XCD – Simple, Modular, Formal Software Architectures

Christos Kloukinas   Mert Ozkaya

School of Informatics
CITY UNIVERSITY LONDON

Department of Computing
Technical Report Series
TR/2012/DOC/01
14 May 2012
ISSN 1364–4009

CONTENTS

LIST OF FIGURES

LIST OF TABLES

XᴄD – Simple, Modular, Formal Software Architectures

Christos Kloukinas, Mert Ozkaya
School of Informatics
City University London
London, U.K.
Email: c.kloukinas@city.ac.uk, mert.ozkaya.1@city.ac.uk

Abstract—**Connector-Centric Design (XᴄD) is a new approach for specifying software architectures that focuses on the use of complex connectors. In XᴄD simple interconnection mechanisms like procedure-calls, event-buses, etc. are abstracted and components take a second place. XᴄD aids the clear separation in a modular manner of the high-level functional, interaction, and control system behaviour, thus increasing the reusability of both components and connectors. As such, XᴄD allows designers to experiment with different interaction behaviours (connectors), without having to modify the functional behaviour specifications (components). It further allows designers to experiment with different control behaviours ("role strategies"), without modifying components or connectors.**

Inspired by JML, XᴄD follows a formal, Design-by-Contract approach, describing behaviour through simple pre/post-conditions, which should make it easier for practitioners to use. XᴄD extends Design-by-Contract so as to separate contracts into functional and interaction sub-contracts, and so as to allow service consumers to specify their own contractual clauses. The specifications of XᴄD connectors are completely decentralized (e.g., no "connector glue") to facilitate their realization and their refinement for further formal analyses.

***Keywords*-Software architecture; Modular specifications; Separation of functional interaction and control behaviours; Design by contract.**

I. 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. Researchers have advocated the need for components and connectors to be first-class architectural entities from the very beginning [1], [2]. However, the support for complex connectors is minimal in languages used in practice currently, e.g., AADL [3], SysML [4]. They mostly rely on simple interconnection mechanisms like procedure-calls and provide no support for specifying complex connectors, focusing instead all their attention upon components. The end result is that architectures end up more like low-level designs [5]. At the same time components have to incorporate specific interaction protocols thus reducing their reusability. Worse yet, when the component specifications do not explicitly specify which protocols they have been designed for, we have the problem of "architectural mismatch" [6], i.e., the inability to compose seemingly compatible components, due to the (undocumented) assumptions these make on their interaction with their environment.

The Connector-Centric Design (XᴄD) approach takes Wirth's equation ***"Algorithms + Data Structures = Programs"*** and advocates that at the architectural level we have ***"Connectors + Components = Systems"***, with connectors being essentially decentralized algorithms and components the equivalent to data structures [7]. XᴄD focuses on improving the structural representation of formal architectural specifications, so as to aid both their development and their formal analysis. Connectors are at the very centre of XᴄD, since it is them that are responsible for meeting system-wide, non-functional requirements that no component can meet, such as reliability, performance, etc.

A. Complex Connectors for Architectural Analysis

Let us consider n electrical resistors, $r_1, \cdots, r_n$. When using a sequential connector ($\rightarrow$), the overall resistance is computed as $R^{\rightarrow}(N, \{R_i\}_{i=1}^{N}) = \sum_{i=1}^{N} R_i$. 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 (n, r_j) we should assign to each variable (N, R_i) of the connector-derived formula. By simply enumerating the wires between resistors, as languages like AADL do, we miss the forest for the trees. Analysis becomes difficult and architectural errors can go undetected until later development phases.

B. Connector Role Strategies for Control and 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. XᴄD 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 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

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

Figure 1: The `sitdown/arise` (i_{SA}) interface

(`++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 want to specify a specific order for evaluating parameters, explicitly ***under-specifying*** the procedure-call connector specification. 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.

C. Design by Contract

Inspired by JML's [9] developer friendliness, XCD follows Design by Contract (DbC) [10] too. XCD extends DbC in two ways. First, it ***separates*** the ***functional behaviour*** of a component from its ***minimal interaction requirements***. Second, it allows ***service consumers*** to specify ***contractual clauses*** of their own.

D. Case Study

We demonstrate the different features of XCD through the dining philosophers problem, chosen because different solutions exist for it – both with no centralized control and with centralized control (a "butler"). We show how a designer can specify the system with different connectors (for decentralized and centralized control), without changing the component specifications, and then specify different control policies, without changing the specifications of either the connectors or the components.

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 complex 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.

II. XCD Components

Figure 1 shows the interface implemented by the Seat component – the Fork one (i_{GP}) has methods `get` and `put` with similar signatures. Method `sitdown` 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 `InteractionEX`. Components throw this special exception when their ***minimal interaction constraints*** (rather than functional ones) have been violated.

