# XCD – Modular, Realizable Software Architectures *

Christos Kloukinas and Mert Ozkaya

School of Informatics
City University London
London EC1V 0HB, U.K.
{c.kloukinas,mert.ozkaya.1}@city.ac.uk

**Abstract.** Connector-Centric Design (XCD) is centred around a new formal architectural description language, focusing mainly on *complex connectors*. Inspired by Wright and BIP, XCD aims to cleanly separate in a *modular* manner the high-level *functional*, *interaction*, and *control* system behaviours. This can aid in both increasing the understandability of architectural specifications and the reusability of components and connectors themselves. Through the independent specification of control behaviours, XCD allows designers to experiment more easily with different design decisions early on, without having to modify the functional behaviour specifications (components) or the interaction ones (connectors).

At the same time XCD attempts to ease the architectural specification by following (and extending) a Design-by-Contract approach, which is more familiar to software developers than process algebras like CSP or languages like BIP that are closer to synchronous/hardware specification languages. XCD extends Design-by-Contract *(i)* by separating component contracts into functional and interaction sub-contracts, and *(ii)* by allowing service consumers to specify their own contractual clauses. XCD connector specifications are completely decentralized, foregoing Wright's connector glue, to ensure their realizability by construction.

**Keywords:** Software architecture, Modular specifications, Separation of functional interaction and control behaviours, Design by contract, Connector realizability.

## 1 Introduction

Architectural descriptions of systems are extremely valuable for communicating high-level system design aspects and the different solutions that have been evaluated for meeting system-wide, non-functional properties. The need for components and connectors to be first-class architectural entities has been advocated from the very beginning [15, 30]. However, support for complex connectors is minimal in languages used more widely by practitioners currently, e.g., AADL [13], SysML [6]. These rely mostly on simple interconnection mechanisms like procedure-calls and provide no support for specifying complex connectors, focusing their attention mostly upon components. The end result is that architectures end up more like low-level designs [11].

* This work has been partially supported by the EU project FP7-257367 IoT@Work.

C.S. Păsăreanu and G. Salaün (Eds.): FACS 2012, LNCS 7684, pp. 152–169, 2012.

With minimal support for connectors, components have to incorporate specific interaction protocols, thus reducing their reusability. Worse yet, when component specifications omit to specify explicitly which protocols they have been designed for, we have the problem of "architectural mismatch" [14], i.e., the inability to compose seemingly compatible components, due to the (undocumented) assumptions these make on their interaction with their environment. In quite a few cases designers are supposed to use specific components that act as connectors in order to represent complex connectors. This hinders analysis, as it is not possible to identify automatically which components represent components and which ones represent connectors. It also places a lot of responsibility upon designers for ensuring that the architectural abstraction constraints are respected. It is similar to trying to encode some O-O features by hand in C – possible but very difficult to get (and keep…) correct. In our view the main value of software architectures is to enable early formal system analysis and not to be used for code generation alone. As such, an ADL needs to cleanly represent the various entities, in order to aid the automation of architectural analysis.

The Connector-Centric Design (XCD) approach attempts to apply Wirth's equation *"Algorithms + Data Structures = Programs"* [35] at an architectural level. We advocate that *"Connectors + Components = Systems"*, with connectors being essentially decentralized algorithms and components the equivalent to data structures [22]. *This means that the main active elements in an architecture are its connectors, not its components.* XCD focuses on improving the modularity of architectural specifications, so to aid their development, their formal analysis, and the experimentation with different design solutions. Complex connectors are at the very centre of XCD, since it is them that are responsible for meeting system-wide, non-functional requirements that no component can meet, such as reliability, performance, etc. In the following we shortly introduce the reasons behind the three orthogonal goals of XCD, namely support for complex connectors, support for external control strategies, and specification through a Design by Contract approach.

**Complex Connectors for Architectural Analysis.** Herein we use an example from electrical engineering to demonstrate the importance of complex connectors for analyzing system architectures. Let us consider $k$ concrete electrical resistors, $r_1, \cdots, r_k$, i.e., the system components. When using a sequential connector ($\rightarrow$), the overall resistance is computed as $R^{\rightarrow}(N, \{R_i\}_{i=1}^{N}) = \sum_{i=1}^{N} R_i$, where $N, R_i$ are variables ($R_i$ correspond to connector roles), to be assigned eventually some concrete values $k, r_j$. If using a parallel connector ($\|$) instead, it is computed as $R^{\|}(N, \{R_i\}_{i=1}^{N}) = 1/\sum_{i=1}^{N} 1/R_i$. So the interaction protocol (connector) used is the one that gives us the formula we need to use to analyze it – if it does not do so, then we are probably using the wrong connector abstraction. The components ($r_j$) are simply providing some numerical values to use in the formula, while the system configuration tells us which specific value ($k$, $r_j$) we should assign to each variable ($N$, $R_i$) of the connector-derived formula. By simply enumerating the wires between resistors, as AADL and SysML do, we miss the forest for the trees. Analysis becomes difficult and architectural errors can go undetected until later development phases. Indeed, we are essentially forced to reverse-engineer the architect's intention in order to analyze our system – after all, the architect did not select the specific wire connections by chance but because they form a specific complex connector. The current situation is similarly to coding with labels and go-to statements and expecting our compilers to identify the higher-level looping and procedural constructs within our code so as to analyse and optimize it.

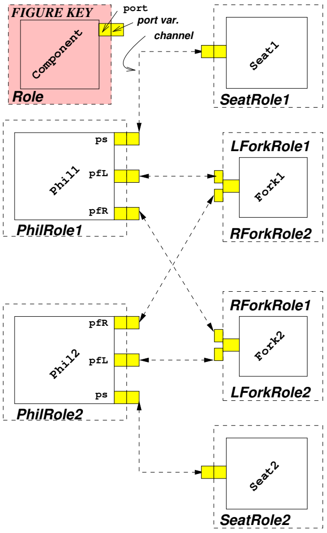
**Connector Role Strategies for Control/Design Decisions.** A cleaner separation of functional and interaction behaviour aids in increasing the reusability of both components and connectors. However, one can go even further, e.g., as in BIP [8], and attempt to separate the control behaviour as well. XCD supports this through modular *connector role strategies*, which are specified *externally* to connectors, and so can be replaced and modified easily. These are used to specify *different design solutions* for various issues that basic role specifications do not address (on purpose) so as to be as reusable as possible. In fact, such role strategies are already being used in good designs implicitly. Consider a simple call in C: `foo(i, ++i)`, where `i=1`. According to the C language specification this call is undefined since the second parameter expression (`++i`) may potentially change the value of the first one (`i`). So we can obtain either `foo(1,2)` or `foo(2,2)`. The C language specification does not specify a specific order for evaluating parameters either in the caller or the callee role, instead *under-specifying* the procedure-call connector specification on purpose. If compilers have multiple cores at their disposal they are allowed to evaluate parameters in parallel, instead of having to evaluate each one in a specific sequential order. The C language specification allows compilers to apply different evaluation strategies on the caller role by delaying this design decision until the optimal choice can be made, based on the call context and the implementation costs of the available strategies.

**Design by Contract.** JML [9] seems to be gaining popularity among developers, as they use it for "test-driven development". XCD attempts to follow this trend so as to maximize adoption by practitioners. Thus, it departs from Wright's [1] use of a process algebra (Hoare's CSP [18]) and follows a Design by Contract (DbC) [28] approach like JML instead, specifying systems through simple *pairs of method pre-/post-conditions*, based upon Hoare's logic [17]. In fact, XCD extends DbC in two ways. First, it *separates* the component *functional behaviour* from its *minimal interaction requirements*. Second, it allows *service consumers* to specify their own *contractual clauses*.

