XCD

Design-by-Contract for Reusable Components & Realizable Architectures

Mert Ozkaya & Christos Kloukinas

CITY UNIVERSITY LONDON

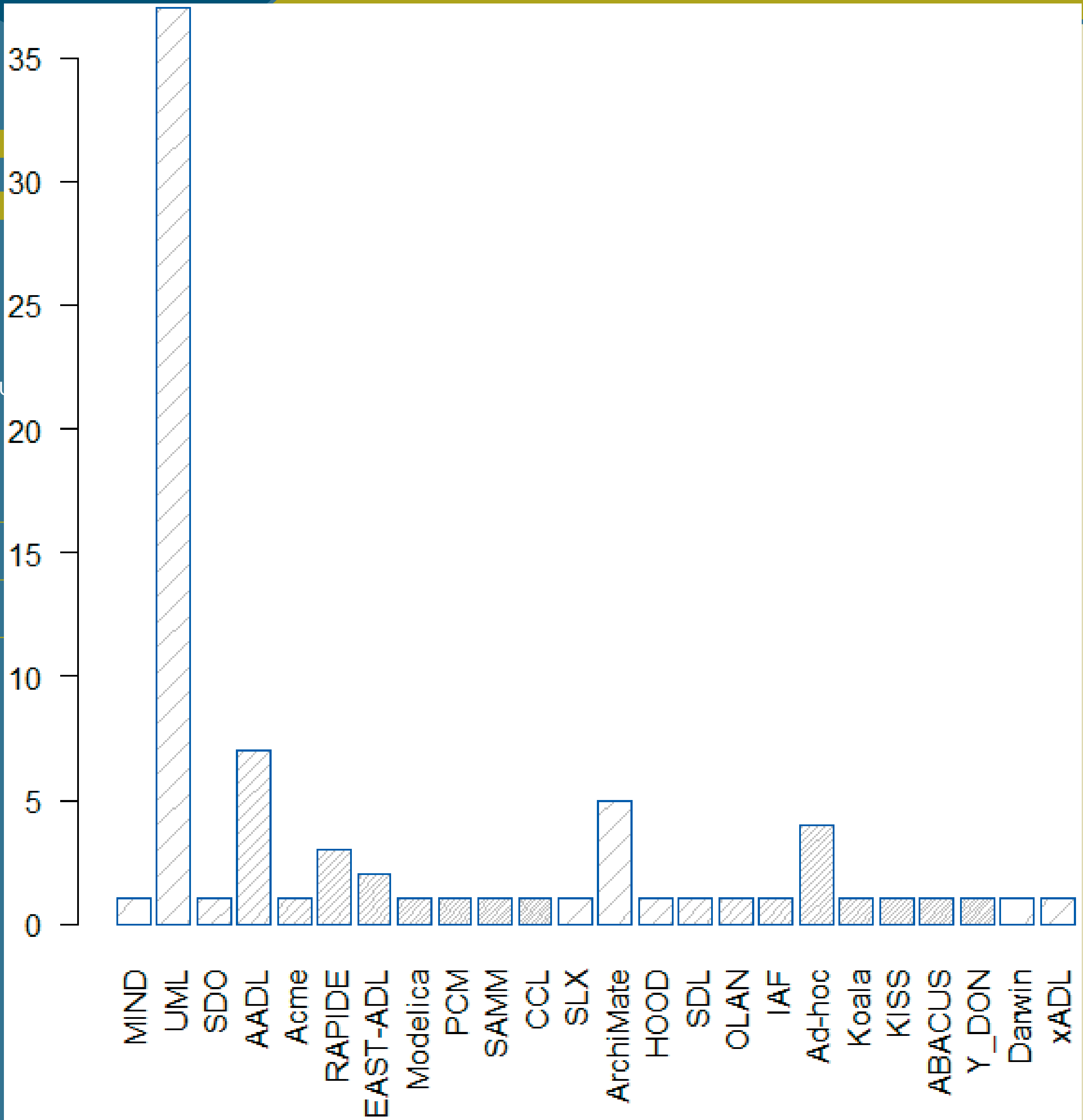
EST 1894

Software Architecture: Beginnings

Two main approaches:

- Components & Connections
  - Darwin 1996
- Components & Connectors
  - Wright 1997

Current State



- Malavolta et al., IEEE TSE v. 39, n. 6 “What Industry Needs from Architectural Languages: A Survey” *(“needs” or “asks for”?)*
- No “steep learning curve” (UML), performance, reliability, . . .

