Generating Proof-Carrying Code for the UDP Protocol

Cordell Green
Garrin Kimmell
Christoph Kreitz
James McDonald
Douglas R Smith
Eric W Smith
Eddy Westbrook
Stephen Westfold

Kestrel Institute
Palo Alto, California

Derivation Structure

Requirements

↓

Specification_1

↓

Specification_2

↓

...

↓

Specification_i

Refinements embody design decisions and preserve properties

Semi-automated tools generate refinements and co-develop programs and proofs

documentation | code | certification evidence, proofs

Derivational Software Engineering

software evolution is primarily requirements evolution

$reqts_0$ → $reqts_1$ → $reqts_2$ → ...

$reqts_0$	$reqts_1$	$reqts_2$
↓	↓	↓
$spec_{01}$	$spec_{11}$	$spec_{21}$
↓	↓	↓
$spec_{02}$	$spec_{12}$	$spec_{22}$
↓	↓	↓
...	...	...
↓	↓	↓
$spec_{0i}$	$spec_{1j}$	$spec_{2k}$
↓	↓	↓
$code_0$	$code_1$	$code_2$

Semi-automated tools regenerate the derivation after requirements change

Challenge

Generally: How to automate the production of CxC code

Specifically: How to generate secure implementations for communication protocols

Goals:

- develop methods to generate high-assurance implementations of standard protocols: UDP/TCP/IP
- develop methods for deriving protocols to mitigate classes of faults and threats

Derivation Structure

Requirements

↓

Specification_1

↓

Specification_2

↓

...

↓

Specification_i

Refinements embody design decisions and preserve properties

Semi-automated tools generate refinements and co-develop programs and proofs

documentation | code | certification evidence, proofs

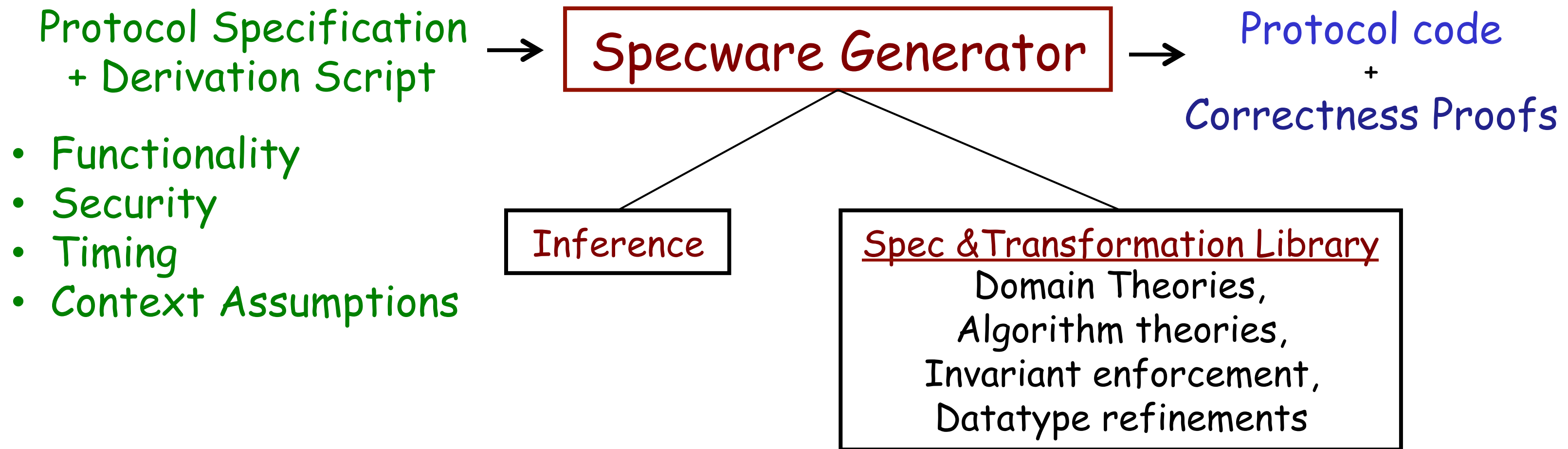
Initial Specification: disjunctive set of cases