### 1.1 Running Example – The Dining Philosophers

We present XCD through the classic *Dining Philosophers* problem, since one needs a complex enough system to demonstrate the need for the different aspects of the approach. This system can be designed with either decentralized or centralized control (i.e., a butler), and for each of these general architectural solutions, there are different specific design solutions for controlling the system in particular ways (e.g., for deadlock-freedom). In the dining philosophers problem a set of *n* philosophers occasionally sit on seats at a round table, sharing a fork at their right and left. Each philosopher needs both forks to be able to eat but if all philosophers get one fork then there is a deadlock, since no philosophers put down a fork until they have finished eating.

We show how designers can specify the system architecture and experiment with different control policies, without changing the specifications of either the connectors



Philosopher ports are named (ps, pfL, pfR) as they are mirrored.

**Fig. 1.** Dining Philosophers System configuration for decentralized control

```
Component                     Connector
 Data, Predicates              Roles
 Socket|Plug Ports              Data, Predicates
  Methods                       Socket|Plug Port Variables
   Interaction Constraints       Methods
   Functional Constraints         Interaction Constraints
                               Channels (connecting role port variables)
Role Strategy
 Data                         Configuration
 Predicates                    Component Instances
 Socket|Plug Port Variables    Connector Instances
  Methods                      Role Strategy Instances
   Interaction Constraints     Component/Role/Strategy Instance Bindings
```

**Fig. 2.** XCD language top-level structure

or components. Fig. 1 shows a possible configuration of the dining philosophers case study, for two philosophers. Some of the elements there (ports, port variables, etc.) are explained later, though Fig. 2 presents a quick summary of the main language structure. All constraints in XCD's elements in Fig. 2 are expressed as pre/post-conditions. Strategies may introduce their own data, predicates, and constraints but can refer only to methods of port variables defined in a role. Designers are expected to start an architectural description by the components, then derive connectors for them, and finally specify appropriate strategies. Connector roles are defined over component interface fragments, as is done in generic programming [10,29]. For example, C++'s STL defines algorithm

```
void sit(ID caller)   throws (NullIDEX);
void arise(ID caller) throws (NullIDEX, WrongCallerEX, InteractionEX);
```

**Fig. 3.** The `sit/arise` ($i_{SA}$) Seat interface

$$<D = \left[\texttt{ID h} := \bot\right], preds = \begin{bmatrix} \mathit{Occupied} = (\texttt{h} \neq \bot), & \mathit{NullCaller}(\texttt{c}) = (\texttt{c} = \bot), \\ \mathit{CallerIsHolder}(\texttt{c}) = (\texttt{c} = \texttt{h}), & \mathit{HolderIsCaller}(\texttt{c}) = (\texttt{h}' = \texttt{c}), \\ \mathit{NoHolder} = (\texttt{h}' = \bot), & \end{bmatrix},$$

$$P^s = \{p_s^{i_{SA}}\}, P^p = \emptyset, \phi, \chi >$$

**(a)** Seat top-level specification

$$\begin{bmatrix} s_1^{\phi} = \left(p_s, \texttt{sit}(\texttt{c}), \neg \mathit{NullCaller}(\texttt{c}), \mathit{HolderIsCaller}(\texttt{c})\right) \\ s_2^{\phi} = \left(p_s, \texttt{sit}(\texttt{c}), \mathit{NullCaller}(\texttt{c}), \texttt{NullIDEX}\right) \end{bmatrix}$$

$$\begin{bmatrix} a_1^{\phi} = \left(p_s, \texttt{arise}(\texttt{c}), \neg \mathit{NullCaller}(\texttt{c}) \wedge \mathit{CallerIsHolder}(\texttt{c}), \mathit{NoHolder}\right) \\ a_2^{\phi} = \left(p_s, \texttt{arise}(\texttt{c}), \mathit{NullCaller}(\texttt{c}), \texttt{NullIDEX}\right) \\ a_3^{\phi} = \left(p_s, \texttt{arise}(\texttt{c}), \neg \mathit{CallerIsHolder}(\texttt{c}), \texttt{WrongCallerEX}\right) \end{bmatrix}$$

**(b)** Seat functional constraints ($\phi$)

$$\begin{bmatrix} s_1^{\chi} = \begin{pmatrix} p_s, \texttt{sit}(\texttt{c}), & \\ \textbf{when}(\neg \mathit{Occupied}), \mathbf{T} & \end{pmatrix} \end{bmatrix} \quad \begin{bmatrix} a_1^{\chi} = \left(p_s, \texttt{arise}(\texttt{c}), \mathit{Occupied}, \mathbf{T}\right) \\ a_2^{\chi} = \left(p_s, \texttt{arise}(\texttt{c}), \neg \mathit{Occupied}, \texttt{InteractionEX}\right) \end{bmatrix}$$

**(c)** Seat interaction constraints ($\chi$)

**Fig. 4.** Seat component specification

`sort` on a sequence of elements of type `T`, using `T`'s less-than, assignment, and copy constructor operations.

### 1.2 Paper Structure

We consider first component specifications in XCD, concentrating then on connectors – their specification in a decentralized manner that facilitates their implementation and analysis, and the fundamental properties that a connector should provide. We then consider role strategies for expressing control and other design decisions, and present an evaluation of the approach before discussing related work and concluding.

## 2 XCD Components

Fig. 3 shows the $i_{SA}$ interface implemented by Seat components; the `get/put` one implemented by Forks ($i_{GP}$) is exactly the same. Method `sit` throws a `NullIDEX` exception, while `arise` also throws `WrongCallerEX` when the Seat is occupied by someone that is not the caller. However, `arise` throws yet another exception – the enigmatic

$$pre(s_1^\chi) \rightarrow \bigwedge \begin{pmatrix} pre(s_1^\phi) \rightarrow post(s_1^\phi) \\ pre(s_2^\phi) \rightarrow post(s_2^\phi) \end{pmatrix}, pre(a_1^\chi) \rightarrow \bigwedge \begin{pmatrix} pre(a_1^\phi) \rightarrow post(a_1^\phi) \\ pre(a_2^\phi) \rightarrow post(a_2^\phi) \\ pre(a_3^\phi) \rightarrow post(a_3^\phi) \end{pmatrix}, pre(a_2^\chi) \rightarrow post(a_2^\chi)$$

**Fig. 5.** Constraint composition semantics

`InteractionEX`. Components "throw" this special exception when their *minimal interaction constraints* (rather than functional ones) have been violated, to denote subsequent chaotic behaviour. If one opens the door of a washing machine while it is washing, subsequent behaviours include everything, even electrocution.

### 2.1 Extending DbC – Different Contract Types

Fig. 4a shows the Seat component specification. It defines its *data* variable set ($D$) and some helper *predicates* (*preds*). Then it defines two sets of *ports*, ($P^s$, $P^p$), for the "*socket*" and "*plug*" ports (empty set) respectively, i.e., the ones *providing* some interface and these *using* some interface – what in CORBA are *facets* and *receptacles*. Finally, it defines *functional* ($\phi$) and *interaction* ($\chi$) constraints, as in Fig. 4b and 4c.