Issues

- Formal process algebras not practitioners' cup of tea. . .
  - Practitioners care about Performance/Reliability/etc.

? But Performance/Reliability @ Deadlocked States = ?

- Components + Connections
  - No Connectors
  - Components must describe the protocols they'll be used with

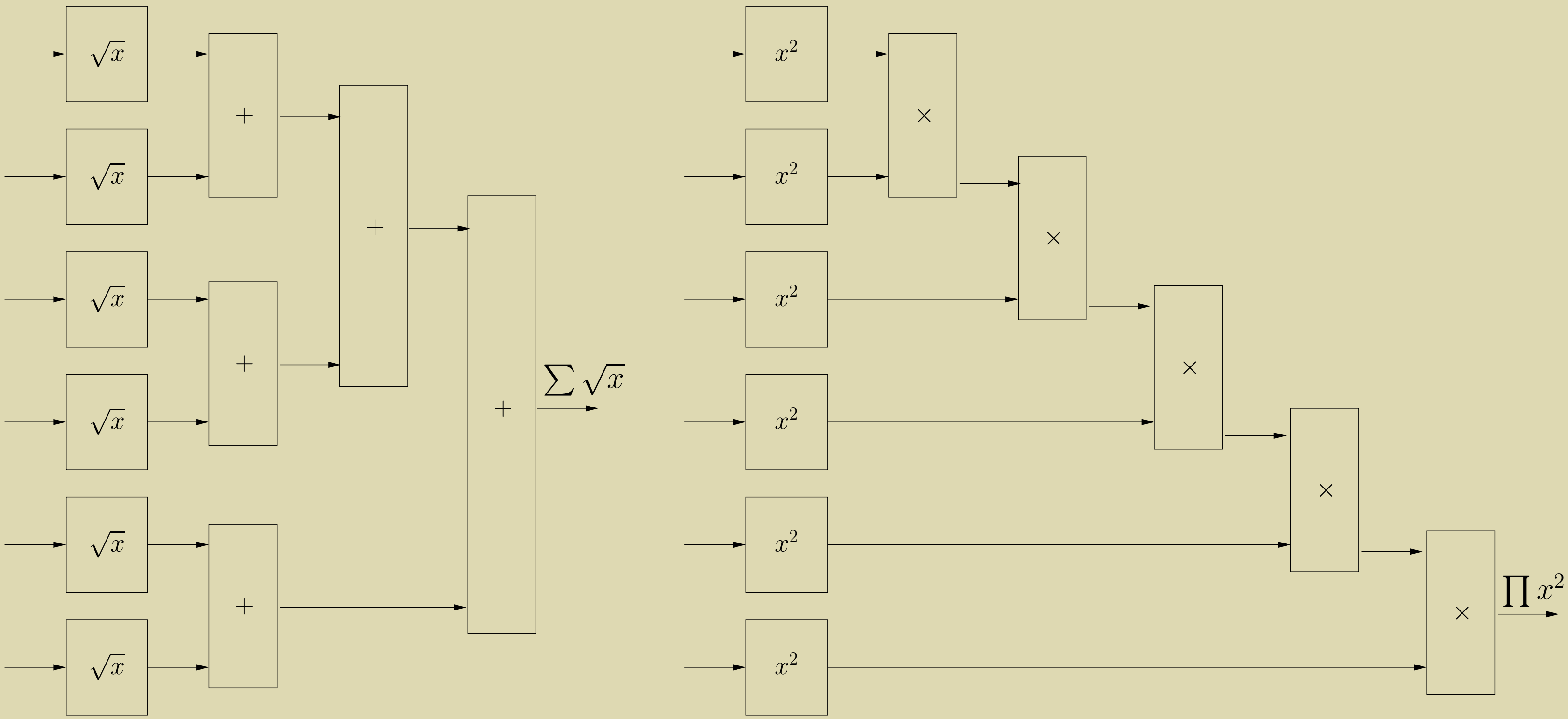
? Modularity, Reusability, Architectural Exploration?

? Architectural Mismatch?

No Connectors?

