The COMPASS base notations: CML and SysML

Alvaro Miyazawa

June 12, 2012

www.compass-research.eu
Introduction

CML

Motivation
The Notation
The Telephone Exchange
The Distributed Telephone Exchange
Analysis
Conclusions

SysML–CML link

Motivation
Overview
State machine diagrams
Conclusions

COMPASS

- Comprehensive Modelling for Advanced Systems of Systems.
COMPASS

- Comprehensive Modelling for Advanced Systems of Systems.
- Extend industrial tools and practice.
COMPASS

- Comprehensive Modelling for Advanced Systems of Systems.
- Extend industrial tools and practice.
- Support model based development.
COMPASS

- Comprehensive Modelling for Advanced Systems of Systems.
- Extend industrial tools and practice.
- Support model based development.
- Modelling language for SoS architectures and contracts.
COMPASS

- Comprehensive Modelling for Advanced Systems of Systems.
- Extend industrial tools and practice.
- Support model based development.
- Modelling language for SoS architectures and contracts.
  - CML
COMPASS

- Comprehensive Modelling for Advanced Systems of Systems.
- Extend industrial tools and practice.
- Support model based development.
- Modelling language for SoS architectures and contracts.
  - CML
  - SysML
COMPASS

- Comprehensive Modelling for Advanced Systems of Systems.
- Extend industrial tools and practice.
- Support model based development.
- Modelling language for SoS architectures and contracts.
  - CML
  - SysML
- Support through methods and tools.
SoS engineering lacks models and tools for

- system analysis
  - design
  - evolution
- contracts
  - uncovering
  - recording
- VDM and CSP/Circus
  - Object orientation
  - Time
  - Mobility
- Formal semantics
- Refinement
  - Contract compliance
Support for graphical modelling

- SysML
- Link from SysML to CML
- Extension of SysML with CML
- Traceability from CML to SysML
The Telephone Exchange
The Telephone Exchange

sd Single Exchange

«block»
Jim:Subscriber
«block»
:Exchange
«block»
Ana:Subscriber

call( Jim, Ana )
callok
startring
answer
answerok
startspeech
suspend
suspendok
unsuspend
unsuspendok
clear
clearok
sd
Single Exchange
The Telephone Exchange

types
  SUBS = token;
  STATUS = <connecting> | <engaged> | <ringing> |
          <speech> | <suspended>;

values
  Connected = {<ringing>,<speech>,<suspended>};
functions
callers: (map SUBS to STATUS) -> set of SUBS
callers(status) == dom(status :> Connected);

...
The Telephone Exchange

```
channels
    init;
    call: SUBS*SUBS;
    clear, suspend: SUBS;
    answer, unsuspend: SUBS;
    callok, clearok, suspendok, answerok, unsuspendok;
    callerror, clearerror, suspenderror, answererror;
    unsuspenderror;
    connectfree, connectbusy: SUBS;
```
The Telephone Exchange

process ExchangeProc = lSUBS: set of SUBS @ begin
  class Exchange
    instance variables
      status: map SUBS to STATUS;
      number: map SUBS to SUBS;
      subs: set of SUBS;
    inv
      exists n: nat & card(subs) = n and
      dom(status) = dom(number) and
      dom(number) subset subs and
      is_(dom(status:>Connected)<:number, inmap SUBS to SUBS) ;
  end
The Telephone Exchange

operations
  InitExchange ()
  ext wr status, number, subs
  post
    status = { |-> } and
    number = { |-> } and
    subs = 1SUBS;

  Add (s: SUBS)
  ext wr subs
  post
    subs = subs ~ union {s};
The Telephone Exchange

Call \((s,t: \text{SUBS})\)

ext wr status, number
rd subs

pre
s in set subs and
t in set subs and
s in set free(status,number,subs)

post
status = status \sim ++ \{s \rightarrow <\text{connecting}>\} \text{ and }
number = number \sim ++ \{s \rightarrow t\};

Clear \((s,t: \text{SUBS})\)

ext wr status, number
rd subs

pre
s in set subs and
t in set subs and
s in set dom(status) and
t = number(s)

post
status = \{s\}<-: status \sim \text{ and } number = \{s\}<-: number \sim;
The Telephone Exchange

Suspend(s, t: SUBS)
...

ConnectFree(s, t: SUBS)
...

ConnectBusy(s, t: SUBS)
...

Answer(s, t: SUBS)
...

Unsuspend(s, t: SUBS)
...

end Exchange
The Telephone Exchange

**state** Exchange

**actions**

Exch = (  
  call?s:(s *in set* subs)?t:(t *in set* subs) ->  
    (if s *in set* free(status, number, subs)  
      then callok -> Call(s,t)  
      else callerror -> SKIP  
    ); Exch  
  ); Exch

[] clear?s:(s *in set* subs) -> ...  
[] suspend?t:(t *in set* subs) -> ...  
[] connectfree?s:(...) -> ...  
[] connectbusy?s:(...) -> ...  
[] answer?t:(t *in set* subs) -> ...  
[] unsuspend?t:(t *in set* subs) -> ...  

