A Case-Study in Component-based Mechanical Verification of Fault-tolerant Programs

S. Kulkarni
Department of Computer and Information Science
The Ohio State University

J. Rushby N. Shankar
Computer Science Laboratory
SRI International

June 5, 1999

Observations

Decomposition of a program is useful in its verification.

A fault-tolerant program can be decomposed into

- a fault-intolerant program

  provides functionality in the absence of faults only

- a set of fault-tolerance components (detectors and correctors)

  provide fault-tolerance only

Feasibility and need for mechanical verification is increasing

Questions

Is the decomposition of a fault-tolerant program into fault-tolerance components useful in its verification?

Do advantages of component-based verification apply in mechanical proofs?

Is it easy to mechanize a 'paper-pencil' proof?

Outline of the Talk

- Dijkstra's token ring program
- Identifying the fault-intolerant program and the corrector
- Our experience
- Concluding remarks

Dijkstra's Token Ring Program

$const\ M, N$		$//\ M \geq N+1\ \wedge\ M \geq 2$
$var\ x.j$	: 0..$(M-1)$	
Initial predicate	: true	
$j \neq 0\ \wedge\ x.(j-1) \neq x.j$	$\longrightarrow$	$x.j := x.(j-1)$
$x.0 = x.N$	$\longrightarrow$	$x.0 := x.0+1\ mod\ M$

Why this program?

- Program is easy to understand
- Proof is considered to be 'tricky'
- Non-component-based mechanical proofs exist for this program

Dijkstra's Token Ring Program

Properties of interest

- token circulation in the absence of faults
  - focus on the fault-intolerant program
- stabilization in the presence of transient faults
  - focus on the corrector

Program Decomposition

Identifying the fault-intolerant program:

$const\ M, N$			// $M \geq 2$
$var\ x.j$	: $0..(M-1)$		
Initial predicate	: S		
$j \neq 0 \ \wedge\ x.(j-1) \neq x.j$		$\longrightarrow$	$x.j := x.(j-1)$
$x.0 = x.N$		$\longrightarrow$	$x.0 := x.0+1\ mod\ M$

where S is the invariant of the token ring program.

Program Decomposition (continued)

Identifying the corrector:

$const\ M, N$		// $M \geq 2$
$var\ x.j$	: $0..(M-1)$	
Initial predicate	: $true$	

$$j \neq 0 \ \wedge \ x.(j-1) \neq x.j \quad \longrightarrow \quad x.j := x.(j-1)$$

Correction predicate $= S$

Steps in Verification

Verification of the fault-intolerant program

- In states where the invariant is true, if the fault-intolerant program executes then the token is circulated along the ring.

Verification of the corrector

- Starting from an arbitrary state, if the corrector executes then eventually a state is reached where the invariant is true.

Interference freedom between the fault-intolerant program and the corrector

Verification of the Corrector

Let $C(j)$ be the action at process j, and let $0 < k \leq N$

Theorem :

$true$ leads to

$\forall j : k \leq j \leq N : x.j = x.N$

in

$([\!]j : k < j \leq N : C(j))$

Proved by induction on $N - k$

Verification of the Corrector (alternative)

Theorem :

$true$ leads to

$\forall j : 0 \le j \le k : x.j = x.0$

in

$([] j : 0 \le j \le N : C(j))$

Proved by induction on k

Requires fairness assumption

Interference Freedom Proof

1. For process 0 to interfere with the corrector, it must execute infinitely often

2. If the action at process 0 executes infinitely often, $x.0$ will take all possible values in the range $0..(M-1)$

3. If the domain of x is large enough then there exists a missing value

4. After $x.0$ is equal to a missing value, convergence to S is achieved when process 0 executes next.

Interference Freedom Proof (continued)

For the first step, we show

> Starting from an arbitrary state, either
>
> – process 0 executes infinitely often, or
>
> – correction to the invariant is achieved.

Theorem :

$$
\begin{array}{l}
\quad true \text{ leads to } S \text{ in } p \\
\Rightarrow \\
\qquad true \text{ leads to } S \text{ in } p[]q \\
\quad \vee\ q \text{ executes infinitely often}
\end{array}
$$

Instantiate :

$S =$ invariant
$p =$ corrector
$q =$ action at process 0

By way of contrast, all non-component based proofs show:

> Starting from an arbitrary state, process 0 executes infinitely often

The proof typically needs to use complex variant functions

Reusability of Proofs

Stabilization is preserved if $x.0$ is incremented by k, where k is relatively prime to M

- Took only 30 minutes for the new proof (compared to approximately 5 days for the original proof)
  - a good heuristic to guess the instantiations

- Program independent theorems

Conclusion

Use of fault-tolerance components in program verification

- focus on the component responsible for the property at hand
- interference freedom proof is often simpler

Application of component-based verification in mechanical proofs

- Reuse of proofs
- Role of assumptions

Mechanizing a ‘paper-pencil’ proof

- In most cases, the steps the effort required to mechanize is comparable to the effort required for the ‘paper-pencil’ proof

Correctors

Specification of a corrector

(correction predicate, X, witness predicate, Z)

- $Z \Rightarrow X$
- $true$ leads to $(X \wedge Z)$
- X *next* X
- Z *next* Z

Examples

error correction codes, reset procedures, voters, rollback recovery, constraint satisfaction

Modeling the Token Ring in PVS

```
state : NONEMPTY_TYPE
trace : TYPE = sequence[state]
prefix : TYPE = finite_sequence[state]
statepredicate : TYPE = PRED[state]
action : TYPE = PRED[[state, state]]
property : TYPE = PRED[trace]
```

Program compositions

```
    Base            % Single action
    Union           % Union of actions
    Restriction     % Restriction of an action
                    % by a state predicate
```

Properties of interest

```
    closure, leads to, convergence,
    detects, corrects, ...
```

Verification of the corrector (continued)

```
same_as_N(j : proc_no) : Assertion =
  (LAMBDA (s : protocol_state) :
     (FORALL (k : above(j)) : x(s)(k) = x(s)(N)))

corr_upto(j : non_zero_proc) : program =
  ProcActions(
     (# length := N - j + 1,
        seq := (LAMBDA (n : below(N - j + 1)) :
        Base(corr(j + n))) #))

conv_induction : LEMMA
  FORALL (k : non_zero_proc, seq : trace) :
      run(corr_upto(k))(seq)
      =>
      (EXISTS (n : nat) :
         FORALL (m : upfrom(n)) :
            same_as_N(k-1)(seq(m)))
```

```
corr(j : non_zero_proc)(s0, s1 : protocol_state) : bool =
   (x(s0)(j) /= x(s0)(j-1)) AND
   (x(s1)(j) = x(s0)(j-1)) AND
   FORALL (k : proc_no) : k /= j => x(s0)(k) = x(s1)(k)


corr_union : finite_sequence[program] =
   (# length := N,
      seq := (LAMBDA (n : below(N)) : Base(corr(1 + n))) #)


corr_prog : program = ProcActions(corr_union)


action_zero(s0, s1 : protocol_state) : bool =
   x(s0)(0) = x(s0)(N) AND
   x(s1)(0) = mod(x(s0)(0) + 1, M) AND
   FORALL (j : proc_no) :  j /= 0 => x(s0)(j) = x(s1)(j)
```