- "All systems can be specified without connectors, therefore connectors are not needed"

Alice (Monday): $\sum \sqrt{x}$ Bob (Friday): $\prod x^2$



- Yes, but what about map-reduce? **`(reduce R (map M x))`**
- Why re-invent the wheel each time?
  - We'll not get it right each time
- Define it once and Reuse-by-Calling it not Reuse-by-Copying it

Connectors

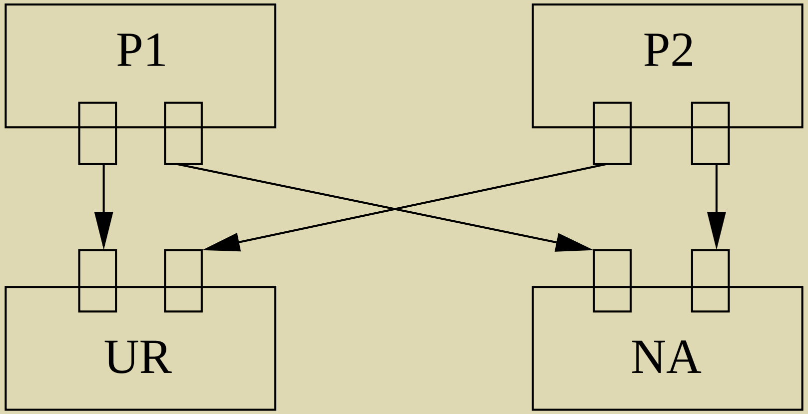
- Specify protocols once, reuse them by calling them
- Components become protocol-independent
- Components are more modular, easier to specify
  - Less work $\rightsquigarrow$ more specs $\rightsquigarrow$ architectural mismatch less likely
- Increase reusability of components & connectors (vector + bubble/quick/merge/. . . sort)
- Easier to verify each independently
- Easier to understand the system structure
- Easier to explore alternatives

Connector Realizability – Their Achilles' Heel. . .

- Wright defined the base notions: Roles + Glue
- Components implement Roles
- System = Components $\|$ Glue
- All ADLs supporting connectors follow Wright but . . .
- Glue adds global constraints

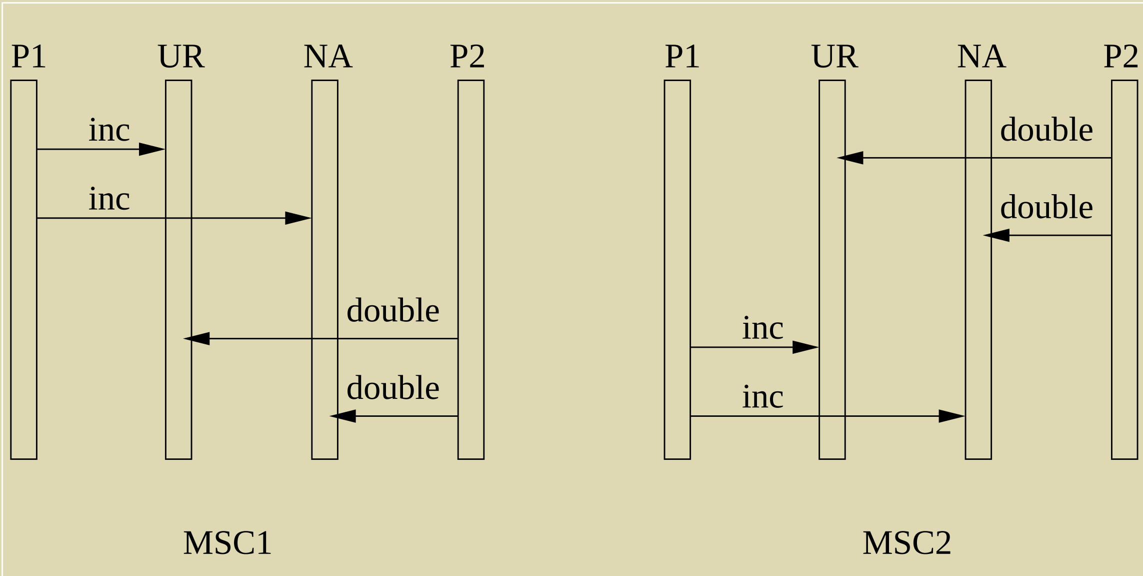
  $\Rightarrow$ Unrealizable in general
  - If we want to preserve "communication integrity"
  - And we do – most analyses depend on it