All constraints use the syntax (*port-expr.*, *method*, *pre-condition*, *post-condition*). They are grouped ([]) by the (*port-expr.*, *method*) pair they apply to. They are labelled here for easy reference as $(s|a)^{(\phi|\chi)}$ – for `sit`/`arise` ($s|a$) and for functional/interaction ($\phi|\chi$). So in $s_1^\phi$, $p_s$'s `sit` pre-condition is $\neg$*NullCaller*(`c`) and *HolderIsCaller*(`c`) its post-condition, where `c` is `sit`'s parameter. This is a JML "normal_behaviour", unlike $s_2^\phi$ that throws a `NullIDEX` if the pre-condition *NullCaller*(`c`) is true. Constraints $a_1^\phi$, $a_2^\phi$ are similar ones for `arise`, while $a_3^\phi$ covers the case when the pre-condition is $\neg$*CallerIsHolder*(`c`). In that case, the post-condition throws a `WrongCallerEX`.

This last constraint $a_3^\phi$ introduces the difference between functional and (minimal) interaction constraints. Method `arise` accepts calls where the caller is not the current seat holder and throws an exception, while `sit` does not specify anything about this. According to $s_1^\phi$ it seems it simply replaces Seat's holder with the caller. However, this is captured in Fig. 4c, through Seat's *minimal interaction constraints*. Constraint $s_1^\chi$ asks that `sit` be delayed until *Occupied* is false. This is expressed using the "**when**" keyword as in JML's extension for multi-threaded programming [33], though in XCD functional constraints are not allowed to use it. To relate it to JML, one can think of it as a "normal" interaction behaviour, describing a method's acceptable concurrent behaviours. For all "normal" interaction constraints of components, the post-condition is always **T**. Fig. 4c also specifies the minimal interaction constraints of `arise`. Constraint $a_1^\chi$ states that calling `arise` on an occupied Seat is acceptable. Constraint $a_2^\chi$, however, states that calling `arise` on an unoccupied Seat, results in an `InteractionEX` exception (which functional constraints cannot use). This is a situation that Seat does not know how to deal with, like calling a method on a component without having initialized it first. An `InteractionEX` exception leads to *undefined/chaotic* component behaviour.

Interaction constraints *take precedence* over functional ones and if both can throw an exception then the exception thrown is `InteractionEX`. With $pre(\phi)$ and $post(\phi)$

standing for the pre-condition and the post-condition respectively of a constraint $\phi$, the real specification of the Seat constraints is shown in Fig. 5. As highlighted there for $a_2^\chi$, when an interaction exception's precondition is true, then the functional constraints are ignored. Otherwise, when the pre-condition of a normal interaction constraint is satisfied, the functional constraints should also be satisfied.

If one specified contracts in the usual JML manner, they would need $F \times I$ cases in the worst case, combining $F$ functional and $I$ interaction constraints, e.g., for `arise`:
**Case 1:** $pre(a_1^\chi) \wedge pre(a_1^\phi) \rightarrow post(a_1^\phi)$ **Case 3:** $pre(a_1^\chi) \wedge pre(a_3^\phi) \rightarrow post(a_3^\phi)$
**Case 2:** $pre(a_1^\chi) \wedge pre(a_2^\phi) \rightarrow post(a_2^\phi)$ **Case 4:** $pre(a_2^\chi) \rightarrow post(a_2^\chi)$
Repeating "$pre(a_1^\chi)$" each time makes specifications more difficult to read than they need be and much easier to get wrong. The introduction of the (minimal) interaction constraints *imposes* a much cleaner and modular manner (and guides the specification of connectors as discussed later).

### 2.2 Extending DbC – Service Consumer Contracts

In DbC service providers specify pre-/post-conditions for their methods but service consumers cannot express their own contractual clauses on them. Indeed, most languages do not allow consumers to even declare the services/interfaces they use. However, in component models like CORBA one declares both the services it provides (our sockets) and those it consumes (our plugs). Here we *extend DbC further*, so that we can *specify contracts for consumed services* as well. This is done for the Philosopher in Fig. 6a. Philosopher has a Boolean variable *wte* ("want to eat"), and three more (*hs*, *hl*, *hr*) to state whether it has a Seat, a left and a right Fork respectively. These change their values according to its functional constraints in Fig. 6b, *which apply when a method does not throw an exception* – that is why we call them "normal". On exceptions components do not update their data. Keyword `self` denotes the `ID` of the component instance.

The Philosopher interaction constraints in Fig. 6c state *when services may be requested* from others. These constraints *specify no resource acquisition/release order*. Philosopher is free to acquire a Seat after both Forks or in between them. In fact, it can even acquire or release a resource multiple times. The constraints state that when it wants to eat it will need to acquire all three resources, without releasing any of them. When it does not want to eat, it will release all three resources (again in some unspecified order), without attempting to re-acquire any of them until all of them have been released. These constraints were added so that the system can deadlock. Otherwise, Philosopher can always release the resources it holds when those it needs are not available. It is exactly for this that we have introduced functional and interaction constraints to plug ports (required interfaces). They are needed to express the constraints under which the service providers must operate, i.e., the service's *"environment model"*.

### 2.3 Component Structure and Its Translation to FSP

XCD components have six components – Data, Predicates, Socket_Ports, Plug_Ports, Functional_Constraints, and Interaction_Constraints. We encode these into the FSP process algebra [26] by first creating a process for the Data component of each component C, that acts as the XCD component's internal memory.

$$<D = \left\{\begin{array}{l}\texttt{Bool wte} := \mathbf{T}, \texttt{Bool hs} := \mathbf{F}, \\ \texttt{Bool hl} := \mathbf{F}, \texttt{Bool hr} := \mathbf{F}\end{array}\right\}, preds = \begin{bmatrix} Eat = (wte \wedge hs' \wedge hl' \wedge lr'), \\ Think = \neg(wte \vee hs' \vee hl' \vee lr') \end{bmatrix},$$

$$P^s = \emptyset, P^p = \{p_{p\_s}^{i_{SA}}, p_{p\_fR}^{i_{GP}}, p_{p\_fL}^{i_{GP}}\}, \phi, \chi >$$

**(a)** Philosopher top-level specification

$$\big[(p_{p\_s}, \texttt{sit}(\texttt{self}), \mathbf{T}, hs' = \mathbf{T} \wedge wte' = \neg Eat)\big] \quad \big[(p_{p\_s}, \texttt{arise}(\texttt{self}), \mathbf{T}, hs' = \mathbf{F} \wedge wte' = Think)\big]$$

$$\big[(p_{p\_fL}, \texttt{get}(\texttt{self}), \mathbf{T}, hl' = \mathbf{T} \wedge wte' = \neg Eat)\big] \quad \big[(p_{p\_fL}, \texttt{put}(\texttt{self}), \mathbf{T}, hl' = \mathbf{F} \wedge wte' = Think)\big]$$

$$\big[(p_{p\_fR}, \texttt{get}(\texttt{self}), \mathbf{T}, hr' = \mathbf{T} \wedge wte' = \neg Eat)\big] \quad \big[(p_{p\_fR}, \texttt{put}(\texttt{self}), \mathbf{T}, hr' = \mathbf{F} \wedge wte' = Think)\big]$$

