First-order Mu-Calculus as a Framework for Program Verification

Mads Dam

SICS and KTH/IMIT

With contributions by Lars-åke Fredlund, Dilian Gurov, Christoph Sprenger, Gennady Chugunov

Background

Experiment on

- source level
- theorem proving
- for distributed applications

Source language: Mainly Erlang

Executed at FDT lab, SICS, 1995-2003+

Approach, experiences, and lessons

Theorem Proving – Why?

The are many interesting distributed programs to verify

- dynamic process structures
- client-server applications
- migrating processes

against many interesting properties

- temporal properties
- functional properties
- as yet undetermined mixes

There is no decidable framework that will allow this

So we need to resort to theorem proving

Is Theorem Proving Easier Than Model Checking?

By using intelligence in proof search, can we bypass the combinatorial difficulties in model checking?

Yes:

- We are not forced to brute force state exploration when an intelligent choice of invariant will do

No:

- The combinatorial explosion of parallellism is for real
- Must tackle, e.g., true concurrency style diamond properties

Handling the combinatorial complexity along with interaction is the fundamental difficulty!

The Setting

Need a framework with at least:

- First-order logic to talk about elements, process identifiers, stores, states, etc
- Induction and coinduction to define data structures, transition relations, and interesting program properties

Our proposal:

First-order logic + induction + coinduction

= first-order mu-calculus

Mu-Calculus

Kleene -Tarski fixed point theorem:

Every monotone function f on a complete lattice has a complete lattice of fixed points

$\mu x.f(x)$: least fixed point of f

$\nu x.f(x)$: greatest fixed point of f

$\mu^0 x.f(x) = ;$ $\qquad$ $\nu^0 x.f(x) =$ “all”

$\mu^{\kappa+1} x.f(x) = f(\mu^\kappa x.f(x)$ $\qquad$ $\nu^{\kappa+1} x.f(x) = f(\nu^\kappa x.f(x))$

$\mu^\lambda x.f(x) = \bigcup_{\kappa<\lambda} \mu^\kappa x.f(x)$ $\qquad$ $\nu^\lambda x.f(x) = \bigcap_{\kappa<\lambda} \nu^\kappa x.f(x)$

Then:

$\mu x.f(x) = \bigcup_\kappa \mu^\kappa x.f(x)$ $\qquad$ $\nu x.f(x) = \bigcap_\kappa \nu^\kappa x.f(x)$

Examples

f = λx.8 y. TransRel(x,y) ! f(y)

- μx.f(x) = AF"terminated"
- νx.f(x) = true

f = λx. good(x) Ç 9y. TransRel(x,y) Æ f(y)

- μx.f(x): EFgood
- νx.f(x): EFgood Ç EGEXtrue

How to Embed Your Favourite Logic

- Data types:

  Nat = μX(n). n=0 Ç 9n1.n=n1+1 ...

- Language:

  Prog = μX(p). p=skip Ç 9p1,p2. ...

- States:

  State(s) = (9p,t. Prog(p) Æ Store(t) Æ s = (p,t)) Ç ...

- Embeddings of operational semantics:

  TransRel =

  μX(s1,s2).(9t.Store(t) Æ s1 = (skip,t) Æ s2 = t) Ç ...

- Embedding of logic:

  {φ}p{ψ} = 8s. State(s) Æ φ(s) ! (νX(s). (Terminal(s) Æ ψ(s)) Ç (9 sn. TransRel(s,sn) Æ X(sn)))(s)

Proof System

Key innovation: Mechanism for lazy handling of induction

Main components:

- Gentzen-type proof system for FOMuC
- Explicit ordinal approximations
- Loop discharge mechanism

Sequent Calculus for FOMuC

Sample goal:

$$\Rightarrow \mathrm{AFgood}(p \mid q)$$

(p and q are message-passing processes)

Obs: Modularity for free!

$$\frac{\Rightarrow \mathrm{subspec}(p) \qquad \mathrm{subspec}(x) \Rightarrow \mathrm{AFgood}(x \mid q)}{\mathrm{subspec}(p) \Rightarrow \mathrm{AFgood}(p \mid q)}$$
$$\Rightarrow \mathrm{AFgood}(p \mid q)$$

No free lunch: Need a proof system + know how to use it!

