A THEORY FOR COMMUNICATING SEQUENTIAL PROCESSES IN COQ

Carlos Alberto da Silva Carvalho de Freitas
(casc2@cin.ufpe.br)

Supervisor: Gustavo Carvalho

Universidade Federal de Pernambuco
Centro de Informática, 50740-560, Brazil

Introduction

Concurrent systems

- Parallel execution of components
- Deadlock, nondeterminism and other issues
- Testing cannot guarantee properties such as determinism

CSP: a theory for Communicating Sequential Processes

- Clear and accurate description of concurrent systems
- Designs can be proven correct with respect to desired properties

Centro de Informática UFPE

Universidade Federal de Pernambuco

Introduction

Refinement (model) checkers

- Analysis and verification of systems via state exploration
- FDR: most popular refinement checker for CSP
- State explosion problem

Verifying properties by proof development

Objectives

Main: *"provide an initial formalisation of the CSP language in Coq."*

Specific objectives

- Define the syntax of a subset of CSP in Coq
- Support for the LTS representation based on the SOS
- Verify traces refinement via property-based random testing

Main contributions

- Abstract and concrete syntax for a subset of CSP operators
- Contextual rules for CSP specifications
- Operational semantics via the SOS approach
- Inductive and functional definitions of LTSs and traces
- LTS visualisation using GraphViz
- Automation for checking contextual rules and is-a-trace relation
- Traces refinement verification using Quickchick

Agenda

Centro de Informática UFPE

Universidade Federal de Pernambuco

Agenda

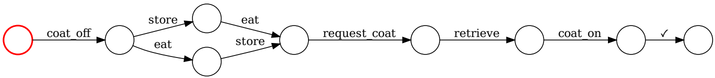
CSP: Communicating Sequential Processes

Example: the cloakroom attendant

- CSP_M

```
channel coat_on, coat_off, store, retrieve, request_coat, eat
SYSTEM =
   coat_off -> store -> request_coat -> retrieve -> coat_on -> SKIP
   [| {coat_off, request_coat, coat_on} |]
   coat_off -> eat -> request_coat -> coat_on -> SKIP
```

- LTS



- Traces
  - $\langle coat\_off, store, eat \rangle$,
    $\langle coat\_off, eat, store, request\_coat \rangle$
    ...

Coq: a proof assistant

Functional and inductive definitions

```
Fixpoint evenb (n:nat) : bool :=
  match n with
    | O ⇒ true
    | S O ⇒ false
    | S (S n') ⇒ evenb n'
  end.
```

```
Inductive ev : nat → Prop :=
  | ev_0 : ev 0
  | ev_SS (n : nat) (H : ev n) : ev (S (S n)).
```

Proof development and the tactics language Ltac

```
Lemma negb_involutive : ∀ (b : bool),
  negb (negb b) = b.
Proof.
  destruct b.
  - simpl. reflexivity.
  - simpl. reflexivity.
Qed.
```

```
Ltac solve_negb_inv b :=
  destruct b; simpl; reflexivity.
```

QuickChick: a property-based testing tool

Example

```
Fixpoint remove (x : nat) (l : list nat) : list nat :=
  match l with
    | [] ⇒ []
    | h::t ⇒ if h =? x then t else h :: remove x t
  end.
Conjecture removeP : ∀ x l, ¬ (In x (remove x l)).
QuickChick removeP.
```

Output

```
0
[0, 0]
Failed!  After 17 tests and 12 shrinks
```

Centro de Informática UFPE

Universidade Federal de Pernambuco

Agenda

Centro de Informática UFPE

Universidade Federal de Pernambuco

Abstract and concrete syntax

Event prefix

Abstract	Concrete
`ProcPrefix (Event "e") STOP`	`"e" --> STOP`

Example: process "PRINTER"

- CSP_{Coq}

Abstract

```
Proc "PRINTER"
 (ProcPrefix (Event "accept")
  (ProcPrefix (Event "print")
   STOP))
```