Current Refinement Sequence: udp_rcv

MQ. mergeRules + QE for udp_queue_rcv_skb
MR. mergeRules + QE for udp_rcv
R1. introduce disjunctive definitions
OR1. Introduce definitions for observers
Cot1. finalizeCotype Host
Defs1. Introduce definitions for testing
Prep. Prepare for C generation
Cgen. Generate C code

CGEN1. substBaseSpecs;
CGEN2. normalizeTopLevelLambdas;
CGEN3. instantiateHOFns;
CGEN4. removeCurrying;
CGEN5. liftUnsupportedPatterns;
CGEN6. translateMatch true
CGEN7. lambdaLift;
CGEN8. expandRecordMerges;
CGEN9. letWildPatToSeq;
CGEN12. conformOpDecls;
CGEN13. encapsulateProductArgs
CGEN14. introduceRecordMerges;
CGEN17. addEqOps;
CGEN18. addTypeConstructors;
CGEN19. deconflictupdates
CGEN20. makeUpdatesStateful
CGEN21. globalize;
CGEN22. simplifySpec;
CGEN23. adjustElementOrder
CGEN24. removeGeneratedSuffixes;
CGEN25. liftSequences
CGEN26. emitCFiles

Project Goals



- Specify network protocols of TCP/IP stack in Specware's logic
- Derive correct-by-construction implementations using Specware's library of transformations
- Generate machine-checkable proof for correctness of generated code
- Develop strategies that automate the derivation of code and the construction of proofs

Receiving a UDP Datagram/Packet

Application

rcv

12 NetSem rules

Socket

sk_receive_queue

sk_buff
sk_buff
sk_buff
sk_buff

3 NetSem rules

UDP processing

firewall

IP processing

Network Interface Controller

DMA

sk_buff
sk_buff
sk_buff
sk_buff

2 NetSem rules

interrupt-driven transfer

sk_buff
sk_buff
sk_buff
sk_buff

ring buffer

per cpu queue

Kernel space

Specification of UDP Processing

Specification

1. We specify the intended behavior via a set of cases or rules
2. each case is expressed logically and denotes a set of traces
3. specification of a UDP operation is the disjunction of the cases

Synthesis

1. ruleMerge - compose the cases to form a postcondition
2. finalizeCotype – synthesize the postconditions into state update code
3. globalize – treat state as mutable and global, with destructive updates
4. generate C

Specifying Algebraic Types

An algebraic type is defined by constructors

- well-founded
- new functions defined inductively over constructors

type List a = nil | cons a (List a)

List is defined
using constructors
nil and cons

op length: List a → Nat
length nil = 0
length (cons a lst) = 1 + length lst

length is defined
in terms of its value
over the constructors

Specifying Coalgebraic Types (aka cotypes)

A coalgebraic type is characterized by observers

- not well-founded: may be circular or infinite
- transformers specified coinductively by effect on observers

```
type Host
op      inq: Host→ List Msg
op    outq: Host→ List Msg

op  enqueue_oq(H:Host) (m:Msg) :
                  {H':Host|     inq H' = inq H
                             & outq H'  = (outq H) ++ [msg]
                             & …  }
```

Host is specified using observers inq, outq, …

enqueue_oq is specified in terms of its effect on the observers

UDP Receive: Rules/Cases

1. Normal case

```
dequeue_iq (iq h) = (iq h', Some (UDP dgram))
&& dgram.is1 = Some i3
&& dgram.is2 = Some i4
&& dgram.ps1 = ps3
&& dgram.ps2 = ps4
&& dgram.data = data
&& i4 in? local_ips (ifds h)
&& ~(is_broadormulticast (ifds h) i4)
&& ~(is_broadormulticast (ifds h) i3)

&& Some sid = lookup_udp (socks h) (i3,ps3,i4,ps4) (bound h) (arch h)
&& sock    = TMApply (socks h, sid)           % bind sock
&& sock.pr = UDP_PROTO (rcvq)                  % bind rcvq

&& rcvq' = rcvq <| Dgram_msg ( Some i3, ps3, data)
&& sock' = sock << {pr=UDP_PROTO (rcvq')}      % update socket component
&& socks h' = update (socks h) sid sock'          % update host components
&& oq h' = oq h
&& udp_host_frame_ax (h,h')
```