Results

Theorem-proving basics:

- Ordinal approximations, soundness and completeness of discharge (Dam, Gurov, Sprenger)

Language embedding framework:

- General, compositional verification (Simpson-95,Dam-95,Fredlund-01)
- Instantiations – CCS, Erlang, pi-calculus, JavaCard (Papers by Dam, Fredlund, Gurov, Chugunov a.o.)
- Completeness for context-free + pushdown cases (Simpson-Schoepp)

Case studies

- Erlang (Arts-Dam), JavaCard (Huisman-Gurov-Barthe)

Tools

- www.sics.se/fdt/vericode (Fredlund)

Issues

I. Theorem-proving framework
II. Programming language embeddings
III. Logic and proof system embeddings
IV. Case studies
V. Tool support
VI. Related work

I. Theorem-Proving Framework

Motivation: Tableau-based model checking

Let P = a.P + b.P

$$\dfrac{\dfrac{\dots \quad \dots \quad \dfrac{[\ P{:}AG(\langle a\rangle true \wedge \langle b\rangle true)\ ]^*}{P{:}[a]AG(\langle a\rangle true \wedge \langle b\rangle true)} \quad \dots}{P{:}\langle a\rangle true \wedge \langle b\rangle true \wedge [a]AG(\langle a\rangle true \wedge \langle b\rangle true) \wedge \dots}}{P{:}AG(\langle a\rangle true \wedge \langle b\rangle true)^*}$$

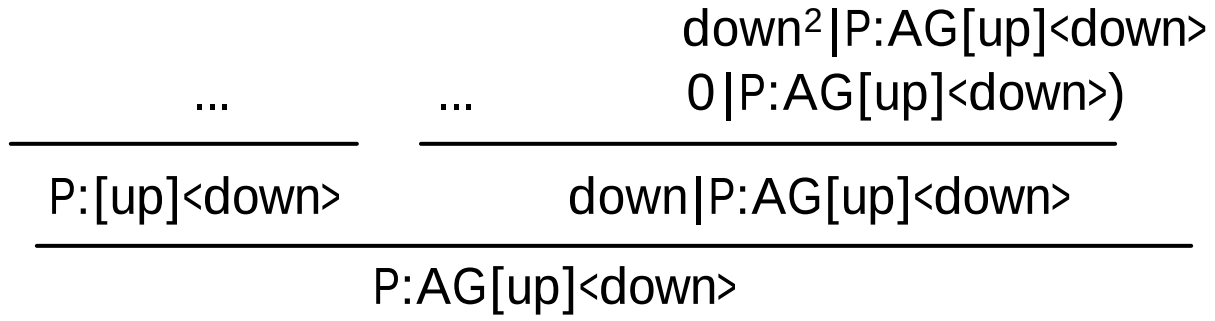
Induction principle: Induction on derivation length

Works for finite state processes

”Counter-example”

Let’s try to do the same for an infinite state process!

Let P = up.(down|P)



