

Context-Aware Safety Checking of Code Example

Lifting of generated sample proof task

```
<- vars([sysy,a,year,i,value,r],[S,A,I,Y,V,R]),  
constraints([sysy,a,year,i,value,r],Vars),S>=100,  
labeling(Vars), not_safe_assign(V,access(data,S-1901),Vars).
```

Checking the task against various contexts and
bringing the results back to object-level

- 'Netscape 7.0': `sysy=100` falsifies
`safe_assign(value,data(sysy-1901))`
which stems from Stmt (4)
- 'IE Explorer': no safety risks can be identified



- Open framework for specification of context: supports re-use, composition, extensibility;
- Covers various isolated techniques for detecting runtime errors and extends them;
- Finer-granularity than Hoare and type verification (constraints, intermediate states);
- Safety counter examples are more useful than unexplained *yes, no* proof results;
- Proof-of-concept implementation;
- Significantly less complex than verification.

Acknowledgements