UDP Receive: Rules/Cases

2a. Exception case: no local/destination socket

```
vars:  dgram, i3,i4,ps3,ps4,data, icmp

dequeue_iq (iq h) = (iq h', Some (UDP dgram))
&& dgram.is1 = Some i3
&& dgram.is2 = Some i4
&& dgram.ps1 = ps3
&& dgram.ps2 = ps4
&& i4 in? local_ips (ifds h)
&& ~(is_broadormulticast (ifds h) i4)
&& ~(is_broadormulticast (ifds h) i3)

&& None = lookup_udp (socks h) (i3,ps3,i4,ps4) (bound h) (arch h)
&& icmp = ICMP {is1 = Some i4,        % Create ICMP message
                is2 = Some i3,
                … }
&& (if icmp_to_go               % update host components
      then (oq h', true) = enqueue_oq(oq h, icmp)
    else oq h' = oq h )
&& socks h' = socks h
&& udp_host_frame_ax (h,h’)
```

UDP Receive: Rules/Cases

2b,c,d. Exception cases: bad packets are dropped

```
dequeue_iq (iq h) = (iq h', Some (UDP dgram))
&& dgram.is2 = Some i4
&& dgram.ps2 = ps4
&& i4 in? local_ips (ifds h)
&& is_broadormulticast (ifds h) i4      % broadcast or multicast dest
&& udp_host_frame_ax (h,h')

dequeue_iq (iq h) = (iq h', Some (UDP dgram))
&& dgram.is2 = Some i4
&& dgram.ps2 = ps4
&& i4 in? local_ips (ifds h)
&& dgram.is1 = None                     % no Datagram
&& udp_host_frame_ax (h,h')

dequeue_iq (iq h) = (iq h', Some (UDP dgram))
&& dgram.is2 = Some i4
&& dgram.ps2 = ps4
&& i4 in? local_ips (ifds h)
&& dgram.is1 = Some i3
&& is_broadormulticast (ifds h) i3   % broadcast or multicast source
&& udp_host_frame_ax (h,h')
```

ruleMerge Transformation

Suppose we have the following simple cases:

C1. $A \wedge B \wedge \alpha(st, st')$ -- when A and B hold, do α
C2. $A \wedge \neg B \wedge \beta(st, st')$ -- when A and ¬B hold, do β
C3. $\neg A \wedge \gamma(st, st')$ -- when ¬A holds, do γ

we can straightforwardly merge the disjunction of these cases:

```
C1 ∨ C2 ∨ C3 =  if A
                then if B
                     then α(st, st')
                     else β(st, st')
                else γ(st, st')
```

ruleMerge Transformation

Handling other first-order features

Data Constructors

type Option t = | None | Some t

C1. x=None $\wedge$ α(st, st') -- when x is None, do α
C2. x=Some y $\wedge$ β(st, y, st') -- when x is Some y, do β

then

```
C1 ∨ C2  =   case x of
               | None    -> α(st, st')
               | Some y -> β(st, st')
```

Elaborations

- handling data dependencies between conjuncts
- heuristic choice of atom to branch on
- need to control the explosion of cases by hiding/nesting some disjunctions
- introduction of quantification
- multiple inputs and outputs (tuples) of functions
- pattern matching to destructure values
- deriving preconditions to preclude false postconditions
- generating proofs as part of the construction

Mergerules Invocation

```
op udp_queue_rcv_skb
     (h:host, sock:Sock_ID, skb: sk_buff_ID | true)
     :{(h',ret):host*Int32  |
              udp_queue_rcv_skb_1 (h,sock,skb,h',ret)
           || udp_queue_rcv_skb_2 (h,sock,skb,h',ret)
           || udp_queue_rcv_skb_3 (h,sock,skb,h',ret)
           || udp_queue_rcv_skb_4 (h,sock,skb,h',ret)
           || udp_queue_rcv_skb_5 (h,sock,skb,h',ret)
           || udp_queue_rcv_skb_6 (h,sock,skb,h',ret)
      }
```