A. Extending Design by Contract – Different Contract Types

Figure 2a 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 respectively, i.e., the ones ***providing*** some interface and these ***using*** some interface – what in CORBA are ***facets*** and ***receptacles***. Finally, it defines sets of ***functional*** (ϕ) and ***interaction*** (χ) constraints, shown in Figure 2c and 2b respectively.

Constraints all follow the syntax (*port-expr.*, *method*, *pre-condition*, *post-condition*) and are grouped ([]) by the (*port-expr.*, *method*) pair they apply to. They are labelled as $(s|a)^{(\phi|\chi)}$ – for `sitdown/arise` ($s|a$) and for functional/interaction ($\phi|\chi$). So in s_1^ϕ, p_{seat}'s `sitdown` pre-condition is $\neg NullCaller(\texttt{c})$ and $HolderIsCaller(\texttt{c})$ its post-condition. This is a JML "normal_behaviour", unlike s_2^ϕ that throws a `NullIDEX` exception if the pre-condition $NullCaller(\texttt{c})$ is true. Constraints a_1^ϕ, a_2^ϕ are similar ones for `arise`, while a_3^ϕ covers the case when the pre-condition is $\neg CallerIsHolder(\texttt{c})$. In that case, the post-condition requires that a `WrongCallerEX` exception is thrown.

This last constraint a_3^ϕ introduces the difference between functional and (minimal) interaction constraints. Method `arise` accepts calls where the caller is not the holder and throws an exception, while `sitdown` does not specify anything about this. According to s_1^ϕ it seems it simply replaces Seat's holder with the caller. However, this is captured in Figure 2b, through Seat's ***minimal interaction constraints***. Constraint s_1^χ asks that `sitdown` be delayed until *Occupied* is false. This is expressed using the "**when**" keyword as in JML [11], 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 True. Figure 2b also specifies the minimal interaction constraints of `arise`. Constraint a_1^χ states that calling `arise` on an occupied Seat is acceptable. Constraint a_2^χ, 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 – similar to calling a method on a component without having initialized it first. The real meaning of an `InteractionEX` exception is that the component behaviour becomes ***undefined***.

Interaction constraints ***take precedence*** over functional ones and if both can throw an exception then the exception thrown is an `InteractionEX`. With $pre(\phi)$ and $post(\phi)$ standing for the pre-condition and the post-condition respectively of a constraint ϕ, the real specification of the Seat constraints is shown in Figure 2d. As shown there for a_2^χ, when an interaction exception's precondition is true, then the functional constraints are ignored. Otherwise, when the pre-condition of a normal interaction constraint

$$
\begin{aligned}
&<D = \left[\mathtt{ID}\ \mathtt{h} := \bot\right],\\
&\quad preds =\\
&\quad\begin{bmatrix}
Occupied = (\mathtt{h} \neq \bot),\\
NullCaller(\mathtt{c}) = (\mathtt{c} = \bot),\\
CallerIsHolder(\mathtt{c}) = (\mathtt{c} = \mathtt{h}),\\
NoVarAssigned = \forall v \in D(\neg assigns(v))\\
HolderIsCaller(\mathtt{c}) = (\mathtt{h}' = \mathtt{c}),\\
NoHolder = (\mathtt{h}' = \bot),
\end{bmatrix},\\
&\quad P^s = \{p^{i_{SA}}_{seat}\}, P^p = \emptyset, \phi, \chi >
\end{aligned}
$$

(a) Seat top-level specification

$$
\left[s^\chi_1 = \begin{pmatrix} p_{seat}, \mathtt{sitdown}(\mathtt{c}),\\ \mathbf{when}(\neg Occupied),\\ \mathrm{True} \end{pmatrix} \right]
$$

$$
\left[\begin{array}{l}
a^\chi_1 = \begin{pmatrix} p_{seat}, \mathtt{arise}(\mathtt{c}),\\ Occupied,\\ \mathrm{True} \end{pmatrix}\\
a^\chi_2 = \begin{pmatrix} p_{seat}, \mathtt{arise}(\mathtt{c}),\\ \neg Occupied,\\ \mathtt{InteractionEX}\ \wedge NoVarAssigned \end{pmatrix}
\end{array} \right]
$$