**(b)** *"Normal"* functional constraints ($\phi$)

$$[(p_{p\_s}, \texttt{sit}(\texttt{self}), \mathbf{when}(wte), \mathbf{T})] \quad [(p_{p\_s}, \texttt{arise}(\texttt{self}), \mathbf{when}(\neg wte), \mathbf{T})]$$

$$[(p_{p\_fL}, \texttt{get}(\texttt{self}), \mathbf{when}(wte), \mathbf{T})] \quad [(p_{p\_fL}, \texttt{put}(\texttt{self}), \mathbf{when}(\neg wte), \mathbf{T})]$$

$$[(p_{p\_fR}, \texttt{get}(\texttt{self}), \mathbf{when}(wte), \mathbf{T})] \quad [(p_{p\_fR}, \texttt{put}(\texttt{self}), \mathbf{when}(\neg wte), \mathbf{T})]$$

**(c)** Philosopher interaction constraints ($\chi$)

**Fig. 6.** Philosopher component specification

```
C_Mem = D([InitialValue(V)])*,
D([Name(V):Type(V)])* = read([Name(V)])* -> D([Name(V)])*
 | write([Name(V)_n:Type(V)])* -> D([Name(V)_n:Type(V)])* .
```

That is, a state of the memory is indexed for each Data variable (V) and the initial state is selected according to the initializations in the Data component. `Name/Type(V)` produces the name, respectively type, of the variable and the star operator means zero or more occurrences of its operand. Our translator currently supports Boolean and bounded integer variables. For Philosopher, this produces:

```
Philosopher_Mem = D[True][False][False][False],
D[wte:Bool][hs:Bool][hl:Bool][hr:Bool]
 = ( read[wte][hs][hl][hr] -> D[wte][hs][hl][hr]
   | write[wten:Bool][hsn:Bool][hln:Bool][hrn:Bool]
      -> D[wten][hsn][hln][hrn] ) .
```

Then each port P of a component C is encoded as an FSP process that locks the memory, reads its current state, and evaluates the interaction constraints of the port's methods.

```
C_P(ID=1) = Port,
Port = (lock -> read([Name(V):Type(V)])* -> P([Name(V)])*),
P([Name(V):Type(V)])* =
 {forall(m : Method, i : Interaction_Constraint)
 when(pre(interaction(m, i))) m([Name(arg):Type(arg)])*
  -> internal_m([Name(arg)])* ([Name(V)])*
  -> internal_m([Name(arg)])* ([Name(V)_n:Type(V)])*
                              [r:RES][e:EX]
  -> ( when (NoEXCEPTION != e) unlock
      -> RES_m([Name(arg)])*[r][e] ([Name(V)])*
                                   ([Name(V)_n])*
    | when (NoEXCEPTION == e) write([Name(V)_n])* -> unlock
      -> RES_m([Name(arg)])*[r][e] ([Name(V)])*
                                   ([Name(V)_n])* )
...
 } // end of forall(m, i)
 | when({forall(m,i) !pre(interaction(m, i))}) unlock->Port,
```

Here, when a method m's i$^{\text{th}}$ interaction precondition is satisfied, a call to it is accepted (m([Name(arg):Type(arg)])*) and it is passed to another process through the internal_m action. The other process, which can be seen as m's "implementation", responds by an action internal_m which has the same values for the arguments and the new values for the Data variables, as well as the result (r) and exception (e) returned. When there is an exception the memory is unlocked and we pass control to sub-process "RES_m". When there is no exception we update the memory, unlock it and then pass control to RES_m. Here, RES_m is a sub-process of P (one per method m) responsible for checking m's functional constraints, using m's arguments (arg), return type (r), exception thrown (e), and values of the Data variables (V). Predicates are expanded wherever they are used.

Processes implementing a method m (those controlling the "internal_m" actions), follow this pattern, where C is the component name and P its port:

```
C_P_m(ID=1) = (
 internal_m([Name(arg)]:Type(arg))* ([Name(V):Type(V)])*
 -> ({forall(f : Functional_Constraint)
      when (pre(functional(m,f)))
      internal_m([Name(arg)])* ([V'])* [r'][e'] -> C_P_m }
    | when !(CP2) incomplete_pre_conditions -> ERROR).
// where CP2 is {∨_f pre(functional(m,f))} -- see eq. (2) below
```

### 2.4 Testing Architectural Components

Following Fig. 5's constraint semantics, one needs to check that $(\mathbf{CP_1})$ the interaction pre-conditions are *complete*; and that *whenever the normal interaction pre-conditions are satisfied* that $(\mathbf{CP_2})$ the functional pre-conditions are *complete*; and $(\mathbf{CP_3})$ the functional constraints are *consistent*.

$$\mathbf{CP_1} = \forall m. \bigvee_n pre(m_n^{\chi}) \tag{1}$$

$$\mathbf{CP_2} = \forall m. \bigwedge_k \left( pre(m_k^{\chi}) \rightarrow \bigvee_n pre(m_n^{\phi}) \right) \tag{2}$$

$$\mathbf{CP_3} = \forall m. \bigwedge_k \left( pre(m_k^{\chi}) \rightarrow \bigwedge_n \left[ pre(m_n^{\phi}) \rightarrow post(m_n^{\phi}) \right] \right) \tag{3}$$

In equation (1) $n$ ranges over both the normal and exceptional interaction constraints and the predicate **when**$(\phi)$ always evaluates to **T**. So for Seat's `sit`, we need to verify that $pre(s_1^{\chi})$ holds. Being a **when** predicate, this is the case. For Seat's `arise` we can also verify that $(pre(a_1^{\chi}) \vee pre(a_2^{\chi})) = (Occupied \vee \neg Occupied)$ holds. In equation (2) $k$ ranges over the *normal* interaction constraints of method $m$ and $n$ ranges over all its functional constraints. Both here and in equation (3) the predicate **when** is evaluated as the *identity* function, i.e., $\textbf{when}(\phi) = \phi$. This is because we want to evaluate the completeness of the functional pre-conditions only when the method is eventually executed, in which case the **when** condition should hold.

The $\mathbf{CP_3}$ condition is effectively checked in our FSP models through the RES_m sub-processes of ports mentioned in the previous section:

```
RES_m([Name(arg):Type(arg)])*
      [r:RES][e:EX] ([Name(V):Type(V)])* =
Let CP3={∧f !pre(functional(m,f)) || post(functional(m,f))}
  when (  CP3) m_ret([Name(arg)])* [r][e] -> Port
| when (! CP3) inconsistent_normal_conditions -> ERROR
```

## 3 XCD Connectors

We can now consider connectors, as Fork is similar to Seat. If we opt for something like procedure-call, event-bus, etc. then we are specifying our system at a very low level. The extra details obfuscate the design, making it difficult to identify the high-level interaction protocols, thanks to which the system achieves its non-functional requirements. This is why XCD focuses instead on *complex connectors*. These connectors consist of a set of *roles*, each one with a set of *port variables*. Role port variables are assumed by some component ports, as specified by the architectural configuration.