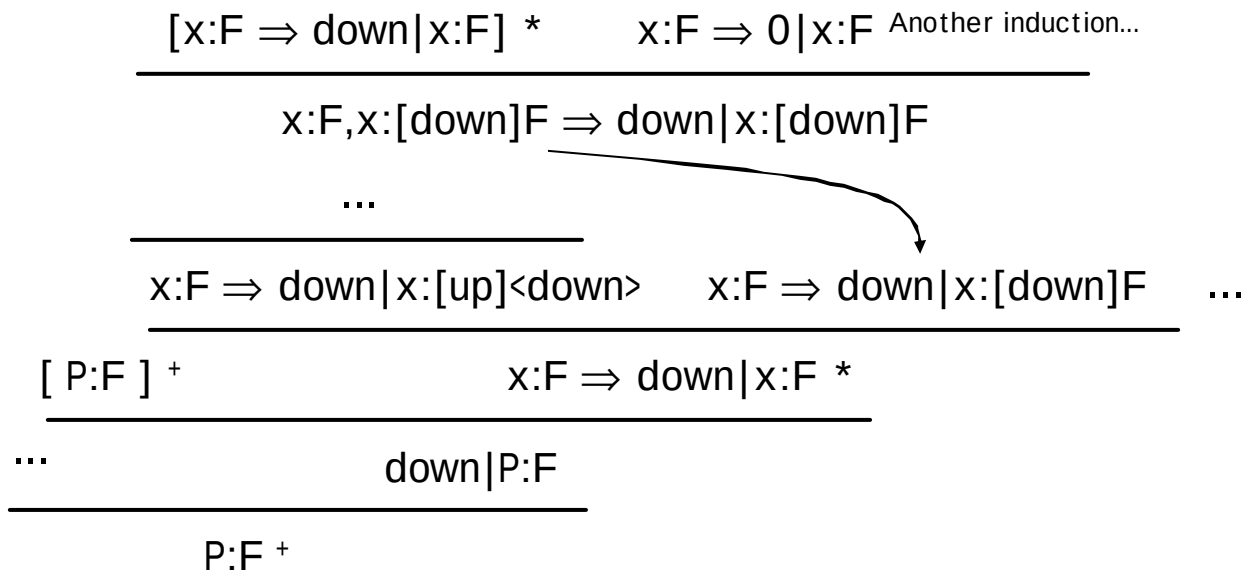
Can we rescue the set-up?

Use a Cut!

Recall P = up.(down|P)

Let F = AG[up]<down>

(= νX.[up]<down> Æ [down]X Æ [up]X)



How to Make This Work?

1. Use mu-calculus
2. How to handle fixed points?
   - Alternating fixed points problematic
   - As for model checking ($\Rightarrow$ P:F)
   - Here also *direct interference* (coming up)
   - Sol'n 1: Terrible mess (Dam'95)
   - Sol'n 2: Explicit ordinal approximants (DG'00)
3. How to embed the operational semantics?
   - Need rules to reflect local behaviour of process connectives
   - Sol'n 1: Sort of ad-hoc (Dam'95)
   - Sol'n 2: Use transition relation embedding (Simpson'95)
   - Sol'n 3: Use 1st-order mu-calculus (Fredlund'01)

How to Do Induction, 1?

Option 1: Fixed point induction a la LCF:

$$\frac{-}{F[\mu x.F/x] \Rightarrow \mu x.F}$$

$$\frac{F[G/x] \Rightarrow G}{\mu x.F \Rightarrow G}$$

Difficult to use in practice

Doesn't fit well with the Gentzen-type framework

How to Do Induction, 2?

Option 2: Unique naming (Stirling),
tagging (Winskel)

$$\frac{\Rightarrow P{:}F[\nu x.\{P\}UA.F/x]}{\Rightarrow P{:}\ \nu x.A.F}$$

$$\frac{-}{\Rightarrow P{:}\ \nu x.\{P\}UA.F}$$

Excellent for model checking

Doesn't fit well with the Gentzen-type framework

Fixed Point Interference

Schematically

Let F = $\mu X1.\nu X2.\langle a\rangle X2 \wedge \langle b\rangle X1$

G = $\mu Y1.\nu Y2.\langle a\rangle Y1 \wedge \langle b\rangle Y2$

$$\dfrac{\dfrac{\dfrac{[\alpha'<\alpha \Rightarrow X2(\alpha'), Y2(\beta'')]^* \qquad [\beta'<\beta \Rightarrow X2(\alpha''), Y2(\beta')]^*}{\alpha'<\alpha \Rightarrow X2(\alpha'), Y1 \qquad \beta'<\beta \Rightarrow X1, Y2(\beta')}}{\alpha'<\alpha, \beta'<\beta \Rightarrow \langle a\rangle X2(\alpha') \wedge \langle b\rangle X1, \langle a\rangle Y1 \wedge \langle b\rangle Y2(\beta')}}{\Rightarrow X2(\alpha), Y2(\beta)}^*$$

$$\Rightarrow X1, Y1$$

Discharge not sound!