Postcondition Generated from Rules

```
refine def udp_queue_rcv_skb (h: host, sock: Sock_ID, skb: sk_buff_ID | ~ false):
    {(h', ret): (host * Int.Int32) |
     if kestrel_xfrm4_policy_check(h, sock, XFRM_POLICY_IN, skb)
      then
        ex(h1: host, csum_ret: Bool)
         (h1, csum_ret) = udp_lib_checksum_complete(h, skb)
          && (if ~ csum_ret
                then
                  if sock_rcv_ok? h1 sock
                   then
                     if ~(sock_owned_by_user(h1, sock))
                      then
                        ex(h2: host, ret1: Int.Int32)
                         (h2, ret1) = sock_queue_rcv_skb(h1, sock, skb)
                          && ret = ret1
                          && (if ret1 = 0 then h' = h2 else h' = kfree_skb(h2, skb))
                     else
                     ret = - 1
                      && (ex(h2: host) h2 = sk_add_backlog(h1, sock, skb)
                                 && h' = kfree_skb(h2, skb))
                  else h' = kfree_skb(h1, skb) && ret = - 1
              else h' = kfree_skb(h1, skb) && ret = - 1)
     else h' = kfree_skb(h, skb) && ret = - 1}
```

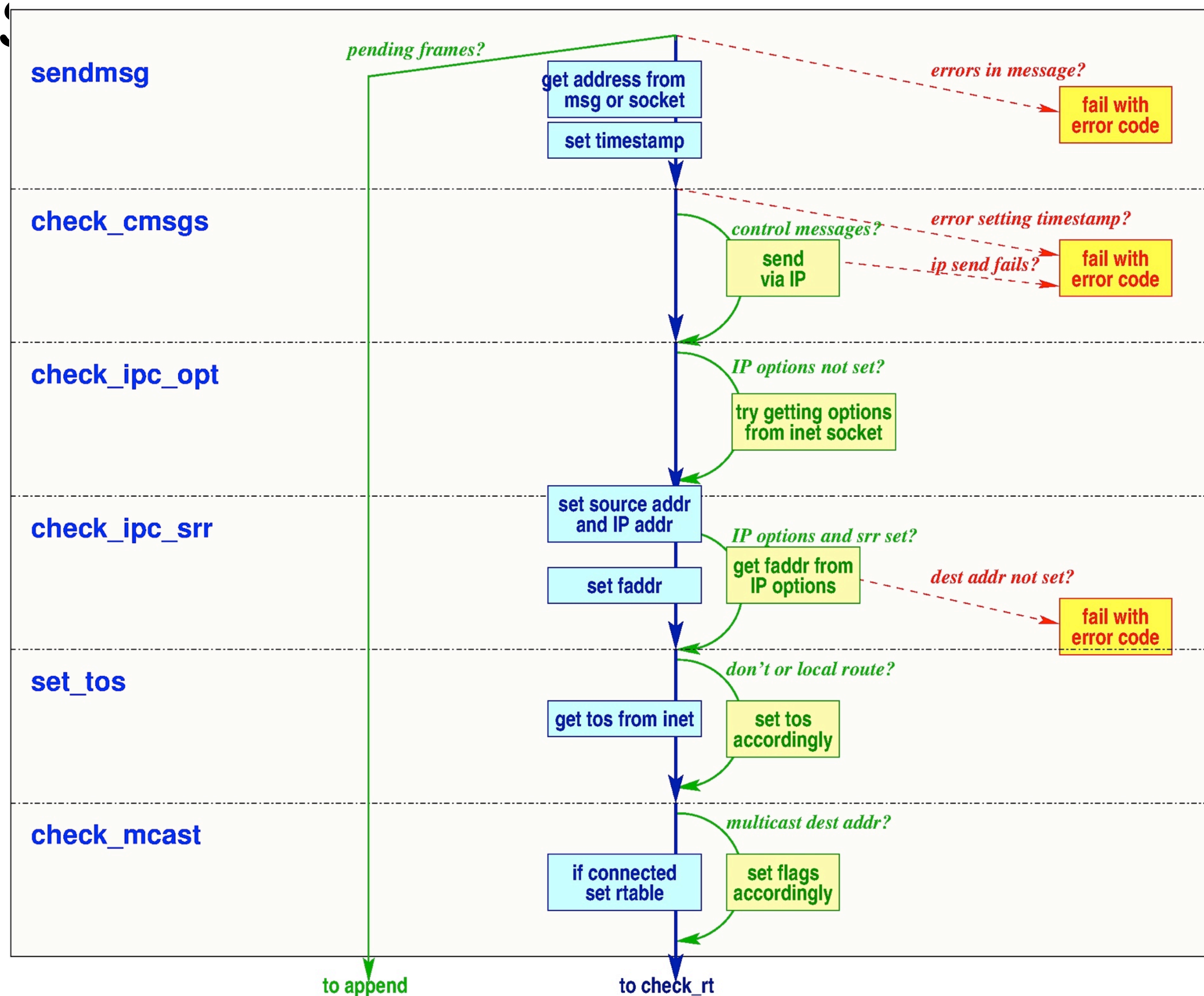
Initial Specification: disjunctive set of cases