**Glue-Less Connectors.** XCD connectors differ from those of Wright [1], since XCD employs no "glue" element for coordinating role behaviours. The glue is problematic for a number of reasons. First, the *glue is a choreography*, so one needs to realize it as a set of individual services (i.e., role implementations) composed in parallel. But [2, 3] have shown that the *choreography realization* problem is *undecidable* in general. Deciding realizability in certain cases is indeed possible, e.g., [7], and in some cases unrealizable choreographies can be repaired by extending the recipient set of messages [25]. However, this is the least of the problems introduced by glues. More importantly, if we need to consider multiple instances of some role, then we need to manually specify in the glue all the acceptable composed behaviours of these instances. For example, when considering a market system with one consumer and two merchants in [12], the glue describes all possible interactions of the three roles. This *does not scale* – it is impractical to specify a glue with five or more merchants and quasi-impossible to do so for $N$ merchants. Finally, the glue hinders the architectural analysis for further non-functional requirements, such as reliability, performance, real-time behaviour, etc. It introduces an *artificial centralization point* in the connector, even if the protocol represented by the connector does not have such a centralization point, e.g., the procedure-call. This makes analysis more difficult, since now one has to consider the real centralization points (e.g., for reliability analysis), while ignoring the fictitious ones (the glue elements of the various connectors). It also makes the modelling more *difficult to validate*. For example, in [12] the authors perform a probabilistic analysis of a market system, assigning a rate

$$< R = \{r_p, r_s, r_{fL}, r_{fR}\}, Chan >$$

**(a)** Connector top-level specification

$$r_p = \begin{bmatrix} D = \begin{Bmatrix} \texttt{Bool}\, sitting := \mathbf{F}, \\ \texttt{Bool}\, gotLF := \mathbf{F}, \\ \texttt{Bool}\, gotRF := \mathbf{F} \end{Bmatrix}, \\ preds = \emptyset, P_v^p = \begin{Bmatrix} pv_{p\_s}^{i_{SA}}, pv_{p\_fL}^{i_{GP}}, \\ pv_{p\_fR}^{i_{GP}} \end{Bmatrix}, \\ P_v^s = \emptyset, \chi_p \end{bmatrix}$$

$$r_s = \left[ P_v^s = \left\{ pv_s^{i_{SA}} \right\} \right]$$

$$r_{fL} = \left[ P_v^s = \left\{ pv_{fL}^{i_{GP}} \right\} \right]$$

$$r_{fR} = \left[ P_v^s = \left\{ pv_{fR}^{i_{GP}} \right\} \right]$$

**(b)** Role $(R)$ definitions

$$\left[ \left( pv_{p\_s}, \texttt{sit(self)}, \mathbf{when}(\neg sitting), sitting' = \mathbf{T} \right) \right]$$

$$\left[ \left( pv_{p\_s}, \texttt{arise(self)}, \mathbf{when}(sitting), sitting' = \mathbf{F} \right) \right]$$

$$\left[ \left( pv_{p\_fL}, \texttt{get(self)}, \mathbf{when}(\neg gotLF), gotLF' = \mathbf{T} \right) \right]$$

$$\left[ \left( pv_{p\_fL}, \texttt{put(self)}, \mathbf{when}(gotLF), gotLF' = \mathbf{F} \right) \right]$$

$$\left[ \left( pv_{p\_fR}, \texttt{get(self)}, \mathbf{when}(\neg gotRF), gotRF' = \mathbf{T} \right) \right]$$

$$\left[ \left( pv_{p\_fR}, \texttt{put(self)}, \mathbf{when}(gotRF), gotRF' = \mathbf{F} \right) \right]$$

**(c)** Philosopher role constraints $(\chi_p)$

$$c_{p,s}^{\rightleftharpoons} = (pv_{p\_s}, pv_s), c_{p,fl}^{\rightleftharpoons} = (pv_{p\_fL}, pv_{fL}),$$
$$c_{p,fr}^{\rightleftharpoons} = (pv_{p\_fR}, pv_{fR})$$

**(d)** Channels $(Chan)$ connecting port-variables

**Fig. 7.** Dining philosophers decentralized control connector

$R_1$ to all transitions between the consumer role and the glue and a rate $R_2$ to all transitions among the glue and the merchant roles. However, transitions between the consumer and the glue represent in reality requests from the consumer to the merchants, as well as responses from the merchants to the consumer. The transitions among the glue and the merchants also represent the same requests and responses. We fail to see how these rate assignments can be justified – in our view, the glue complicates the situation so much that it is very easy to produce models that are difficult to understand, and, thus, difficult to ensure that they represent the real system faithfully.

**Wrapper-Like Connectors.** In [1], a component should implement the roles it assumes, $\mathscr{L}(Comp) \subseteq \mathscr{L}(Role)$, i.e., have the same set of behaviours as the role or a subset of that. This seems too constraining and limiting component reusability. Instead, XCD components focus on implementing just the minimum interaction constraints that they need to operate correctly. The roles they assume act as a sort of *wrapper*, controlling their behaviour so that it meets the expected role behaviours.

Another way of looking at it is to consider components as machines that (modulo their constraints) execute the script (constraints) specified by the connector roles they assume, just like human actors do.

### 3.1 Decentralized Control Connector

Fig. 7 shows the specification of a complex Decentralized Control connector for the dining philosophers. The connector defines a set of roles and interaction channels (Fig. 7a). The specifications of the roles are shown in Fig. 7b. Each of them has five constituent parts: a set of *role data variables* $(D)$, a set of *predicates* $(preds)$, a set of *plug port variables* $(P_v^p)$, a set of *socket port variables* $(P_v^s)$, and a set of *interaction constraints* $(\chi)$. Roles $r_s$, $r_{fL}$, and $r_{fR}$, have socket port variables only (rest omitted for brevity). Role $r_p$

uses variables to keep track of the state of resources and to control it through its constraints in Fig. 7c so that it only acquires resources when it does not hold them already and releases them when it does hold them. Constraints modify role variables only when the respective methods do not raise exceptions. Channels in Fig. 7d state which role port variables are linked to each other – all the channels we use are *rendez-vous* ones.

This connector does not describe the full system configuration. If there are $n$ instances of the Philosopher, Seat, and Fork components in the system then there should be $n$ instances of the Decentralized connector as well, since a single connector instance can only connect one Philosopher, with one Seat and two Forks.

### 3.2 FSP Encoding of XCD Connector Roles

Encoding connector role port variables is similar to encoding component ports. The only difference is that since roles do not have functional constraints, a request for a method "m([Name(arg):Type(arg)])*" is immediately followed by a response through its corresponding "m_ret" action (performed by some component's RES_m sub-process).

### 3.3 Fundamental Connector Properties

There is *only one fundamental property* for a connector to be complete: $(\mathbf{XP_1})$ *interaction exception-freedom.* $(\mathbf{XP_1})$ is a connector-level property. It requires that component socket ports never throw an interaction exception, no matter how the component plug ports behave. This can be checked by composing the connector with the corresponding components that assume its roles, *while setting all interaction pre-conditions of component plug ports to* **T** (i.e., those in Fig. 6c). Doing so allows us to explore all possible interaction patterns that the connector roles allow for the components and verify that interaction exceptions have been rendered impossible by it. However, a connector can be left incomplete on purpose and be completed later using role strategies.

It should be noted that a connector is not necessarily deadlock-free, not even at a local, role level. That is, a role may introduce such constraints that it renders its component's actions impossible after some point. XCD permits this as the designer may need it to essentially remove a component from a protocol at some point. If one wants to check that this is not the case, then they need to check local deadlock-freedom $(\mathbf{XP_2})$: $\mathscr{L}(Comp \parallel Role)@\Sigma^{Role} \subseteq \mathscr{L}(Role)$, where @ projects a language on an alphabet.

Even when a connector is complete and locally deadlock-free it is not necessarily deadlock-free as a whole. Nevertheless we do not view this as being problematic because we believe that connector-level deadlock-freedom is best met through external role strategies as discussed in section 4.