(Not easy to handle using constants or tagging)

How to Do Induction, 3?

Option 3: Well-founded induction

Use Kleene-Tarski through:

$$\frac{\Gamma, \forall k' < k.F[k'/k] \Rightarrow F, \Delta}{\Gamma \Rightarrow \forall k.F, \Delta}$$

+ Kleene-Tarski = the canonical proof method for mu-calculus
- Use of explicit ordinal arithmetic
- "Eager" solution to interference problem

How to Do Induction, 4?

Option 4: Lazy induction (here)

Unfolding +
Global check of interference freedom

+ Lazy handling of interference
- Use of explicit ordinal arithmetic
- Global check can be problematic

Mu-Calculus With Explicit Ordinal Approximations*

Syntax: FOL + (approximated) fixed points

$$F ::= \text{FOL formula} \mid F_X(t)$$

$$F_X ::= X \mid \mu X(y).F \mid \mu^k X(y).F$$

Remarks:

- t term
- Individual, predicate, ordinal variables
- Both X and y bound in $\mu X(y).F$ and $\mu^k X(y).F$
- Usual syntactic monotonicity condition applies
- No ordinal arithmetic

Semantics

Model M = (A,e)

- A first-order structure
- e valuation

Let $H = \lambda P.\lambda a.||F||_{e[P/X][a/y]}$

Then

- $|| \mu X(y).F \; ||e = \mu H$
- $|| \mu^k X(y).F \; ||e = \mu^{e(k)} H$

Proposition:

- $\mu H = \sup_\alpha \mu^\alpha H$
- $\mu^\alpha H = \sup_{\beta<\alpha} H(\mu^\beta H)$

Sequents, Validity

Sequents:

$$\Gamma \Rightarrow_O \Delta$$

where O finite partial order on ordinal variables

Validity: $\Gamma \Rightarrow_O \Delta$ *valid,* if

$$\wedge\Gamma \Rightarrow_O \vee\Delta$$

true in all models that respect O:

- whenever $k <_O k'$ then $e(k) < e(k')$

Local Proof Rules

4 basic rules + symmetric version for ν if needed

$$\mu\text{-L} \quad \frac{\Gamma, (\mu^k X(y).F)(t) \Rightarrow_{O'} \Delta}{\Gamma, (\mu X(y).F)(t) \Rightarrow_O \Delta} \quad O' = OU\{k\}$$

$$\mu\text{-R} \quad \frac{\Gamma \Rightarrow_O \Delta, F[(\mu X(y).F)/X, t/y]}{\Gamma \Rightarrow_O \Delta, (\mu X(y).F)(t)}$$

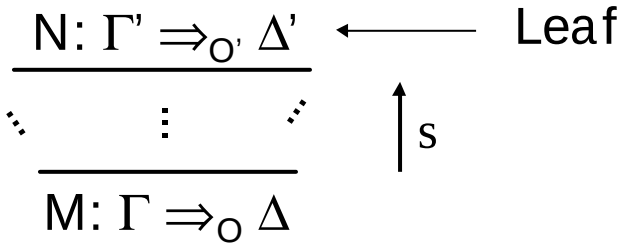
$$\mu^k\text{-L} \quad \frac{\Gamma, F[\mu^{k'} X(y).F/X, t/y] \Rightarrow_{O'} \Delta}{\Gamma, (\mu^k X(y).F)(t) \Rightarrow_O \Delta} \quad O' = OU\{k' < k\}$$

$$\mu^k\text{-R} \quad \frac{\Gamma \Rightarrow_O \Delta, F[(\mu^{k'} X(y).F)/X, t/y]}{\Gamma \Rightarrow_O \Delta, (\mu^k X(y).F)(t)} \quad (k' <_O k)$$