); Exch

@ init -> InitExchange(); Exch

end
The Distributed Telephone Exchange
The Distributed Telephone Exchange
The Distributed Telephone Exchange
The Distributed Telephone Exchange

types
MSG = <vcallmsg> | <vclearmsg> | <callokmeg> | <callerrormsg>;

values
n: nat = undefined;
LOC = {0,...,n-1};
The Distributed Telephone Exchange

functions
next (i: nat) j: nat

pre
  i in set LOC

post
  j = (i+1) mod n;
The Distributed Telephone Exchange

channels
my.init;
my.call: SUBS*SUBS;
my.clear, my.suspend: SUBS;
my.connectfree, my.connectbusy: SUBS;
my.answer, my.unsuspend: SUBS;
my.callerr, my.clearerr, my.suspenderr;
my.answererr, my.unsuspenderr
process NodeProc = i: nat @ begin
  class Node
    instance variables
    away: map SUBS to nat;
    origin: map SUBS to nat;
    inv
    rng(away) subset LOC and
    rng(origin) subset LOC and
    dom(away) subset home(i) and
    dom(origin) inter home(i) = {}

    operations
    InitNode ()
    ext wr away, origin
    post
    away = {|->} and
    origin = {|->};
  end Node
**The Distributed Telephone Exchange**

```plaintext
state Node
actions
    Node = mu X @ ( 
        call?s:(s in set home(i) \ (dom (away)))
        ?t:(t in set home(i) \ (dom (away))) ->
            my.call!s!t -> 
                (my.callokok -> callokok -> SKIP 
                   [] my.callerror -> callerror -> SKIP
                )
        [] call?s:(s in set dom (away))?t -> ... 
        [] call?s:(s in set home(i))
            ?t:(t not in set home(i)) -> ...
        [] clear?s:(...)?t:(...) ->
        [] ...
        [] link.i.vcallmsg?j?k?s?t ->
        [] ...
    ); X

@ init -> my.init -> InitNode(); Node
end
```
The Distributed Telephone Exchange

process LinkProc = i: nat @ begin
    @ mu X @ mid.i?msg?l?m?s?t ->
        link.next(i)!msg!l!m!s!t -> X
end

process NodeLinkProc = i: nat @
(NodeProc(i)
    [||mid.i||]
    LinkProc(i))\{|mid.i|}

process NodeLinkExchange = i: nat @
    NodeLink(i) \ ExchangeProc(i)

process ExchangeSoS = || i: LOC @
    [RingChannels(i)] NodeLinkExchange(i)
Meaning of

```plaintext
process SlaveExchangeProc = i: LOC @
  ExchangeProc(home(i))[
    call <- my.call, clear <- my.clear,
    suspend <- my.suspend,
    connectfree <- my.connectfree,
    connectbusy <- my.connectbusy,
    answer <- my.answer,
    unsuspend <- my.unsuspend,
    callok <- my.callok,
    clearok <- my.clearok,
    suspendok <- my.suspendok,
    answerok <- my.answerok,
    unsuspendok <- my.unsuspendok,
    callerror <- my.callerror,
    clearerror <- my.clearerror,
    suspenderror <- my.suspenderror,
    answererror <- my.answererror,
    unsuspenderror <- my.unsuspenderror]
```
chansets
SlaveChannels = { my.call, my.clear, my.suspend,
my.connectfree, my.connectbusy, my.answer,
my.unsuspend, my.callok, my.clearok, mu.suspendok,
my.answerok, my.unsuspendok, my.callerror,
my.clearerror, my.suspenderror, my.answererror,
my.unsuspenderror |}

process NodeLinkExchange = i: nat @
(NodeLink(i)
 [ | { | SlaveChannels |} |]
SlaveExchangeProc(i))\SlaveChannels
Analysis

- Is ExchangeSoS deadlock-free?
- Is ExchangeSoS divergence-free?
- Does ExchangeSoS refine TelephoneExchange?
Tool support

- Simulator
- Model-checker
- Theorem prover
Conclusions

- CML – notation for modelling and reasoning about SoS.
  - Contracts between constituent systems
  - Architecture
- CML is well suited for
  - automatic analysis
  - refinement based analysis
Motivation

- **SysML**
  - Graphical notation
  - Focus on architecture
  - Used in industry
  - Informal/incomplete semantics
  - Tool support

- **CML**
  - Textual notation
  - Architecture and Contracts
  - Not well received in industry*
  - Formal semantics
  - Tool support**
Overview

channels
  start_Op1: X
  end_Op1: Y
  get_i, set_i: A
  get_j, set_j: B
  get_b, set_b: C

class A =
  begin
  state i,j: int; ... A
  process Model = Block_A || stm_A
  process Consistency1 = Model || seq
  process Consistency2 = Model || Act1
  act Act1
State machine diagrams