## 4 Role Strategies – Control/Design Decisions

XCD *advocates the underspecification of connectors* – additional interaction properties are to be imposed through *modular role strategies* [22]. These can enforce an action order, e.g., that Seat is acquired before the Forks, or render the system deadlock-free. Deadlock-freedom can usually be achieved through different techniques. Instead of

$$\big[\big(pv_{p\_fL}, \texttt{get}(c), \textbf{when}(\mathit{sitting}), \mathbf{T}\big)\big]$$
$$\big[\big(pv_{p\_fR}, \texttt{get}(c), \textbf{when}(\mathit{sitting}), \mathbf{T}\big)\big]$$

**(a)** Resource acquisition order

$$\big[\big(pv_{p\_s}, \texttt{arise}(c), \textbf{when}\neg(\mathit{gotLF} \vee \mathit{gotRF}), \mathbf{T}\big)\big]$$

**(b)** Resource release order

$$\big[\big(pv_{p\_fL}, \texttt{get}(c), \textbf{when}(\mathit{gotRF} \vee c\%2 = 0), \mathbf{T}\big)\big]$$
$$\big[\big(pv_{p\_fR}, \texttt{get}(c), \textbf{when}(\mathit{gotLF} \vee c\%2 \neq 0), \mathbf{T}\big)\big]$$

**(c)** Deadlock-avoidance by asymmetry – for even/odd $c$

$$\big[\big(pv_{p\_fL}, \texttt{get}(c), \textbf{when}(\mathit{gotRF} \vee (r_{fL}.\texttt{ID} < r_{fR}.\texttt{ID})), \mathbf{T}\big)\big]$$
$$\big[\big(pv_{p\_fR}, \texttt{get}(c), \textbf{when}(\mathit{gotLF} \vee \neg(r_{fL}.\texttt{ID} < r_{fR}.\texttt{ID})), \mathbf{T}\big)\big]$$

**(d)** Deadlock-avoidance by resource order – the fork with the least ID has priority

**Fig. 8.** Philosopher role strategies (their constraints)

hard-coding one in the connector, XCD allows designers to re-use the same connector specification and experiment with different strategies for it in a modular fashion.

Fig. 8 shows examples of such strategies for the Philosopher role. The strategy in Fig. 8a forces Seat to be acquired before the Forks, while that of Fig. 8b forces Forks to be released first. Then the asymmetry strategy in Fig. 8c avoids deadlocks by picking a different Fork when the ID of the caller $c$ is odd or even. The strategy in Fig. 8d also avoids deadlocks but does so by always acquiring the Fork with the smallest ID first.

Strategies are encoded in FSP like roles are. Finally, configurations are encoded by a series of action prefixing, renaming, etc., that are too tortuous to describe in detail.

## 5 Evaluating XCD's Modular Specifications

We have encoded (first manually, then automatically) these architectural specifications in the FSP process algebra [26] and have verified them automatically. Our goal was to establish that our architectural specifications can be verified automatically indeed and to obtain some early results on the usefulness of modular specifications. In particular we wanted to evaluate the usefulness of control strategies and how these could aid designers when developing an architecture. In total, we considered 12 different configurations for the decentralized system, shown in Fig. 1 for two philosophers, using different combinations of strategies. In all these cases our models remained the same, with the only difference being the enabling/disabling of strategies. This cannot be stressed enough – without such a modular specification it would have been extremely difficult to encode in FSP the different models of connector/strategy combinations or, even worse, the different models of connector/strategy/component combinations if we were to use AADL-like simple connectors. Not having a compiler initially (a prototype one is available now) had forced us to increase the modularity of our language as much as possible. This modularity maximizes architectural exploration in practice – one can start with minimal component and connector specifications and test multiple strategies without having to modify any specifications.

The different role strategies defined in Fig. 8 allow designers to easily experiment with controlling their system and evaluating different design decisions early on. XCD

**Table 1.** Different decentralized control strategy combinations

**(a)** 2 Philosophers

| **Strategies** | **States** | **Red. (%)** | **Trans.** | **Red. (%)** | **Dead-lock** |
|---|---|---|---|---|---|
| No strategies | 505 | 0.00 | 1104 | 0.00 | Yes |
| **Acq**(uisition) | 303 | 40.00 | 628 | 43.12 | Yes |
| **Rel**(ease) | 345 | 31.68 | 732 | 33.70 | Yes |
| **As**(ymmetry) | 335 | 33.66 | 708 | 35.87 | No |
| Acq./Rel. | 179 | 64.55 | 352 | 68.12 | Yes |
| Acq./As. | 245 | 51.49 | 504 | 54.35 | No |
| Rel./As. | 205 | 59.41 | 412 | 62.68 | No |
| Acq./Rel./As. | 133 | 73.66 | 256 | 76.81 | No |
| | | | | | |
| Res. Order (**RO**) | 335 | 33.66 | 708 | 35.87 | No |
| Acq./RO | 245 | 51.49 | 504 | 54.35 | No |
| Rel./RO | 205 | 59.41 | 412 | 62.68 | No |
| Acq./Rel./RO | 133 | 73.66 | 256 | 76.81 | No |

**(b)** 3 Philosophers

| **Strategies** | **States** | **Red. (%)** | **Trans.** | **Red. (%)** | **Dead-lock** |
|---|---|---|---|---|---|
| No strategies | 12750 | 0.00 | 42060 | 0.00 | Yes |
| **Acq**(uisition) | 6381 | 49.95 | 20178 | 52.03 | Yes |
| **Rel**(ease) | 6615 | 48.12 | 21030 | 50.00 | Yes |
| **As**(ymmetry) | 7550 | 40.78 | 24320 | 42.18 | No |
| Acq./Rel. | 2532 | 80.14 | 7452 | 82.28 | Yes |
| Acq./As. | 4850 | 61.96 | 15278 | 63.68 | No |
| Rel./As. | 3260 | 74.43 | 9892 | 76.48 | No |
| Acq./Rel./As. | 1667 | 86.93 | 4804 | 88.58 | No |
| | | | | | |
| Res. Order (**RO**) | 7550 | 40.78 | 24320 | 42.18 | No |
| Acq./RO | 4850 | 61.96 | 15278 | 63.68 | No |
| Rel./RO | 3260 | 74.43 | 9892 | 76.48 | No |
| Acq./Rel./RO | 1667 | 86.93 | 4804 | 88.58 | No |

**(c)** 4 Philosophers

| **Strategies** | **States** | **Red. (%)** | **Trans.** | **Red. (%)** | **Dead-lock** |
|---|---|---|---|---|---|
| No strategies | 304325 | 0.00 | 1340320 | 0.00 | Yes |
| **Acq**(uisition) | 123327 | 59.48 | 521992 | 61.05 | Yes |
| **Rel**(ease) | 124545 | 59.08 | 527864 | 60.62 | Yes |
| **As**(ymmetry) | 146925 | 51.72 | 631480 | 52.89 | No |
| Acq./Rel. | 34775 | 88.57 | 136496 | 89.82 | Yes |
| Acq./As. | 85725 | 71.83 | 361960 | 72.99 | No |
| Rel./As. | 44455 | 85.39 | 178168 | 86.71 | No |
| Acq./Rel./As. | 19561 | 93.57 | 75136 | 94.39 | No |
| | | | | | |
| Res. Order (**RO**) | 156675 | 48.52 | 675680 | 49.59 | No |
| Acq./RO | 86925 | 71.44 | 366896 | 72.63 | No |
| Rel./RO | 50305 | 83.47 | 204108 | 84.77 | No |
| Acq./Rel./RO | 20173 | 93.37 | 77568 | 94.21 | No |