Derivation Trees and Pre-Proofs

Derivation tree D = (N,E,L) sequent-labelled

Repeat:



Condition:

- $substition $s.\Gamma s \supseteq \Gamma'$, $\Delta s \supseteq \Delta'$, $O s \supseteq O'$
- N is called *repeat* node, M is *companion*

Pre-proof graph:

- Each leaf is a repeat, add back edges

Runs –Semantic Discharge

Run of pre-proof: Rooted path of pre-proof, labelled by valuations:

$$\Pi = (N_0,e_0) \ldots (N_i,e_i) \ldots$$

Labels: e_i respects O_i

Tree edges: $(N_i,N_{i+1}) \in E$ implies that e_{i+1} agrees with e_i on variables common to N_i and N_{i+1}

Repeat: (N_{i+1},N_i,s) repeat implies $e_{i+1} = e_i \bullet s$

Semantic Discharge, II

Proof: Pre-proof for which all runs are finite

- Proof = pre-proof + well-foundedness
- Reference discharge condition to which others are compared

Theorem:

If there is a proof of $\Gamma \Rightarrow_O \Delta$ then $\Gamma \Rightarrow_O \Delta$ is valid

Syntactic Discharge

Trace: Rooted path of pre-proof, labelled by ordinal constraints:

$$\Pi = (N_0,(k_0,k_0')) \ldots (N_i,(k_i,k_i')) \ldots$$

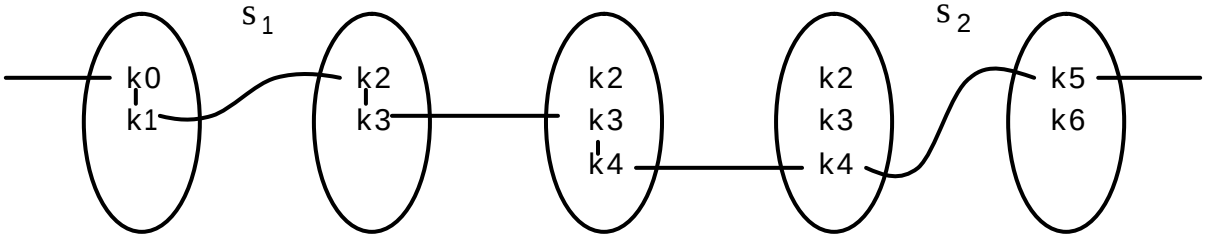
Labels: $k_i' \leq_{O_i} k_i$

Tree edges: $(N_i,N_{i+1}) \in E$ implies $k_i' = k_{i+1}$

Repeat: (N_{i+1},N_i,s) repeat implies $k_i' = s(k_{i+1})$

Syntactic Discharge, 2

Example:



Corresponding trace fragment:

$(N_0,(k_0,k_1))$ $(N_1,(k_2,k_3))$ $(N_2,(k_3,k_4))$ $(N_3,(k_4,k_4))$ $(N_4,(k_5,k_5))$

repeat companion repeat companion

Syntactic Discharge, 3

Progress:

Trace: Progress at i: $k_i' <_{O_i} k_i$

Progressive – progresses i.o.

Path: Exists progressive trace along suffix

Syntactical discharge condition:

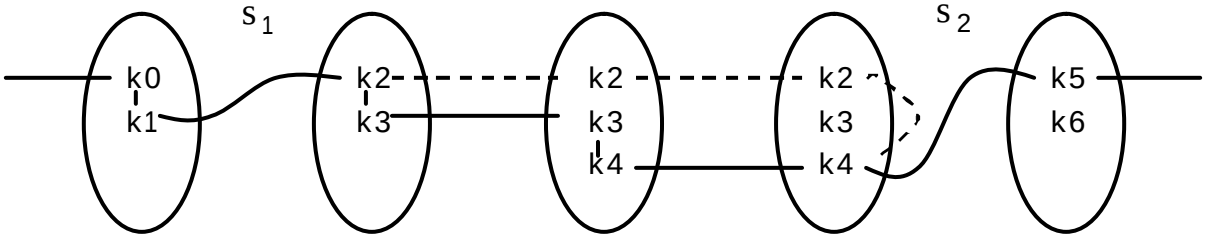
All infinite paths of pre-proof graph are progressive