Current Refinement Sequence: udp_rcv

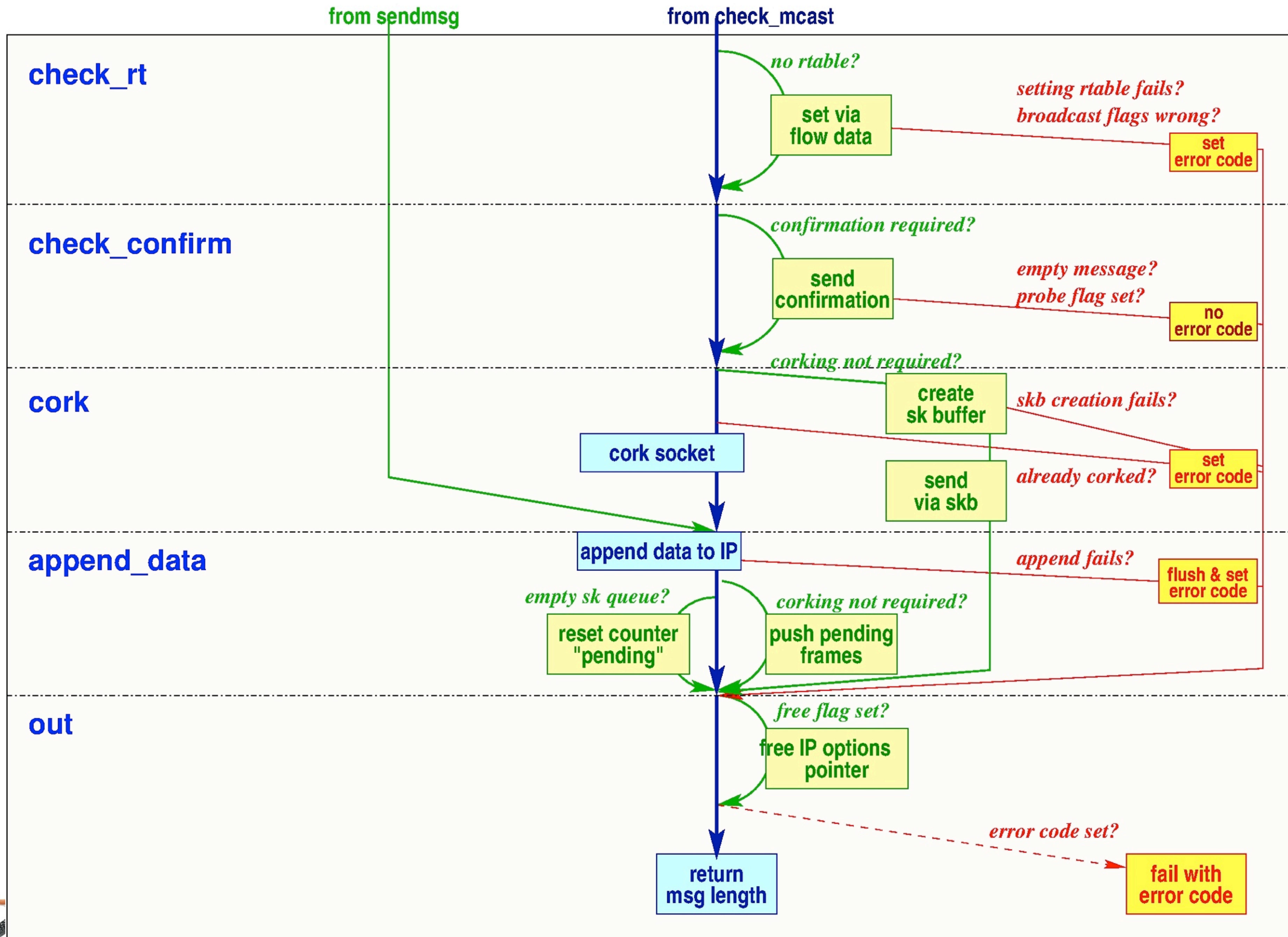
- MQ. mergeRules + QE for udp_queue_rcv_skb
- MR. mergeRules + QE for udp_rcv
- R1. introduce disjunctive definitions
- OR1. Introduce definitions for observers
- Cot1. finalizeCotype Host
- Defs1. Introduce definitions for testing
- Prep. Prepare for C generation
- Cgen. Generate C code