(b) Seat interaction constraints (χ)

$$
\left[\begin{array}{l}
s^\phi_1 = \begin{pmatrix} p_{seat}, \mathtt{sitdown}(\mathtt{c}),\\ \neg NullCaller(\mathtt{c}),\\ HolderIsCaller(\mathtt{c}) \end{pmatrix}\\
s^\phi_2 = \begin{pmatrix} p_{seat}, \mathtt{sitdown}(\mathtt{c}),\\ NullCaller(\mathtt{c}),\\ \mathtt{NullIDEX} \wedge NoVarAssigned \end{pmatrix}
\end{array} \right]
$$

$$
\left[\begin{array}{l}
a^\phi_1 = \begin{pmatrix} p_{seat}, \mathtt{arise}(\mathtt{c}),\\ \neg NullCaller(\mathtt{c}) \wedge CallerIsHolder(\mathtt{c}),\\ NoHolder\\ \quad \wedge \forall v \in (D \setminus \{\mathtt{h}\})(\neg assigns(v)) \end{pmatrix}\\
a^\phi_2 = \begin{pmatrix} p_{seat}, \mathtt{arise}(\mathtt{c}),\\ NullCaller(\mathtt{c}),\\ \mathtt{NullIDEX} \wedge NoVarAssigned \end{pmatrix}\\
a^\phi_3 = \begin{pmatrix} p_{seat}, \mathtt{arise}(\mathtt{c}),\\ \neg CallerIsHolder(\mathtt{c}),\\ \mathtt{WrongCallerEX} \wedge NoVarAssigned \end{pmatrix}
\end{array} \right]
$$

(c) Seat functional constraints (ϕ)

$$
pre(s^\chi_1) \to \bigwedge \begin{pmatrix} pre(s^\phi_1) \to post(s^\phi_1)\\ pre(s^\phi_2) \to post(s^\phi_2) \end{pmatrix} \quad pre(a^\chi_1) \to \bigwedge \begin{pmatrix} pre(a^\phi_1) \to post(a^\phi_1)\\ pre(a^\phi_2) \to post(a^\phi_2)\\ pre(a^\phi_3) \to post(a^\phi_3) \end{pmatrix} \quad pre(a^\chi_2) \to post(a^\chi_2)
$$

(d) Combination of functional and interaction pre-/post-conditions

Figure 2: Seat component specification

is satisfied, the functional constraints should also be satisfied.

If one specified contracts in the usual 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: $pre(a^\chi_1) \wedge pre(a^\phi_1) \to post(a^\phi_1)$
Case 2: $pre(a^\chi_1) \wedge pre(a^\phi_2) \to post(a^\phi_2)$
Case 3: $pre(a^\chi_1) \wedge pre(a^\phi_3) \to post(a^\phi_3)$
Case 4: $pre(a^\chi_2) \to post(a^\chi_2)$
Repeating "$pre(a^\chi_1)$" 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. Figure 3 shows how the Seat component specification would look like in pseudo JML. Non JML parts are preceded by two "`@`" characters instead of one. There are two new groups "`InteractionConstraints`" and "`FunctionalConstraints`", and the new "`UNDEFINED_BEHAVIOUR`" type to be used in the former group only. Keyword **when** cannot be used in `FunctionalConstraints` and **`Let`** constructs (in Figure 3a) introduce new predicates.

B. Extending Design by Contract – Service Consumer Contracts

In DbC service providers specify pre-/post-conditions for each method they provide but service consumers cannot express their own contractual clauses. Indeed, programming languages do not allow service consumers to even declare the services they consume. However, in component models like CORBA one declares both the services it provides (our sockets) and these 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 Figure 4a. 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 Figure 4c, ***which apply when a method does not throw an exception*** – that is why we call them "normal". On exceptions the Philosopher does not update its data. Keyword `self` denotes the `ID` of the component instance.

The Philosopher interaction constraints in Figure 4b 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 –

```
/*@ instance model ID holder;
  @ initially holder == NullID;
  @@Let Occupied
  @@  \old(holder) != NullID;
  @@Let NullCaller(c) c == NullID;
  @@Let CallerIsHolder(c)
  @@  c == \old(holder);
  @@Let HolderIsCaller(c)
  @@  holder == c;
  @@Let NoHolder holder == NullID;
  @*/
```

(a) Specification of component variables

```
/*@@InteractionConstraints
  @@ public normal_behavior
  @@  when !Occupied;
  @@FunctionalConstraints
  @public normal_behavior
  @  requires !NullCaller(c);
  @  ensures HolderIsCaller(c);
  @  assignable holder;
  @also
  @ public exceptional_behavior
  @  requires NullCaller(c);
  @  signals (NullIDEX) true;
  @  assignable \nothing;  @*/
void sitdown(ID c);
```

(b) Specification of `sitdown` constraints

```
/*@
  @@InteractionConstraints
  @@ public normal_behavior
  @@  requires Occupied;
  @@also
  @@ public UNDEFINED_BEHAVIOR
  @@  requires !Occupied;
  @@FunctionalConstraints
  @public normal_behavior
  @  requires !NullCaller(c)
  @     && CallerIsHolder(c);
  @  ensures NoHolder;
  @  assignable holder;
  @also
  @ public exceptional_behavior
  @  requires NullCaller(c);
  @  signals (NullIDEX) true;
  @  assignable \nothing;
  @also
  @ public exceptional_behavior
  @  requires !NullCaller(c)
  @     && !CallerIsHolder(c);
  @  signals (WrongCallerEX) true;
  @  assignable \nothing;
  @*/
void arise(ID c);
```