Theorem:

Syntactic and semantic discharge are equivalent

Normal Traces

Observation: Any trace can be converted into *normal trace*



Only progress at repeats:

$(N_0,(k_0,k_1))$ $(N_1,(k_2,k_2))$ $(N_2,(k_2,k_2))$ $(N_3,(k_2,k_4))$ $(N_4,(k_5,k_5))$

repeat companion repeat companion

Automata-Theoretic Discharge

Construct two Buchi automata B_1 and B_2 over repeats:

- B_1 recognises traversed sequences of repeats
- B_2 recognises repeats potentially connected through a normal trace

Automata-theoretic discharge condition:

$$L(B_1) \subseteq L(B_2)$$

Automata-Theoretic Discharge, 2

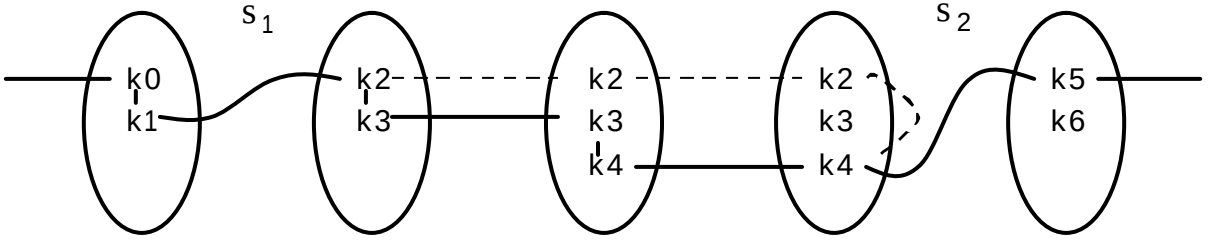
Automaton B2:

States $\{(k_1,R,k_2) \mid R = (M,N,s),\ s(k_2) \leq_{O_N} k_1\}$

Accepting $\{\{(k_1,R,k_2) \mid s(k_2) <_{O_N} k_1\}$

Transitions (k_1,R,k_2) -> (k_2,R',k_3)

Example:



Discharge, Results

Theorem:

The semantic, syntactic, and automata-theoretic discharge conditions are equivalent

The automata-theoretic DC can be checked in time $2^{O(n^3 \log n)}$ where n is number of nodes

Subsumes earlier Rabin-like conditions by Schöpp-Simpson and DFG+DG

- Obtained by restricting B_2 to (k,R,k)
- Complexity drops to $2^{O(n^2 \log n)}$
- Are these conditions complete?

Related Work

Sprenger-Dam, ESOP'03:

Equivalence of well-founded (local) and lazy (global) induction

By explicit proof conversion

II. Programming Language Embeddings

Example: CCS

$P ::= 0 \mid a.P \mid P + P \mid P|P$

Stick to merge || for simplicity

TransRel = $\mu X(p,a,q).(p=a.q)$ \/
(∃p1,p2.p=p1+p2 /\ TransRel(p1,a,q)) \/
(∃p1,p2.p=p1+p2 /\ TransRel(p2,a,q)) \/
(∃p1,p2,q1,q2.p=p1|p2 /\ q=q1|p2 /\
TransRel(p1,q1)) \/ (...symmetric case))

Embedding HML

Define

p:<a>F = (<a>F)(p) = $\exists$q.TransRel(p,a,q) /\ F(q)

p:[a]F = ([a]F)(p) = $\forall$q. TransRel(p,a,q) $\Rightarrow$ F(q)

Can derive:

$$\frac{\Gamma \Rightarrow_O \text{TransRel}(p,a,q), F(q), \Delta}{\Gamma \Rightarrow_O p{:}\langle a\rangle F, \Delta}$$

$$\frac{\Gamma, \text{TransRel}(p,a,x), F(x) \Rightarrow_O \Delta}{\Gamma, p{:}\langle a\rangle F \Rightarrow_O \Delta} \quad x \text{ fresh}$$