Concrete

```
"PRINTER" ::= "accept" -->
    "print" --> STOP
```

- CSP_{M}

```
PRINTER = accept ->
      print -> STOP
```

Abstract and concrete syntax

Constructor	**CSP$_M$**	**CSP$_{Coq}$**
Stop	STOP	STOP
Skip	SKIP	SKIP
Event prefix	e -> P	e --> P
External choice	P [] Q	P [] Q
Internal choice	P |~| Q	P |~| Q
Alphabetised parallel	P [A || B] Q	P [[A \\ B]] Q
Generalised parallel	P [| A |] Q	P [| A |] Q
Interleave	P ||| Q	P ||| Q
Sequential composition	P ; Q	P ;; Q
Event hiding	P \ A	P \ A
Process definition	P := Q	P ::= Q
Process name	P	ProcRef "P"

Structured Operational Semantics

Inference rule

Event prefix

$$\frac{}{(a \to P) \xrightarrow{a} P}$$

External choice

$$\frac{P \xrightarrow{a} P'}{P \Box Q \xrightarrow{a} P'} \quad (a \neq \tau)$$

Inductive definition: *sosR*

```
Inductive sosR : specification →
  proc_body → event_tau_tick → proc_body → Prop :=
  | prefix_rule (S : specification) (P : proc_body) (a : event) :
    S # (a --> P) // Event a ==> P
  | ext_choice_left_rule (S : specification) (P Q : proc_body) :
    ∀ (P' : proc_body) (a : event_tau_tick),
        ¬ eq a Tau →
        (S # P // a ==> P') →
        (S # P [] Q // a ==> P')
```

Labelled Transition Systems

Inductive definition: *ltsR'* (part 1/2)

```
Inductive ltsR' :
  specification →
  set transition →
  set proc_body →
  set proc_body →
  Prop :=
  | lts_empty_rule (S : specification) (visited : set proc_body) :
      ltsR' S nil nil visited
```

Labelled Transition Systems

Inductive definition: *ltsR'* (part 2/2)

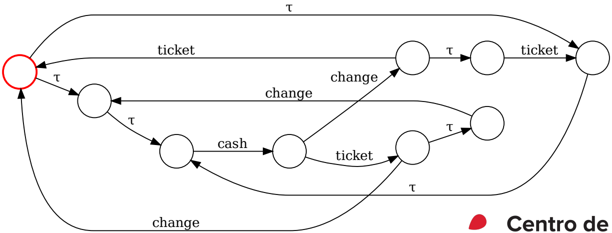
```
| lts_inductive_rule
        (S : specification)
        (T : set transition)
        (P : proc_body)
        (tl visited : set proc_body) :
      let T' := transitions_from P T in
      let T'' := set_diff transition_eq_dec T T' in
      let visited' := set_add proc_body_eq_dec P visited in
      let to_visit := set_diff proc_body_eq_dec
          (set_union proc_body_eq_dec tl (target_proc_bodies T'))
          visited' in
      (∀ (a : event_tau_tick) (P' : proc_body),
         (S # P // a ==> P') ↔ In (P,a,P') T') →
      ltsR' S T'' to_visit visited' →
      ltsR' S T (P :: tl) visited.
```

Labelled Transition Systems

Functional definitions: *compute_ltsR* and *generate_dot*

```
Definition compute_ltsR
  (S : specification) (name : string) (limit : nat) : option (set transition).
Definition generate_dot (lts : option (set transition)) : string.
```

Graph visualisation (GraphViz)



Traces refinement

Inductive definition: *traceR'*

```
Inductive traceR' : specification → proc_body → trace → Prop :=
  | empty_trace_rule (S : specification) (P : proc_body) :
     traceR' S P nil
  | event_trace_rule (S : specification) (P P' : proc_body)
    (h : event_tau_tick) (tl : trace) :
     ¬ eq h Tau →
     (S # P // h ==> P') →
     traceR' S P' tl →
     traceR' S P (h::tl)
  | tau_trace_rule (S : specification) (P P' : proc_body) (t : trace) :
     (S # P // Tau ==> P') →
     traceR' S P' t →
     traceR' S P t.
```

Traces refinement

Proof automation for the is-a-trace relation: *solve_trace*

```
Example MACHINE_TRACE :
  traceR PARKING_PERMIT_MCH "MACHINE" ["cash" ; "ticket" ; "change"].
Proof. solve_trace. Qed.
```

Traces refinement formalisation

```
Definition trace_refinement
 (S : specification) (Spec Imp : string) : Prop :=
  ∀ (t : trace), traceR S Imp t → traceR S Spec t.
Notation "S '#' P '[T=' Q" := (trace_refinement S P Q)
  (at level 150, left associativity).
```