(c) Specification of `arise` constraints

Figure 3: Seat component specification in pseudo-JML

$$
\begin{aligned}
&<D = \left\{ \begin{matrix} \texttt{Bool wte} := \text{True}, \texttt{Bool hs} := \text{False}, \\ \texttt{Bool hl} := \text{False}, \texttt{Bool hr} := \text{False} \end{matrix} \right\}, \\
&\quad preds = \emptyset, P^s = \emptyset, \\
&\quad P^p = \{p^{i_{SA}}_{phil\_seat}, p^{i_{GP}}_{phil\_forkR}, p^{i_{GP}}_{phil\_forkL}\}, \phi, \chi >
\end{aligned}
$$

(a) Philosopher top-level specification

$$[(p_{phil\_seat}, \texttt{sitdown}(\texttt{self}), \textbf{when}(wte), \text{True})]$$

$$[(p_{phil\_seat}, \texttt{arise}(\texttt{self}), \textbf{when}(\neg wte), \text{True})]$$

$$[(p_{phil\_forkL}, \texttt{get}(\texttt{self}), \textbf{when}(wte), \text{True})]$$

$$[(p_{phil\_forkL}, \texttt{put}(\texttt{self}), \textbf{when}(\neg wte), \text{True})]$$

$$[(p_{phil\_forkR}, \texttt{get}(\texttt{self}), \textbf{when}(wte), \text{True})]$$

$$[(p_{phil\_forkR}, \texttt{put}(\texttt{self}), \textbf{when}(\neg wte), \text{True})]$$

(b) Philosopher interaction constraints (χ)

$$\left[\left(\begin{matrix} p_{phil\_seat}, \texttt{sitdown}(\texttt{self}), \text{True}, \\ hs' = \text{True} \wedge wte' = \neg(wte \wedge hl \wedge hr) \end{matrix}\right)\right]$$

$$\left[\left(\begin{matrix} p_{phil\_seat}, \texttt{arise}(\texttt{self}), \text{True}, \\ hs' = \text{False} \wedge wte' = \neg(wte \vee hl \vee hr) \end{matrix}\right)\right]$$

$$\left[\left(\begin{matrix} p_{phil\_forkL}, \texttt{get}(\texttt{self}), \text{True}, \\ hl' = \text{True} \wedge wte' = \neg(wte \wedge hs \wedge hr) \end{matrix}\right)\right]$$

$$\left[\left(\begin{matrix} p_{phil\_forkL}, \texttt{put}(\texttt{self}), \text{True}, \\ hl' = \text{False} \wedge wte' = \neg(wte \vee hs \vee hr) \end{matrix}\right)\right]$$

$$\left[\left(\begin{matrix} p_{phil\_forkR}, \texttt{get}(\texttt{self}), \text{True}, \\ hr' = \text{True} \wedge wte' = \neg(wte \wedge hs \wedge hl) \end{matrix}\right)\right]$$

$$\left[\left(\begin{matrix} p_{phil\_forkR}, \texttt{put}(\texttt{self}), \text{True}, \\ hr' = \text{False} \wedge wte' = \neg(wte \vee hs \vee hl) \end{matrix}\right)\right]$$

(c) ***"Normal"*** functional constraints (ϕ)

Figure 4: Philosopher component specification

$$\mathbf{CP_1} = \forall m. \bigvee_n pre(m^\chi_n) \qquad (1)$$

$$\mathbf{CP_2} = \forall m. \bigwedge_k \left(pre(m^\chi_k) \rightarrow \bigvee_n pre(m^\phi_n) \right) \qquad (2)$$

$$\mathbf{CP_3} = \forall m. \bigwedge_k \left[pre(m^\chi_k) \rightarrow \bigwedge_n \left(pre(m^\phi_n) \rightarrow \left[post(m^\phi_n) \bigwedge_{j \neq n} \left(pre(m^\phi_j) \rightarrow post(m^\phi_j) \right) \right] \right) \right] \qquad (3)$$

otherwise, Philosopher can 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. Without them designers cannot express the constraints under which the service providers must operate. They are essentially a service's ***"environment model"***.

C. Testing Architectural Components

Following the constraint semantics in Figure 2d, one needs to check that ($\mathbf{CP_1}$) the interaction pre-conditions are complete; and ***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.

In equation 1 n ranges over both the normal and exceptional interaction constraints and the predicate $\textbf{when}(\phi)$ should always evaluate to True. So for Seat's `sitdown`, we need to verify that $pre(s^\chi_1)$ holds. Being a **when** predicate, this is the case. For Seat's `arise` we can also verify that $(pre(a^\chi_1) \vee pre(a^\chi_2)) = (Occupied \vee \neg Occupied)$ 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** should be 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.