CGEN1. substBaseSpecs;
CGEN2. normalizeTopLevelLambdas;
CGEN3. instantiateHOFns;
CGEN4. removeCurrying;
CGEN5. liftUnsupportedPatterns;
CGEN6. translateMatch true
CGEN7. lambdaLift;
CGEN8. expandRecordMerges;
CGEN9. letWildPatToSeq;
CGEN12. conformOpDecls;
CGEN13. encapsulateProductArgs
CGEN14. introduceRecordMerges;
CGEN17. addEqOps;
CGEN18. addTypeConstructors;
CGEN19. deconflictupdates
CGEN20. makeUpdatesStateful
CGEN21. globalize;
CGEN22. simplifySpec;
CGEN23. adjustElementOrder
CGEN24. removeGeneratedSuffixes;
CGEN25. liftSequences
CGEN26. emitCFiles

Structure of udp_sendmgs modules and cases (top half)



Structure of udp_sendmgs modules and cases (bottom half)



Generating Proofs of Refinements

Transformation t: Spec A ↦ <morphism σ, spec B, proof>

Spec A

spec morphism σ

generate proof obligations on σ

Metaslang formula

translate

conjecture

Isabelle proof checker

axioms of A are provable under translation to B

Spec B, proof term

translate spec and proof

proof

yes/no

MergeRules Transformation

- MergeRules is implemented as a recursive function, guided by the structure of the input, INP.
- There are 6 main cases
  1. Contra - contradiction
  2. Tauto - tautology
  3. Factor – factoring global constraints
  4. Local - factoring local definitions
  5. Case – branching on sum-type cases
  6. If - branching on logical cases

Proof Data Structure

```
type TraceTree =

   | Contra (CDNFRep * (List MSTerm) * List (Id * MSType))
                inputs     assumptions     vars

   | Tauto (CDNFRep * (List MSTerm) * List (Id * MSType))
               inputs      assumptions     vars

   | TLocal (MSTerm * CDNFRep * List MSTerm * List (List (Id*MSType) * MSTerm) *
             TraceTree * DNFRep * List (Id * MSType))

   | TFactoring  (MSTerm * CDNFRep * (List MSTerm) * List MSTerm * TraceTree * DNFRep * List (Id*MSType))
                  result     input      assumptions     factors      child     precond    vars

   | TCase (MSTerm * CDNFRep * (List MSTerm) * MSTerm *
             List (MSPattern * TraceTree) * DNFRep * List (Id * MSType))

   | Tif  (MSTerm * CDNFRep * (List MSTerm) * MSTerm * TraceTree * TraceTree * DNFRep * List (Id*MSType))
           input     assumptions  branch     term     true child  false child  precondition  vars
```