Unrealizable Architecture – A Nuclear Plant

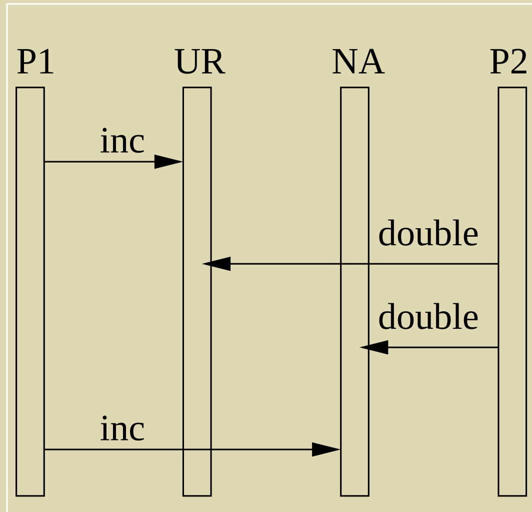


(a) A decentralized architecture

[AEY03] R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. IEEE TSE, 29(7):623-633, 2003



(b) The plant's (unrealizable) MSCs



(c) An unavoidable bad behaviour

Unrealizable Wright Connector


```
connector Plant_Connector =
role P1 = ur → na → P1. // increase both
role P2 = ur → na → P2. //   double both
role UR = inc → UR □ double → UR.
role NA = inc → NA □ double → NA.
glue G = P1.ur→UR.inc → P1.na→NA.inc
         → P2.ur→UR.double → P2.na→NA.double
         → G
       ⊓ P2.ur→UR.double → P2.na→NA.double
         → P1.ur→UR.inc → P1.na→NA.inc
         → G. // → link, → global constraint
SYSTEM = (P1:P1 ∥ P2:P2 ∥ UR:UR ∥ NA:NA ∥ G).
```

Wright's *(unrealizable)* connector for the nuclear plant

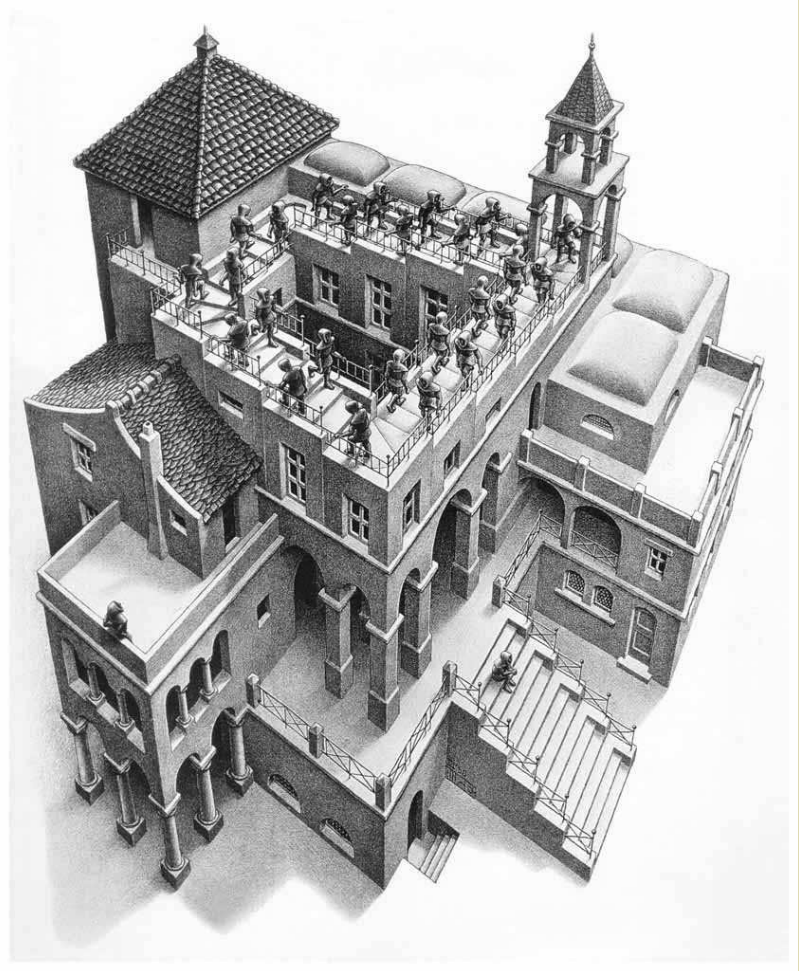
- Property is verified!
- But no way to realize it while preserving *comm. integrity* . . . :-(
- Architect needs to be told requirement isn't satisfied