Traces refinement

Traces generator: *gen_valid_trace'*

```
Fixpoint gen_valid_trace'
 (S : specification) (P : proc_body) (size : nat)
 : G (option semantics_trace.trace) :=
 match size with
 | O ⇒ ret nil
 | S size' ⇒
   freq_ (ret nil) [
     (1, ret nil) ;
     (size,
       bind (gen_valid_trans S P) (
         fun t ⇒ (
           match t with
           | nil ⇒ ret nil
           | (Event e, Q) :: _ ⇒
             bind (gen_valid_trace' S Q
               size') (
                 fun ts ⇒ ret (Event e :: ts)
               )
           | (Tick, Q) :: _ ⇒
             bind (gen_valid_trace' S Q
               size') (
                 fun ts ⇒ ret (Tick :: ts)
               )
           | (Tau, Q) :: _ ⇒
             bind (gen_valid_trace' S Q
               size') (
                 fun ts ⇒ ret ts
               )
           end
         )
       )
     )
   ]
 end.
```

Centro de Informática UFPE

Universidade Federal de Pernambuco

Traces refinement

Demonstrating the random generator of valid traces

Sample (*gen_valid_trace PARKING_PERMIT_MCH* "MACHINE" 10).

Output:

```
[Some []; Some ["cash"; "change"; "ticket"; "cash";
"change"]; Some ["cash"]; Some ["cash"; "ticket";
"change"; "cash"]; Some []; Some ["cash"; "change";
"ticket"; "cash"]; Some ["cash"; "change"; "ticket"];
...]
```

Traces refinement

Executable property: *traceP*

```
Definition traceP
  (S : specification)
  (proc_id : string)
  (fuel : nat)
  (t : option semantics_trace.trace) : bool.
```

Refinement checker: *trace_refinement_checker*

```
Definition trace_refinement_checker
  (S : specification)
  (Imp Spec : string)
  (trace_max_size : nat)
  (fuel : nat) : Checker :=
    forAll (gen_valid_trace S Imp trace_max_size)
      (traceP S Spec fuel).
```

Traces refinement

Searching for counterexamples with QuickChick

```
Definition EXAMPLE : specification.
Proof.
  solve_spec_ctx_rules (
    Build_Spec
      [ Channel {{"a", "b", "c"}} ]
      [ "P" ::= "a" --> "b" --> ProcRef "P" ;
        "Q" ::= ("a" --> "b" --> ProcRef "Q") [] ("c" --> STOP) ]
  ).
Defined.
QuickChick (trace_refinement_checker EXAMPLE "Q" "P" 5 1000).
```

Output:

```
Some ["c"]
*** Failed after 3 tests
and 0 shrinks. (0 discards)
```

Agenda

Centro de Informática UFPE

Universidade Federal de Pernambuco

Conclusion

CSP_{Coq}: an initial formalisation of the CSP language in Coq

- Inductive and functional definitions of LTSs and traces
- Third-party visualisation support for LTS representation
- Automation for checking contextual rules and is-a-trace relation
- Random testing for refinement relations using QuickChick

Centro de Informática UFPE

Universidade Federal de Pernambuco

Related work

CSP-Prover

- Interactive theorem prover based on Isabelle
- Stable-failures model as the underlying denotational semantics
- Semi-automated proof tactics for refinement verification

Isabelle/UTP

- Implementation of the *Unifying Theories of Programming*
- Support for construction of denotational semantic meta-models
- Useful to construct program verification tools

Distinguishable features of CSP_{Coq}

- Graphical representation of LTSs
- Property-based testing for checking traces refinement relations

Future work

- Extend the CSP_{Coq} dialect to include other CSP operators
- Check for invalid recursions (hiding and parallelism operations)
- Define a tactic to automate proofs involving the relation *ltsR*
- Prove correctness of definition *compute_ltsR*
- Prove correctness of generator *gen_valid_trace*
- Define traces refinement in terms of bi-simulation

A THEORY FOR COMMUNICATING SEQUENTIAL PROCESSES IN COQ

Carlos Alberto da Silva Carvalho de Freitas
(casc2@cin.ufpe.br)

Supervisor: Gustavo Carvalho

Universidade Federal de Pernambuco
Centro de Informática, 50740-560, Brazil