Simpson's Embedding

Can derive also:

$$\frac{\Gamma \Rightarrow_O \mathrm{TransRel}(p1,a,q1),\Delta}{\Gamma \Rightarrow_O \mathrm{TransRel}(p1|p2,a,q1|p2),\Delta}$$

$$\frac{\Gamma[p1|y/x],\mathrm{TransRel}(p2,a,y) \Rightarrow_O \Delta[p1|y/x] \quad \Gamma[y|p2/x],\mathrm{TransRel}(p1,a,y) \Rightarrow_O \Delta[y|p2/x]}{\Gamma,\mathrm{TransRel}(p1|p2,a,x) \Rightarrow_O \Delta}$$

Compositional Proof Rules

Can derive also compositional rules in style of Dam, Stirling, Winskel:

$$\frac{\Gamma \Rightarrow_O p1:\langle a\rangle F',\Delta \qquad \Gamma,x:F' \Rightarrow_O x|p2:F,\Delta}{\Gamma \Rightarrow_O p1|p2: \langle a\rangle F,\Delta}$$

$$\frac{\Gamma \Rightarrow_O p1:[a]F1,\Delta \qquad \Gamma,x:F1 \Rightarrow_O x|p2:F,\Delta \qquad \Gamma \Rightarrow_O p2:[a]F2,\Delta \qquad \Gamma,y:F2 \Rightarrow_O p1|y:F,\Delta}{\Gamma \Rightarrow_O p1|p2: [a]F,\Delta}$$

III Logic and Proof System Embeddings

- Temporal logic, finite state model checking
- Context-free and pushdown processes (Schöpp-Simpson)
- Hoare logic, compositional Owicki-Gries
- Pi-calculus
- ...

IV Case Studies

Main exercise so far:

- EVT (now VeriCode, VCPT) – Erlang Verification Tool
- 1996-2001+
- Developed everything: Framework, Erlang semantics, algorithms, proof system, tactics, case studies, documentation,...
- Main focus on dynamic process networks
  - Arts-Dam: Part of distributed database lookup manager
  - Fredlund-Dam: Billing agent
  - Noll-Arts: Generic server

Erlang

- Functionally flavoured programming language for concurrent and distributed applications, developed at Ericsson Computer Science Lab
- Actor-like, first-order, call-by-value
- Asynchronous buffered message passing
- Dynamic process creation
- Error detection and recovery – within a process - between processes
- Other features, modules, distribution, interfacing to non-Erlang code, hot module replacement - not yet considered
- In production use (AXD, Engine)

Example 1 – Simple 2-Process System

```
sys ->
  Pid = self(),
  spawn(analyzer,[Pid,K,L]),
  receive
    {ok,B} -> ... ;
    error -> ... ;
    after 12 -> ...
  end.

analyzer(From,N,M) ->
  case analyse(N,M) of
    ok -> From!{ok,leq(N,M)} ;
    _ -> From!error
  end.
```

Example 2 – RPC

```
server ->
  receive
    {Client,{apply,F,Args}} ->
      spawn(reply,[Client,F,Args]), server

reply(Client,F,Args) ->
  Client!(apply(F,Args))
```

Obs: Dynamic process creation!

```
server @ P1 -> ...
  server @ P1 || P2!(apply(f,args)) @ P3 -> ...
    server @ P1 || P2!(apply(f,args)) @ P3
      || P4!(apply ...) @ P5 -> ...
```

Erlang Operational Semantics

Sequential process state: <E, P,Q>

- E: Erlang term under elaboration
- P: Process identifier (pid)
- Q: P's mailbox/input queue