III. XCD CONNECTORS

Fork being similar to Seat we can now specify the system connectors. 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.

A. Glue-less Connectors

XCD connectors differ from those of Wright [12], 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 [13], [14] have shown that the ***choreography realization*** problem is ***undecidable*** in general. Second, 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 [15], the glue describes all possible interactions of the three roles – this ***does not scale***. 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 that is being 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 while ignoring the fictitious ones (the glue elements of the various connectors). It also makes the modelling more ***difficult to validate***. For example, in [15] the authors perform a probabilistic analysis of a market system, assigning a rate 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 map to reality.

B. Wrapper-like Connectors

In [12], a component should implement the roles it assumes, $\mathscr{L}(Comp) \subseteq \mathscr{L}(Role)$. 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.

C. Decentralized Connectors

Figure 5 shows the specification of a complex Decentralized connector for the dining philosophers. The connector defines a set of roles and interaction channels (Figure 5a). The specifications of the roles are shown in Figure 5b. Each of them has four constituent parts: a set of ***plug port variables*** (P^p_v), a set of ***socket port variables*** (P^s_v), a set of ***role data variables*** (D), and a set of ***interaction constraints*** (χ). Roles r_{seat}, r_{forkL}, and r_{forkR}, have no data or interaction constraints. Role r_{phil} uses variables to keep track of the state of resources and to control it through its interaction constraints in Figure 5c so that it only acquires resources when they are free and releases them when it already holds them. These constraints modify role variables only when the respective methods do not raise an exception. Channels in Figure 5d state which role port variables are linked to each other – all the channels we use are *rendez-vous* ones.

It should be noted that this connector is not describing 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.

D. Fundamental Connector Properties

There are ***two fundamental connector safety properties***: ($\mathbf{XP_1}$) ***local deadlock-freedom***; and ($\mathbf{XP_2}$) ***interaction exception-freedom***. Local deadlock-freedom ($\mathbf{XP_1}$) requires each connector role to not cause its component to deadlock by constraining it too much. This can be checked at a local level by showing that $\mathscr{L}(Comp \parallel Role) \subseteq \mathscr{L}(Role)$.

However, interaction exception-freedom ($\mathbf{XP_2}$) 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* True (i.e., those in Figure 4b). 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.

Of course, these two safety properties do not guarantee that the connector as a whole (or for that matter the system) will be deadlock-free. 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 IV.

$$< R = \left\{ r_{phil}, r_{seat}, r_{forkL}, r_{forkR} \right\}, Chan >$$

(a) Connector top-level specification

$$r_{phil} =$$
$$\left[\begin{array}{l} P_v^p = \left\{ pv_{phil\_seat}^{i_{SA}}, pv_{phil\_forkL}^{i_{GP}}, pv_{phil\_forkR}^{i_{GP}} \right\}, \\ P_v^s = \emptyset, \\ D = \left\{ \begin{array}{l} \texttt{Bool}\, sitting := \text{False}, \\ \texttt{Bool}\, haveLF := \text{False}, \\ \texttt{Bool}\, haveRF := \text{False} \end{array} \right\}, \chi_{phil} \end{array} \right]$$
$$r_{seat} =$$
$$\left[P_v^p = \emptyset, P_v^s = \left\{ pv_{seat}^{i_{SA}} \right\}, D = \emptyset, \chi_{seat} = \emptyset \right]$$
$$r_{forkL} =$$
$$\left[P_v^p = \emptyset, P_v^s = \left\{ pv_{forkL}^{i_{GP}} \right\}, D = \emptyset, \chi_{forkL} = \emptyset \right]$$
$$r_{forkR} =$$
$$\left[P_v^p = \emptyset, P_v^s = \left\{ pv_{forkR}^{i_{GP}} \right\}, D = \emptyset, \chi_{forkR} = \emptyset \right]$$

(b) Role (R) definitions

$$\left[\left(\begin{array}{l} pv_{phil\_seat}, \texttt{sitdown(self)}, \\ \textbf{when}(\neg sitting), sitting' = \text{True} \end{array} \right) \right]$$
$$\left[\left(\begin{array}{l} pv_{phil\_seat}, \texttt{arise(self)}, \\ \textbf{when}(sitting), sitting' = \text{False} \end{array} \right) \right]$$
$$\left[\left(\begin{array}{l} pv_{phil\_forkL}, \texttt{get(self)}, \\ \textbf{when}(\neg haveLF), haveLF' = \text{True} \end{array} \right) \right]$$
$$\left[\left(\begin{array}{l} pv_{phil\_forkL}, \texttt{put(self)}, \\ \textbf{when}(haveLF), haveLF' = \text{False} \end{array} \right) \right]$$
$$\left[\left(\begin{array}{l} pv_{phil\_forkR}, \texttt{get(self)}, \\ \textbf{when}(\neg haveRF), haveRF' = \text{True} \end{array} \right) \right]$$
$$\left[\left(\begin{array}{l} pv_{phil\_forkR}, \texttt{put(self)}, \\ \textbf{when}(haveRF), haveRF' = \text{False} \end{array} \right) \right]$$