**(d)** 5 Philosophers

| **Strategies** | **States** | **Red. (%)** | **Trans.** | **Red. (%)** | **Dead-lock** |
|---|---|---|---|---|---|
| No strategies | 7178125 | 0.00 | 39529000 | 0.00 | Yes |
| **Acq**(uisition) | 2334189 | 67.48 | 12361790 | 68.73 | Yes |
| **Rel**(ease) | 2340375 | 67.40 | 12398970 | 68.63 | Yes |
| **As**(ymmetry) | 2996250 | 58.26 | 16129250 | 59.20 | No |
| Acq./Rel. | 475359 | 93.38 | 2332320 | 94.10 | Yes |
| Acq./As. | 1497825 | 79.13 | 7915260 | 79.98 | No |
| Rel./As. | 691550 | 90.37 | 3484630 | 91.18 | No |
| Acq./Rel./As. | 235655 | 96.72 | 1132228 | 97.14 | No |
| | | | | | |
| Res. Order (**RO**) | 3191250 | 55.54 | 17227750 | 56.42 | No |
| Acq./RO | 1518225 | 78.85 | 8020772 | 79.71 | No |
| Rel./RO | 773450 | 89.22 | 3929690 | 90.06 | No |
| Acq./Rel./RO | 242387 | 96.62 | 1165100 | 97.05 | No |

aids designers to decide on, and *explicitly document*, the relative importance of the various system properties and the specific solutions they have provided for each. XCD also makes it easier to experiment with different strategies and configurations of strategies, as these are represented explicitly and externally to connectors.

Table 1a shows results from combinations of the two ordering strategies of Fig. 8a and Fig. 8b with the asymmetry strategy of Fig. 8c and the resource order strategy of Fig. 8d, for a system with 2 philosophers. Table 1b, Table 1c, and Table 1d show results for 3, 4, and 5 philosophers respectively. We used LTSA v. 2.2 with 7000 MB of RAM. Surprisingly, we see that the best state space reduction (third column, headed "**Red. (%)**") for two strategies is obtained when combining the two strategies that constrain the acquisition (**Acq.**) and release order (**Rel.**) of resources (64%, 80%, 88%, and 93% respectively), even though these do not render the system deadlock-free. These reductions are almost the double of those achieved by the strategies for deadlock-freedom (**As.**, **RO**) on their own (33%, 40%, 51%, and 58% respectively).

As these results indicate, it is not necessarily true that a designer should choose to apply a deadlock-freedom strategy first. In fact, the results obtained by the two deadlock-freedom strategies for 2 and 3 philosophers in Table 1a and Table 1b give a reason for not doing so, since they are identical. So designers have to consider a larger system, with 4 philosophers and possibly with 5, to be able to choose one deadlock-freedom strategy over another. There the two strategies produce different results (a 51% versus 48% reduction and a 58% versus a 55% one respectively). However, checking a larger system is far more expensive and may lead to state-space explosion. So we can see that constraining first with some strategies that do not meet any critical properties, as with the acquisition and release ordering strategies, is a sensible step for reducing the overall state-space. It allows designers to explore larger instances of the system, which may potentially help identify further problems, opportunities for optimization, or simply provide evidence for choosing among alternative strategies for meeting a particular property, as it does here. Designers can then easily remove some of the non-critical strategies, if they need to use the extra degrees of freedom for meeting other critical properties, e.g., performance. This is made possible by the modular nature of the strategies – adding and removing them requires no modifications to either component or connector specifications.

A connector for centralized control (with a "Butler" role) and associated evaluation results is described in a separate technical report [23].

## 6 Related Work

Research in software architectures identified the need for first-class connectors from the very beginning [15, 30]. The problems created by the non-documentation of protocols was also identified early on in [14] and a formalization of connectors was presented in [1] shortly after that – a formalization that is still being used today, e.g., [19,34]. Indeed, the connectors in CONNECT [19] follow the same general structure as Wright's (roles and glue), but seem to be specified in FSP instead of CSP. Compared to Wright [1], XCD adds the extra element of *role strategy*, and the additional constraint that connectors and strategies should *not have a glue*. As such XCD avoids the glue realizability problem – XCD connector roles are realizable by construction, as they only require access to local data (Booleans, integers, buffers, etc.). XCD also abandons the use of CSP for what we believe is a more developer-friendly approach.

Work which has been done at identifying different types of connectors [16, 27] has tended to focus at cataloguing and specifying basic interaction mechanisms, e.g., procedure calls, event buses, etc., especially since these were needed to base upon them more complex connectors. However, the use of basic interaction mechanisms as connectors in an architectural specification makes it difficult to understand what the real protocols in the system are and leads to system specifications that are at a very low level of abstraction, as is the case with AADL [11]. Indeed, designers are forced to incorporate the behaviour of the more complex connectors they wish to use into their components, decreasing their re-use potential and increasing the chance of architectural mismatch [14]. In fact, the presence of low-level connectors [16, 27] in a system architecture should alert designers that they have a potential problem. That is, they have *over-designed* the architectural description and/or have failed to describe the general protocols that are supposed to be used among their components in a way that is sufficiently abstract, and therefore understandable and analyzable. Blackboards, event buses, tuple spaces, etc., are low-level interconnection mechanisms that give precious little information on what interaction protocols a system uses and how these meet its non-functional requirements.

Languages used by practitioners suffer from this problem in particular. A connector in UML 2.0 is just a UML association, so architects must use modelling elements other than UML connectors to describe architectural connectors [20]. AADL [13] only supports certain specific, basic connector types and does not offer the possibility to define more complex connector types, while SysML [6] does not support architectural connectors at all (only UML ones).

Plasil et al.'s work [5, 31, 32] is somewhat similar to ours, in particular the need to describe component interactions as separate entities, albeit ones which still form part of the component. Instead, XCD cleanly separates component and connector behaviour, and further separates the control parts of the connectors through role strategies.

It should be noted here that the constraints introduced through strategies are orthogonal to architectural style constraints, such as those of ACME [21]. The latter are global constraints enforcing a style, while strategies are local constraints. So there are cases where the strategy constraints are met but the style ones are not, as in a pipe-and-filter style prohibiting cycles, something that cannot be enforced through role strategies.

Compared to BIP [8], XCD differs in the fact that it tries to support complex connectors as first class entities, while BIP only provides two basic connectors, for "rendezvous" and "atomic broadcast". We believe that latter can be misused very easily by designers who mistake it for "broadcast". At the same time, BIP offers a specification framework that is closer to synchronous/hardware description languages that XCD tries to avoid as we believe that languages like JML will prove much more popular with software developers.

Compared to Exogenous Connectors [24] and Reo [4], XCD differs by introducing role strategies and by not trying to remove interaction constraints from components entirely. We believe that components still need to be able to specify some interaction constraints so as to describe what they expect of their environment and how they plan to use it. Another difference is with the way a designer is expected to specify their system. XCD uses pre-/post-conditions to specify the behaviour of components, connectors, and strategies, while exogenous connectors uses a graphical representation, which to our eyes looks too much like hardware block diagrams. Reo also constructs complex connectors by the appropriate composition of simpler channel specifications, in a manner that again resembles a circuit design. We do not expect such languages to gain a significant follow up from the general software development community – they do not look like "code" enough. XCD does use some of Reo's basic connectors as channel types that link connector roles together.