Process configurations: C ::= {E,P,Q} | C || C

Transition rule flavour:

$$\frac{\text{<E\_1,P,Q> =alpha=> <E\_1',P',Q'>}}{\text{<(E\_1,E\_2),P,Q> =alpha=> <(E\_1',E\_2),P',Q'>}}$$

$$\frac{\text{<E,P,Q> =spawn(f,A,P'')=> <E',P',Q'>}}{\text{\{E,P,Q\} =tau=> \{P'',P',Q'\} || \{f A,P'',empty\}}}$$

Specification Logic

Types (terms, pids, queues, states, configurations)

FOMuC with a number of (now) defined predicates:

- value(pid) = t
- unevaluated(t)
- queue(t1) = t2
- local(t)
- … others …

Specification Logic - Example

```
nat(T) <= T=0 \/ exists X.T=X+1 /\ nat(X) ;
ground(T) = nat(T) \/ tuple(T) \/ ... ;
groundterm(Pid) = exists X.value(Pid)=X /\ ground(X) ;

terminating(Pid) <=
  groundterm(Pid) \/
  ((<>true \/ exists X,Y.<X!Y>true) /\
   []terminating(Pid) /\
   forall X,Y.[X!Y]terminating(Pid) /\
   forall X,Y.[X?Y]sort_of_terminating(Pid)) ;

sort_of_terminating(Pid) =>
  terminating(Pid) /\
  forall X,Y.[X?Y]sort_of_terminating(Pid) ;
```

Specification of server

Property of {server,p1,eps}:

Suppose {p2,{apply,f,v}} is received by p1

Suppose p1=/=p2

If {f v,p,q}:quiet /\ terminating(p) for all p and q

Then eventually(exists v'.<p2!v'>true)

In other words:

```
serverspec(P1) =
  forall P2,F,V.
  [P1?{P2,{apply,F,V}}]
  P1=/=P2 implies
  (forall P,Q.
    {F V,P,Q}: quiet /\
    terminating(P))
  implies
  eventually(exists
    V'.<P2!V'>true)
```

Proof of server

Outline:

```
|- {server,P1,eps}:serverspec(P1)
              Ý
P2=/=P1,forall P,Q.{F V,P,Q}:quiet/\terminating(P)
    |- {P2!(F V),P,empty}||{server,P1,empty} :
          eventually(exists V'.<P2!V'>true
             Ý
P2=/=P1,forall P,Q.{X,P,Q}:quiet/\terminating(P)
    |- {P2!X,P,empty}||{server,P1,empty} :
          eventually(...)
             Ý (by 2 process cuts)
```

Proof cont'd

```
forall P,Q.{X,P,Q}:quiet/\terminating(P)
    |- {P2!X,P,empty} : eventually ... /\
         ``only output to P2''
```

and

```
P2=/=P1 |- {server,P1,empty} :
    ``no output and only input to P1''
```

and

```
P2=/=P1,
  C1:eventually ... /\ ``only output to P2'',
  C2:``no output and only input to P1''
    |- C1||C2 :eventually ...
```

Experiences with EVT

Proof of concept – it actually works

The eternal truth of software verification:

- It's all about finding the right invariant

The eternal truths of (theorem-proving) tool building

- It's a lot of work
- It's not for beginners
- With more resources we could have built a really useful tool ;-)

The eternal half-truth of mu-calculus

- It is tricky
- (But expressing complex properties of infinite trees is far more so)

V The VeriCode Tool

Features:

- Traditional interactive theorem prover EXCEPT
- Prover manipulates graphs, not tree frontiers
- Discharge + subsumption
- Tactics and tacticals as usual
- Tactic applications grow the graph
- Tactics + scripting language: SML
- Theory facility:

  Input your favorite operational semantics, and presto…
- Logical variables
- URL: http://www.sics.se/fdt/projects/vericode/vcpt.html

VI Related Work

Simpson'95: Compositionality via cut-elimination

- For HML and GSOS

Spatial logic (Caires,Cardelli, Gordon):

- Spatial connectives for structural congruence
- Modal ops for reduction/transition
- Fixed points/monadic second order quantification for recursive properties

Earlier work on compositional verification (Stirling, Winskel, Andersen)