(c) Philosopher role ***"normal"*** constraints (χ_{phil})

$$c^{\rightleftharpoons}_{p,s} = (pv_{phil\_seat}, pv_{seat}), c^{\rightleftharpoons}_{p,fl} = (pv_{phil\_forkL}, pv_{forkL}), c^{\rightleftharpoons}_{p,fr} = (pv_{phil\_forkR}, pv_{forkR})$$

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

Figure 5: Decentralized dining philosophers connector

$$<R = \left\{ r_{phil}, r_{seat}, r_{forkL}, r_{forkR}, r_{btlr} \right\}, Chan >$$

(a) Connector top-level specification

$$P_v^p =$$
$$\left\{ pv_{btlr\_seat\_p}{}^{i_{SA}}, pv_{btlr\_forkL\_p}{}^{i_{GP}}, pv_{btlr\_forkR\_p}{}^{i_{GP}} \right\},$$
$$P_v^s =$$
$$\left\{ pv_{btlr\_seat\_s}{}^{i_{SA}}, pv_{btlr\_forkL\_s}{}^{i_{GP}}, pv_{btlr\_forkR\_s}{}^{i_{GP}} \right\},$$
$$D = \left\{ \begin{array}{l} \texttt{Bool}\, sitting := \text{False}, \\ \texttt{Bool}\, haveLF := \text{False}, \\ \texttt{Bool}\, haveRF := \text{False} \end{array} \right\}, \chi_{btlr}$$

(b) ***Local*** Butler role (r_{btlr}) specification

$$P_v^p = \emptyset, P_v^s = \emptyset, D^{shared} = \{ \texttt{Int}\, diners := 0 \},$$
$$\chi_{btlr}^{shared} = \emptyset$$

(c) ***Shared*** Butler role specification

$$\left[\left(\begin{array}{l} pv_{btlr\_seat\_p}, \texttt{sitdown(c)}, \text{True}, \\ sitting' = \text{True} \wedge diners{+}{+} \end{array} \right) \right]$$
$$\left[\left(\begin{array}{l} pv_{btlr\_seat\_p}, \texttt{arise(c)}, \text{True}, \\ sitting' = \text{False} \wedge diners{-}{-} \end{array} \right) \right]$$
$$\left[\left(pv_{btlr\_forkL\_p}, \texttt{get(c)}, \text{True}, haveLF' = \text{True} \right) \right]$$
$$\left[\left(pv_{btlr\_forkL\_p}, \texttt{put(c)}, \text{True}, haveLF' = \text{False} \right) \right]$$
$$\left[\left(pv_{btlr\_forkR\_p}, \texttt{get(c)}, \text{True}, haveRF' = \text{True} \right) \right]$$
$$\left[\left(pv_{btlr\_forkR\_p}, \texttt{put(c)}, \text{True}, haveRF' = \text{False} \right) \right]$$

(d) Butler role ***"normal"*** constraints (χ_{btlr})

$$c^{\rightleftharpoons}_{p,bs} = (pv_{phil\_seat}, pv_{btlr\_seat\_s}), c^{\rightleftharpoons}_{bss,bsp} = (pv_{btlr\_seat\_s}, pv_{btlr\_seat\_p}), c^{\rightleftharpoons}_{bsp,s} = (pv_{btlr\_seat\_p}, pv_{seat}),$$
$$c^{\rightleftharpoons}_{pfl,bfls} = (pv_{phil\_forkL}, pv_{btlr\_forkL\_s}), \cdots$$

(e) Channels $(Chan)$ connecting port-variables

Figure 6: Centralized dining philosophers connector

E. Centralized Connectors

Figure 6 shows a Centralized connector for the dining philosophers. Unlike the Decentralized connector of Figure 5, this one introduces a further role, the Butler (r_{btlr}). The Butler sits among the other roles, as the connector channels (Figure 6e) indicate, and observes the interactions of the other roles. Unlike the other roles, the ***Butler has two parts*** – a ***local*** and a ***shared*** one, shown in Figure 6b and 6c respectively. While each Centralized connector instance creates a new local Butler role part, the shared part of the Butler role will be unique and shared among all instances of the Centralized connector, just like all objects of a class share the "static" class attributes. It is the shared part of the Butler that introduces a centralized element into the system. So in the final system configuration of n Philosopher, etc. components and n Centralized connectors, there will be a single Butler shared role instance that will group together the n instances of the local Butler roles of the different connector instances. The Butler role instance will be assumed by a single Butler component, which serves just as something for the Butler role to wrap itself around and does not need any data or behaviour of its own, only ports for the Butler role port variables.