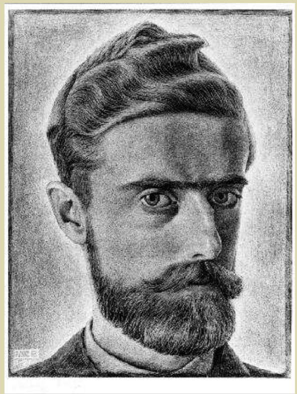
  ⇒ *Global constraints are requirements*
- Architecture shouldn't simply repeat the requirements

Requirements vs Architecture

Requirement "I want infinite stairs"

Architecture





M. C. Escher
(1898 - 1972)

- Architecture needs to be realizable

  In a way that respects communication integrity

- Otherwise, performance/reliability/etc analyses are invalid

Realizable Plant

Problem
Beginnings
State
Issues
No Connectors?
Connectors
Realizability
Unrealizable Architecture
Unrealizable Wright Connector
Req vs Arch
Realizable Plant

XCD

Conclusions

Extras


```
connector Realizable_Plant_Connector =
role P1 = ur → na → P1.                    // same
role P2 = ur → na → P2.                    // same
role UR = inc → UR □ double → UR.          // same
role NA = inc → NA □ double → NA.          // same
glue G = (G1 ‖ G2 ‖ G3 ‖ G4).// Real glue
// where Gi's are simple links:
G1= P1.ur → UR.inc     →G1.
G2= P1.na → NA.inc     →G2.
G3= P2.ur → UR.double  →G3.
G4= P2.na → NA.double  →G4.
SYSTEM = (P1:P1 ‖ P2:P2 ‖ UR:UR ‖ NA:NA ‖ G).
GlueProperty = UR.inc      → NA.inc
                → UR.double → NA.double
               → GlueProperty
             □  UR.double → NA.double
                → UR.inc    → NA.inc
               → GlueProperty.
```

- *Of course, GlueProperty is no longer satisfied. . .*
- At least now we *know* that it doesn't work!
- And so do the designers/developers/clients . . .

XCD: Main Ideas

- Support arbitrary connectors
  - ⇒ Modular specifications
    Simpler component specifications
  - ⇒ Reusable components & reusable connectors
- Only local constraints can be imposed
  - ⇒ Realizable architectures + communication integrity
- Programming language-like syntax
  - ⇒ Not (too) scary (?)
- Formal (extended Design-by-Contract approach – JML-inspired)
  - ⇒ Can verify (with SPIN):

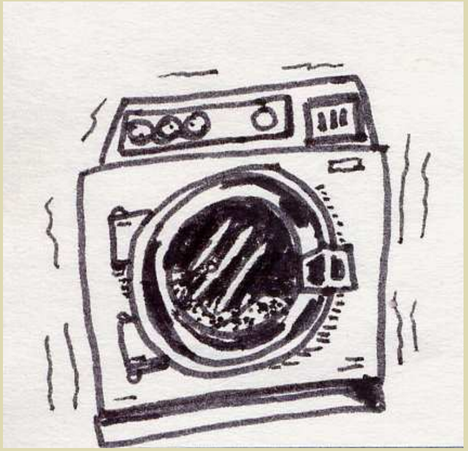
1. Are provided services (local) interaction constraints satisfied?
2. Are provided services functional pre-conditions complete?
3. Are there race-conditions?
4. Do event buffer sizes suffice? and
5. Are there (global) deadlocks?

Plus, general LTL properties (new)

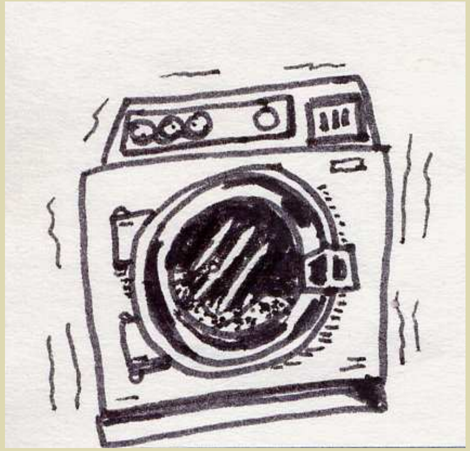
Component Interaction Constraints

Cleanliness is next to Godliness. . .

Machine A



Machine B



Same interfaces but. . .

```
@interaction {
    waits: ! washing; }
void open_door();
```

A blocks my call till it's safe

```
@interaction {
    accepts: ! washing; }
void open_door();
```

B may reject it with undefined behaviour (might electrocute me!)