- Translation strategy
  - Simple and composite states
  - Transitions and completion events
  - Final and junction pseudostates
  - Join and fork pseudostates

- Translation rules
  - SysML/UML meta-model
  - CML
CML Model

types
id = \langle t_1 \rangle \mid \langle t_2 \rangle \mid \langle t_3 \rangle \mid \langle m_1 \rangle \mid \langle s_1 \rangle \mid \langle s_2 \rangle \mid \langle r_{11} \rangle \mid \langle r_{12} \rangle \mid \langle s_{111} \rangle \mid \langle s_{112} \rangle \mid \langle s_{121} \rangle \mid \langle s_{122} \rangle

values
transitions = \{\langle t_1 \rangle, \langle t_2 \rangle, \langle t_3 \rangle\}
states = \{\langle m_1 \rangle, \langle s_1 \rangle, \langle s_2 \rangle, \langle r_{11} \rangle, \langle r_{12} \rangle, \langle s_{111} \rangle, \langle s_{112} \rangle, \langle s_{121} \rangle, \langle s_{122} \rangle\}

types
signal = \langle \text{sig1} \rangle \mid \langle \text{sig2} \rangle
data = \langle \text{empty} \rangle

channels
in\_event: id \times signal \times data
off_s1 = enter?x!s1 -> setstate!s1!s1!true -> SKIP;
entered!x!s1 -> (instate!s1!r11?b -> (if b = false then enter.s1.r11 -> entered.s1.r11 -> SKIP else SKIP)
||| instate!s1!r12?b -> (if b = false then enter.s1.r12 -> entered.s1.r12 -> SKIP else SKIP))
); on_s1
on_s1 = a_s1/\(\text{exit?x!s1} \rightarrow \text{setstate!s1!s1!false} \rightarrow (\)

\text{instate!s1!r11?b} \rightarrow (\text{if } b = \text{true} \text{ then exit!s1!r11} \rightarrow \text{exited!s1!r11} \rightarrow \text{SKIP} \text{ else SKIP})
\)

\text{|||}

\text{instate!s1!r12?b} \rightarrow (\text{if } b = \text{true} \text{ then exit!s1!r12} \rightarrow \text{exited!s1!r12} \rightarrow \text{SKIP} \text{ else SKIP})
\)

); \text{exited!x!s1} \rightarrow \text{SKIP}
\)

); \text{off_s1}
a_s1 = in_event!s1?d ->

dcl b1,b2: bool @ ( 
    in_event.r11.s.d -> consumed.r11?b -> b1:=b
    [|| {b1} | {b2} ||]
    in_event.r12.s.d -> consumed.r12?b -> b2:=b
); 
(if card({b: {b1,b2} | b = true}) > 0 
    consumed.s1.true -> SKIP 
else in_event.t1.s.d -> enabled.t1?b -> 
    consumed.s1.b -> SKIP 
); 
a_s1
in_s1 = (dcl b: bool := false @
    (mu X @ instate?x!s1!b -> X
        []
            setstate?x!s1?y -> b:=y; X
        )
    )
)

state_s1 = (in_s1
    []{{|instate.s1.s1,setstate|}}]
    off_s1)\{|instate.s1.s1,setstate|}{
trans_t1 = (in_event!t1?s?d -> x!t1?vx -> ( 
    if s == sig1 and vx <= 0 
    then enabled.t1.true -> fire -> 
        exit!t1!s1 -> 
        exited!t1!s1 -> enter!t1!s2 -> 
        entered!t1!s2 -> fired -> SKIP 
    else enabled.t1.false -> fire -> fired -> SKIP 
    ) 
    []
    fire -> fired -> SKIP); trans_t1
machine_m1 = enter!m1!s1 -> entered!m1!s1 ->
    (mu X @
     in_event!m1?{s,d} ->
     in_event?x:{s1,s2}!s!d ->
     consumed!x?b -> fire -> fired -> X)
model = (let
    Hidden = {|in_event.x, enabled, consumed
        enter, entered, exit, exited,
        fire, fired | x <- id\{m1} |}
    Actions = {
        (machine_m1, chan_m1),
        (state_s1, chan_s1), (state_s2, chan_s2),
        (state_r11, chan_r11), (state_r12, chan_r12),
        (state_s111, chan_s111), (state_s112, chan_s112),
        (state_s121, chan_s121), (state_s122, chan_s122),
        (trans_t1, chan_t1), (trans_t2, chan_t2),
        (trans_t3, chan_t3)
    }
    in
      (|| (A,CS): Action @ [CS] A)\Hidden
  )
Conclusions

- Initial translation strategies for
  - Sequence diagrams
  - Activity diagrams
  - State machine diagrams
- Integration
  - block definition diagrams
  - internal block diagrams
- CML in SysML: parametric diagrams
- Traceability: requirement diagrams