$$\left[\left(pv_{phil\_forkL}, \texttt{get}(c), \textbf{when}(sitting), \text{True}\right)\right]$$

$$\left[\left(pv_{phil\_forkR}, \texttt{get}(c), \textbf{when}(sitting), \text{True}\right)\right]$$

(a) Resource acquisition order

$$\left[\begin{pmatrix} pv_{phil\_seat}, \texttt{arise}(c), \\ \textbf{when}\neg(haveLF \lor haveRF), \text{True} \end{pmatrix}\right]$$

(b) Resource release order

$$\left[\begin{pmatrix} pv_{phil\_forkL}, \texttt{get}(c), \\ \textbf{when}(c\%2 = 0 \lor haveRF), \text{True} \end{pmatrix}\right]$$

$$\left[\begin{pmatrix} pv_{phil\_forkR}, \texttt{get}(c), \\ \textbf{when}(c\%2 \neq 0 \lor haveLF), \text{True} \end{pmatrix}\right]$$

(c) Deadlock-avoidance by asymmetry

$$\left[\begin{pmatrix} pv_{phil\_forkL}, \texttt{get}(c), \\ \textbf{when}((r_{forkL}.\texttt{ID} < r_{forkR}.\texttt{ID}) \lor haveRF), \text{True} \end{pmatrix}\right]$$

$$\left[\begin{pmatrix} pv_{phil\_forkR}, \texttt{get}(c), \\ \textbf{when}(!(r_{forkL}.\texttt{ID} < r_{forkR}.\texttt{ID}) \lor haveLF), \text{True} \end{pmatrix}\right]$$

(d) Deadlock-avoidance by resource order

Figure 7: Philosopher role strategies

$$\left[\left(pv_{btlr\_forkL\_p}, \texttt{get}(c), \textbf{when}(sitting), \text{True}\right)\right]$$

$$\left[\left(pv_{btlr\_forkR\_p}, \texttt{get}(c), \textbf{when}(sitting), \text{True}\right)\right]$$

(a) Resource acquisition order

$$\left[\left(pv_{btlr\_seat\_p}, \texttt{arise}(c), \textbf{when}\neg(haveLF \lor haveRF), \text{True}\right)\right]$$

(b) Strategy for resource release order

$$\left[\begin{pmatrix} pv_{btlr\_seat\_p}, \texttt{sitdown}(c), \\ \textbf{when}(diners < \text{numOf}(btlr_{local}) - 1), \\ \text{True} \end{pmatrix}\right]$$

$$\left[\left(pv_{btlr\_seat\_p}, \texttt{arise}(c), \textbf{when}(diners > 0), \text{True}\right)\right]$$

(c) Deadlock-avoidance butler strategy

Figure 8: Butler role strategies

IV. System Control and Design Decisions – Role Strategies

XCD *advocates the underspecification of connectors*. Additional interaction properties are to be imposed through ***modular role strategies*** [7]. These can enforce an action order, e.g., that Seat is acquired before the Forks, or render the system deadlock-free. Deadlock-freedom itself can usually be achieved through different techniques. Instead of 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.

Figure 7 shows examples of such strategies for the Philosopher role. The strategy in Figure 7a forces Seat to be acquired before the Forks, while that of Figure 7b forces Forks to be released first. Then the asymmetry strategy in Figure 7c avoids deadlocks by picking a different Fork when the `ID` of the caller is odd or even. The strategy in Figure 7d also avoids deadlocks but does so by always acquiring the Fork with the smallest `ID` first. In the Centralized connector we can also employ strategies on the Butler role, as shown in Figure 8. The strategies in Figure 8a and 8b are similar to those in Figure 7a and 7b respectively – only they are applied to the Butler role instead of the Philosopher. Finally, Figure 8c shows how to avoid deadlocks by preventing Philosophers from ever filling the table.