Java Thread as an XCD Component


```
component Thread {
  bool started := false; // component data.
  bool died := false;

  aliveP() {return started && !died;} // helper function

  provided p {
    @functional  {ensures: \result := aliveP();}
    bool isAlive();

    @interaction {waits: !aliveP();}
    void join();

    @interaction {accepts: ! started;}
    @functional  {ensures: started := true;}
    void start();
    // ... other methods
  };
};
```

?

- Constraints are more modular *(JML allows but doesn't enforce it)*
- Functional constraints must be complete! Call already accepted!!!

Software Components & DbC

- Software Components have:
  - Provided method ports ✔
  - Required method ports ✘
  - Consumer event ports ✘
  - Emitter event ports ✘
- DbC created for objects
  - No events
  - No required methods
  - Ignores clients' needs
- Separation of interaction/functional contracts

  Restaurant Provide a service from 7pm-11pm – Italian menu

  Client Require a service from 9pm-12pm – pizza or Peking duck
- Extension for required methods & event consumption/emission

```
provided p {
 @interaction {waits/ accepts: φ1;}
 @functional {

   requires: φ2;
   ensures: d := f(d,p);}
 void service(p); };
```

```
required p {
 @interaction {waits: φ1;}
 @functional {
   promises: p := f1(d);
   requires: φ2;
   ensures: d := f2(d);}
 void service(p); };
```

Some Choices

- Ensures are assignments (possibly non-det), not formulæ:

	$z := x/y$	vs	$x = y * z$
$z \in [0, 255]$:	1 state	vs	256
	$x \in [0,n]; y \in [0, n-x]; z \in [0, n-x-y];$	vs	$0 \le x+y+z \le n$
	$(n+1)(n^2+5n+6)/6$ states	vs	$(n+1)^3$
n = 255:	2.8 M states	vs	16.7 M

- req_1/ens_1 \XXX req_2/ens_2 $\equiv$
  - **JML –** **\also** $((\mathrm{req}_1 \rightarrow \mathrm{ens}_1) \wedge (\mathrm{req}_2 \rightarrow \mathrm{ens}_2))$
  - **XCD –** **\otherwise** $((\mathrm{req}_1 \rightarrow \mathrm{ens}_1) \vee (\mathrm{req}_2 \rightarrow \mathrm{ens}_2)) \wedge (\bigvee_i \mathrm{req}_i)$
  - Why otherwise instead of also:
    - Avoid conflicting/circular assignments
    - Introduce frames for race-condition checks
  - $(\bigvee_i \mathrm{req}_i)$: action is accepted from interaction constraints

    $\Rightarrow$ req_i must be complete

- Roles cannot reject an action, only disable it (**waits**, no **accepts**)

XCD Structure & Roles


```
Basic Component: Type Param* Data* HelperFunction* Port*;
  Port: Provided | Required | Emitter | Consumer;
  P/R/E/C: Action*;
  Action: IntCon? FunCon? Signature;

Connector: HOParam+ Param* Role* ConInst*;
  Role: Type Param* Data* HelperFunction* Port*; // Like Compon

Composite Component: Param* CompInst* ConInst*;
```

Higher-Order Composite Components

Connectors = +

Interaction Constraints

- XCD roles add extra data and port constraints to components
- Component port must have all role port actions

  Wright: Component ports *implement* role ports

  XCD: Component ports *interpret* role ports

- *Role's not a wrapper – it's a script to be interpreted*
  - *Wrappers cannot disable active (required/emitter) actions*

Translator to SPIN's ProMeLa


```
CompA aInst;  CompB bInst;
ConnC cInst(aInst, bInst);
```

- Each basic comp. instance (aInst, bInst), a ProMeLa process

```
do
:: port_I_atomic_action_J_translation
:: port_K_nonatomic_action_M_translation_partA
:: port_K_nonatomic_action_M_translation_partB
od
```