Generating Proof Scripts

- Specware generates an ISAR proof from the TraceTree value returned by Mergerules
- This is straightforward template instantiation

```
| TFactoring (res,inps,assumps,factors,sub,fail,vars) ->
    mkIsarProof spc isabelleTerm None sub indent ^
    indent ^ "from result have factors: \"" ^ isabelleTerm (mkAnd factors) ^ "\" by auto \n" ^
    indent ^ "(* Assumptions for subterm *)\n" ^
    indent ^ "from factors assumptions have assumptions' : \"" ^ (isabelleTerm (mkAnd (traceAssumptions sub))) ^ "\" by auto\n" ^
    indent ^ "(* Failures for subterm *)\n" ^
    indent ^ "from factors fails have fails' : \"" ^ (isabelleTerm (mkNot (dnfToTerm (traceFailure sub)))) ^ "\" by auto\n" ^
    indent ^ "from result factors  have result': \"" ^ (isabelleTerm (traceResult sub)) ^ "\" by simp\n"  ^
    indent ^ "have rfactor: \"" ^ (equant isabelleTerm sub) ^ "\" by (fact ok[OF assumptions', OF fails', OF result'])\n"  ^
    indent ^  "from factors assumptions rfactor show ?thesis by auto\n" ^
    uindent ^ "qed\n"
```

Resulting Proof (snippet)

```
proof - (* Factoring *)
  assume assumptions : "\<not> (kestrel_xfrm4_policy_check(h, sock, XFRM_POLICY_IN, skb))"
  assume fails : "\<not> False"
  assume result : "h_cqt = kfree_skb(h, skb) \<and> ret = - 1"
  have ok: "h_cqt = kfree_skb(h, skb)   \<and> ret = - 1
               \<and> \<not> (kestrel_xfrm4_policy_check(h, sock, XFRM_POLICY_IN, skb))
                ==> (\<not> False) ==> True ==> True"
  proof - (* Tautology *)
    assume assumptions : "h_cqt = kfree_skb(h, skb)  \<and> ret = - 1
                          \<and> \<not> (kestrel_xfrm4_policy_check(h, sock, XFRM_POLICY_IN, skb))"
    assume fails : "\<not> False"
    assume result : "True"
    show ?thesis by simp
  qed
  from result have factors: "h_cqt = kfree_skb(h, skb) \<and> ret = - 1" by auto
  (* Assumptions for subterm *)
  from factors assumptions have assumptions' : "h_cqt = kfree_skb(h, skb)
                  \<and> ret = - 1 \<and> \<not> (kestrel_xfrm4_policy_check(h, sock, XFRM_POLICY_IN, skb))"
                  by auto
  (* Failures for subterm *)
  from factors fails have fails' : "\<not> False" by auto
  from result factors  have result': "True" by simp
  have rfactor: "True" by (fact ok[OF assumptions', OF fails', OF result'])
  from factors assumptions rfactor show ?thesis by auto
qed
```

Synthesis Measurements

	udp_rcv lines of text	udp_sendmsg lines of text
domain specification	740	2200
metaprogram	500	1067
generated Metaslang	1025	2470
generated CommonLisp	2300	3000
generated C	220	955
Isabelle proof scripts	9300 lines, 350 proof steps	>20450 lines, 1500 proof steps

Summary

- **Elaborated Design Process**: Complex design requires multiple transformations/refinements
- **Specification style**: mixed algebraic/coalgebraic spec style supports design of stateful/concurrent algorithms & systems
- **New Transformations**: six new coalgebraically-oriented transformations
- **Proofs**: transformations automatically generate both refinement steps and their proofs
- **Synthesized working code**: plug-compatible UDP code for Linux kernel
  - udp_rcv
  - udp_send