## 7 Conclusions

XCD is a new connector-centric approach for designing systems, aimed at facilitating their formal analysis at an early stage. XCD views connectors as the most important architectural element and uses them to cleanly separate *functional behaviour* from *interaction behaviour*. XCD attempts to further modularize architectural specifications by separating *control behaviour* into external *controller role strategies* that can be easily combined and replaced, without having to modify the component or connector specifications. These structural characteristics of XCD mean that designers can experiment more easily with different combinations of components, connectors, and strategies, to formally evaluate the properties of their systems and the potential solutions that exist for meeting those, without having to modify the specifications of any of these elements.

Inspired by JML, XCD follows a *Design by Contract (DbC)* specification approach, through the use of *simple* pre-/post-conditions so that it is easier to use. XCD *extends DbC* in two ways. First XCD introduces a *new structure for contracts*, to distinguish between the different behaviour/contract types (functional/interaction) in a clean manner. Second, XCD extends DbC so that *service consumers can specify contractual terms too*, expressing their intended use of the services they are interested in, i.e., providing a *service "environment model"*. Finally, by foregoing the use of Wright's [1] connector glue element and instead expressing all constraints through local pre-/post-conditions, XCD ensures that *connectors can be realized by construction* and that *connectors can be easily specified even in the case where the number of roles is high (or a parameter).*

Apart from improving tool support, we are currently considering extensions of XCD so that it can deal with events (i.e., asynchronous oneway calls), and different types of interaction channels (buffered, lossy, etc.).

**Acknowledgements.** This work has been partially supported by the EU project FP7-257367 IoT@Work – "Internet of Things at Work".

## References

1. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM TOSEM 6(3), 213–249 (1997)
2. Alur, R., Etessami, K., Yannakakis, M.: Inference of message sequence charts. IEEE Trans. Software Eng. 29(7), 623–633 (2003)
3. Alur, R., Etessami, K., Yannakakis, M.: Realizability and verification of MSC graphs. Theor. Comput. Sci. 331(1), 97–114 (2005)
4. Arbab, F.: Reo: A channel-based coordination model for component composition. Mathematical Structures in Computer Science 14(3), 329–366 (2004)
5. Bálek, D., Plasil, F.: Software connectors and their role in component deployment. In: IFIP Conf. Proc. vol. 198, pp. 69–84. Kluwer (2001)
6. Balmelli, L.: An overview of the systems modeling language for products and systems development. J. of Obj. Tech. 6(6), 149–177 (2007), `www.sysml.org`
7. Basu, S., Bultan, T., Ouederni, M.: Deciding choreography realizability. In: Field, J., Hicks, M. (eds.) POPL 2012, pp. 191–202. ACM (2012)
8. Bliudze, S., Sifakis, J.: The algebra of connectors – Structuring interaction in BIP. In: EmSoft, pp. 11–20 (October 2007)
9. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 342–363. Springer, Heidelberg (2006)
10. Dehnert, J.C., Stepanov, A.A.: Fundamentals of Generic Programming. In: Jazayeri, M., Loos, R., Musser, D. (eds.) Generic Programming 1998. LNCS, vol. 1766, pp. 1–11. Springer, Heidelberg (2000)
11. Delanote, D., Van Baelen, S., Joosen, W., Berbers, Y.: Using AADL to model a protocol stack. In: ICECCS, pp. 277–281 ( April 2008)

12. Di Giandomenico, F., Kwiatkowska, M., Martinucci, M., Masci, P., Qu, H.: Dependability Analysis and Verification for CONNECTed Systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 263–277. Springer, Heidelberg (2010)
13. Feiler, P.H., Lewis, B.A., Vestal, S.: The SAE architecture analysis & design language. In: IEEE Intl. Symp. on Intell. Control, pp. 1206–1211 (October 2006), `http://www.aadl.info`
14. Garlan, D., Allen, R., Ockerbloom, J.: Architectural mismatch or why it's hard to build systems out of existing parts. In: ICSE, pp. 179–185 (April 1995)
15. Garlan, D., Shaw, M.: An introduction to software architecture. In: Adv. in SW Eng. and Knowledge Eng., pp. 1–39. World Scientific Publishing Company, Singapore (1993)
16. Hirsch, D., Uchitel, S., Yankelevich, D.: Towards a Periodic Table of Connectors. In: Ciancarini, P., Wolf, A.L. (eds.) COORDINATION 1999. LNCS, vol. 1594, p. 418. Springer, Heidelberg (1999)
17. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969)
18. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM 21(8), 666–677 (1978)
19. Issarny, V., Bennaceur, A., Bromberg, Y.-D.: Middleware-Layer Connector Synthesis: Beyond State of the Art in Middleware Interoperability. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 217–255. Springer, Heidelberg (2011)
20. Ivers, J., Clements, P., Garlan, D., Nord, R., Schmerl, B., Silva, J.R.O.: Documenting component and connector views with UML 2.0. TR CMU/SEI-2004-TR-008 (2004)
21. Kim, J.S., Garlan, D.: Analyzing architectural styles with Alloy. In: ROSATEA (July 2006)
22. Kloukinas, C.: Better abstractions for reusable components & architectures. In: ICSE-NIER – ICSE Companion, pp. 199–202. IEEE Press, Vancouver (2009)
23. Kloukinas, C., Ozkaya, M.: Xcd – Simple, modular, formal software architectures. Tech. Rep. TR/2012/DOC/01, Department of Computing, School of Informatics, City University London, Northampton Square, London, EC1V 0HB, U.K. (May 2012), ISSN 1364–4009
24. Lau, K.K., Elizondo, P.V., Wang, Z.: Exogenous Connectors for Software Components. In: Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Ren, X.-M., Wallnau, K. (eds.) CBSE 2005. LNCS, vol. 3489, pp. 90–106. Springer, Heidelberg (2005)
25. Lekeas, G., Kloukinas, C., Stathis, K.: Producing Enactable Protocols in Artificial Agent Societies. In: Kinny, D., Hsu, J.Y.-J., Governatori, G., Ghose, A.K. (eds.) PRIMA 2011. LNCS, vol. 7047, pp. 311–322. Springer, Heidelberg (2011)
26. Magee, J., Kramer, J.: Concurrency – state models and Java programs, 2nd edn. Wiley (2006)
27. Mehta, N.R., Medvidovic, N., Phadke, S.: Towards a taxonomy of SW connectors. In: ICSE, pp. 178–187 (2000)
28. Meyer, B.: Applying "Design by Contract". IEEE Computer 25(10), 40–51 (1992)
29. Musser, D.R., Stepanov, A.A.: Generic Programming. In: Gianni, P. (ed.) ISSAC 1988. LNCS, vol. 358, pp. 13–25. Springer, Heidelberg (1989)
30. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. SIGSOFT Softw. Eng. Notes 17(4), 40–52 (1992)
31. Plasil, F., Besta, M., Visnovsky, S.: Bounding component behavior via protocols. In: TOOLS, vol. (30), pp. 387–398. IEEE (1999)
32. Plasil, F., Visnovsky, S.: Behavior protocols for software components. IEEE Trans. Software Eng. 28(11), 1056–1076 (2002)
33. Rodríguez, E., Dwyer, M.B., Flanagan, C., Hatcliff, J., Leavens, G.T., Robby: Extending JML for Modular Specification and Verification of Multi-threaded Programs. In: Gao, X.-X. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 551–576. Springer, Heidelberg (2005)
34. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory, and Practice. John Wiley & Sons (2010), ISBN-13: 978-0470167748
35. Wirth, N.: Algorithms + Data Structures = Programs. Prentice-Hall (1975)