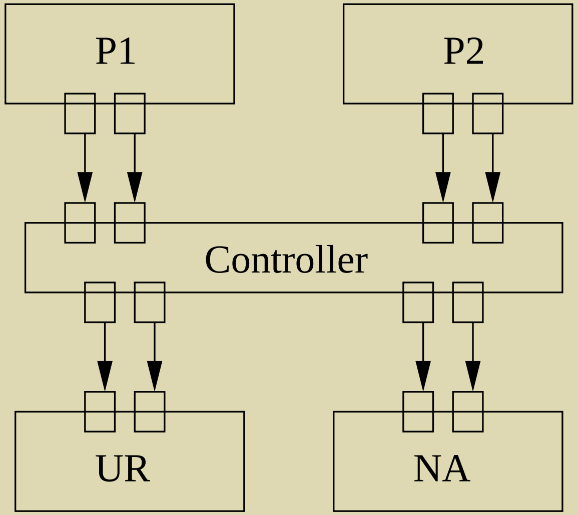
- Ports allow at most one action to be active at any time
- Each action is guarded by its own waits/accepts guard, *and by all the waits guards of its roles*
- Event emission/consumption actions are atomic
- Required methods are two atomic blocks (send request, receive response)
- Provided methods are either atomic (req/comp/res) or two atomic blocks (req/comp, comp/res)
  - Non-atomic provided method used for call chaining
- Non-atomic methods encoded so as to catch race-conditions

Component ProMeLa Structure


```
::atomic { // a provided port method
  port.Channel_req ? method.Args
       : roleWaits && method.Waits && method.Accepts  ->
     assert(FunctRequires); // Ensure functional completeness
     calcData(...);
     port.Channel_res ! method.Args;
  }
::atomic { // same provided port method
  port.Channel_req ? method.Args
       : roleWaits && ! method.Accepts  ->
     assert(False); // Request rejected - CHAOS
  }
...

::atomic { // sending a request - a required port
  selectParams(method.Args,
     roleWaits && methodWaits && ! port.Lock , ...) ->
    port.Lock = method;
    port.Channel_req ! method.Args;
  }
::atomic { // receiving a response - same required port
  port.Channel_res ? method.Args
       : port.Lock == method  ->
    calcData(...);
    port.Lock = 0;
  }
```

Centralized Plant




```
enum ordr := {none, incFirst,
                    dblFirst};
role roleController {
  ordr order := none;
  bool p1_incNARcvd := false;
  bool p1_incURRcvd := false;
  bool ur_incUREmtd := false;
  bool na_incNAEmtd := false;
 all_received(){return
    p1_incURRcvd && p1_incNARcvd
 && p2_dblURRcvd && p2_dblNARcvd;}

provided port_variable P1toUR {
  @interaction{
  waits: !p1_incURRcvd;
  ensures: p1_incURRcvd :=true;
    order := pre(order) == none
      ? incFirst : pre(order); }
  void incUR(); }
```

```
provided port_variable P1toNA {
  @interaction{
  waits:  !p1_incNARcvd;
  ensures: p1_incNARcvd :=true; }
  void incNA(); }
required port_variable CtoURinc {
  @interaction{
  waits: all_received()
    && !ur_incUREmtd
    && ( (order==incFirst)
      || (order==dblFirst
            && dbl_emitted()) );
  ensures: ur_incUREmtd := true; }
  void incUR(); }
required port_variable CtoNAinc {
  @interaction{
  waits: ur_incUREmtd && !na_incNAEmtd;
  ensures: // clear flags if dblFirst
    p1_incURRcvd
      := !(pre(order) == dblFirst);
    ...
    ur_dblUREmtd
      := pre(order) == dblFirst
         ? false : pre(ur_dblUREmtd);
    na_dblNAEmtd
      := pre(order) == dblFirst
         ? false : pre(na_dblNAEmtd);
    order
      := pre(order) == dblFirst
         ? none : pre(order); }
  void incNA(); }
```

Evaluation

Case Study	Issues	State-vector (Bytes)	States Stored	States Matched	Memory (MB)	Time (sec)
Decentralized Nuclear Plant	glue	240	137	73	130	0.00
Centralized Nuclear Plant		424	168349	407776	186	1.21
Lunar Lander v. 1	Ovrflw	372	118	78	131	0.01
Lunar Lander v. 2		392	4223125	8072166	3793	15.50
Gas Station (1 customer)		188	1003	1401	130	0.00
Gas Station (2 customers)		288	1136214	2793961	382	3.23
Gas Station (3 customers)		368	25056808	89254880	7024†	78.00
BITSTATE (3 customers)		368	62792292	207452380	24	242.00
BITSTATE (4 customers)		456	66989014	289982810	25	321.00
BITSTATE (5 customers)		544	69607515	356984080	26	365.00
Aegis v. 1	L-DDLCK	620	13834057	71301546	7024†	52.00
BITSTATE v. 1	L-DDLCK	620	64408848	266469200	37	330.00
BITSTATE v. 2		548	63568962	268078040	35	304.00
English auction v. 1 (1 part.)	DDLCK, Ovrflw	140	296	295	130	0.00
English auction v. 2 (1 part.)	Ovrflw	144	776	1642	130	0.00
English auction v. 2 (2 part.)	Ovrflw	232	1293488	3732650	367	5.00
English auction v. 2 (3 part.)	Ovrflw	312	27315867	96797687	7024†	134.00
BITSTATE v. 2 (3 part.)	Ovrflw	312	57105380	189090640	20	310.00

"States Stored" unique global states.
"States Matched" states revisited during the search.
† run out of memory (7024 MB).

All case studies available at the XCD web site

Conclusions

XCD – a new ADL

- Support for connectors
  - Arbitrary, complex connectors
  - Always realizable
- Formal
  - Architectures can be model-checked
  - Reasonable results for a number of case studies
- A less steep learning curve (?)
  - Closer to a programming language than process algebras
  - Extended DbC

Future Work

- Model optimization (generalize SPIN's conditional message reception)
- Tool
  - Support recursive connectors
  - Support comp/port arrays in connectors
- *Verification of components (OK) & connectors (??) in isolation*
  - Construct testing environments to complete a sub-system
- *Compute interface of a composite component*
  - Spin + learning? Not so good – complex interfaces. . .
  - SMT solvers?
- Property language
  - LTL not so easy, even with KSU's SAnToS lab's LTL patterns
  - Observer processes can be used
- Extend
  - Timed, Probabilistic, Hybrid automata

Thank You!

Extras – Verifying Arbitrary Connectors in Isolation

Verifying Connectors – A generic C++ Algo


```
#include <iostream>
using namespace std;
template <typename T>
void myswap(T & x, T & y) { T tmp = x; x = y; y = tmp; }

int main() {
  int i = 3, j = 5;
  cout << i << '␣' << j << endl; // 3 5
  myswap(i, j);
  char c; cin >> c;
  cout << i << '␣' << j << endl; // 5 3 myswap swaps!

  return 0;
}
```

Swapping Algo


```
template <typename T>
void myswap(T & x, T & y) { T tmp = x; x = y; y = tmp; }
```

```
  T { provides: Value copy(); }
    { requires: Value copy(); }
```

```
tmp: request a copy from (x) -> accept copy request from (y);

  x: accept copy request from (tmp) -> request a copy from (y);

  y: accept copy request from (x) -> request a copy from (tmp);
```

- Algo: A set of ordering constraints on functions

A generic C++ Algo – B

```
#include <iostream>
#include <string>
using namespace std;
template <typename T>
void myswap(T & x, T & y) { T tmp = x; x = y; y = tmp; }

class person {
  string name; int age;
 public:
  person(string n, int a) : name(n), age(a) {}
  ostream &print(ostream &o) const;
};
ostream &operator<<(ostream &o, const person &p)
 {return p.print(o);}//

int main() {
  person p1("alice", 30), p2("bob", 19);
  cout << p1 << endl << p2 << endl;
  myswap(p1, p2);
  char c; cin >> c;
  cout << p1 << endl << p2 << endl;

  return 0;
}
```

```
alice is 30 years old.
bob is 19 years old.
.
bob (not really) is 39 years old.
alice (not really) is 34 years old.
```

A generic C++ Algo – C


```
ostream &person::print(ostream &o) const {
  return o << name << "␣is␣" << age << "␣years␣old.";
}
person & person::operator=(const person &o) {
  name = o.name + "␣(not␣really)";
  age = age + o.age/2;
  return (*this);
}
```

Swapping Algo – B


```
template <typename T>
void myswap(T & x, T & y) { T tmp = x; x = y; y = tmp; }
```

```
  T { provides: Value copy(); }
    { requires: Value copy(); }
```

```
tmp: request a copy from (x) -> accept copy request from (y);

  x: accept copy request from (tmp) -> request a copy from (y);

  y: accept copy request from (x) -> request a copy from (tmp);
```

- Algo: A set of ordering constraints on functions
- *No functional guarantees*
  - *Function implementations?*
  - *Other unconstrained functions not in T?*
  - *Multi-threading?*
- So which property should **`myswap`** satisfy?