V. Evaluation

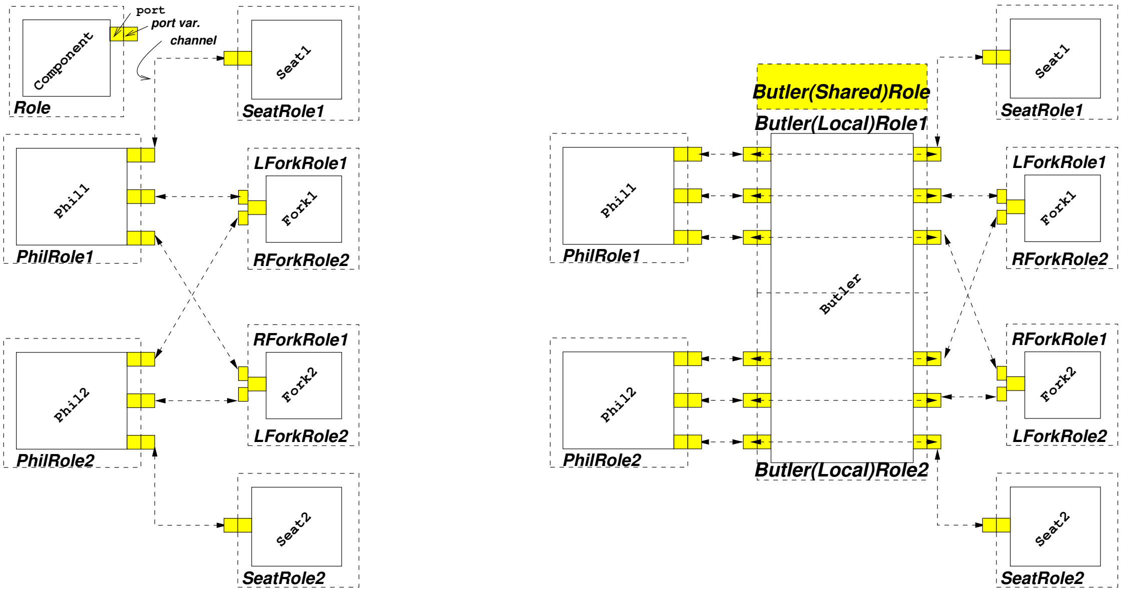
We have manually encoded these architectural specifications in the FSP process algebra [16] and have verified them automatically. In the FSP encoding each port and port variable are represented by one process that establishes the interaction constraints of their methods and for components we also employ one process per port method to establish its functional constraints. In total, we considered 12 different configurations for the decentralized system in Figure 9a and 4 different configurations for the centralized system Figure 9b, using different combinations of strategies. In all these cases our models remained the same, with the only difference being the enabling/disabling of strategies.

The different role strategies defined in Figure 7 and Figure 8 allow designers to easily experiment with controlling their system and evaluating different design decisions early on. Thus, XCD 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 independently of connectors.

Table Ia shows results from combinations of the two ordering strategies of Figure 7a and Figure 7b with the asymmetry strategy of Figure 7c and the resource-based strategy of Figure 7d, for a system with 2 philosophers. Table Ib, Table Ic, and Table Id 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 for two strategies is obtained when combining the two strategies that constrain the acquisition and release order 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 on their own (33%, 40%, 51%, and 58% respectively).

Table IIa shows results from the combined "Good Manners" strategy of Figure 8a and Figure 8b with the deadlock-free strategy of Figure 8c, for a system with 2 philosophers. Table IIb shows results for these strategies for 3 philosophers respectively.

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



(a) Decentralized configuration

(b) Centralized system configuration

Figure 9: System configurations

3 philosophers in Table Ia and Table Ib is a reason for not doing so is – they are identical. So for designers to argue why they have chosen one strategy over the other, they have to consider a larger system, with 4 philosophers and possibly with 5. 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 which 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 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.

VI. Related Work

Research in software architectures identified the need for a first-class connector notion from the very beginning [1], [2]. The problems created by the non-documentation of protocols was also identified early on in [6] and a formalization of connectors was presented in [12] shortly after that – a formalization that is still being used today, e.g., [17], [18]. Compared to [12], XCD adds the extra element of ***role strategy***, and the additional constraint that connectors and strategies should ***not have a glue***.

Work which has been done at identifying different types of connectors [19], [20] 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 [5]. 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 [6]. In fact, the presence of low-level connectors [19], [20] 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 connectors [21]. AADL [3] only supports certain specific, basic connector types and does not offer the possibility to define more complex connector types, while SysML [4] does not support connectors at all.

Plasil et al.'s work [22]–[24] 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.

Table I: Different decentralized 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

Table II: Different centralized strategy combinations

(a) 2 Philosophers

Strategies	**States**	**Red. (%)**	**Trans.**	**Red. (%)**	**Dead-lock**
No strategies	30953	0.00	79726	0.00	Yes
Good Manners	12779	58.71	33214	58.34	Yes
Deadlock-Freedom	18265	40.99	47374	40.58	No
GM+DF	3757	87.86	9894	87.59	No

(b) 3 Philosophers

Strategies	**States**	**Red. (%)**	**Trans.**	**Red. (%)**	**Dead-lock**
No strategies	735376	0.00	1936512	0.00	Yes
Good Manners	196118	73.33	521484	73.07	Yes
Deadlock-Freedom	584136	20.57	1543092	20.32	No
GM+DF	80560	89.05	215940	88.85	No

It should be noted here that the constraints introduced through strategies are orthogonal to architectural style constraints, such as those of ACME [25]. 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.

VII. Conclusions

XCD is a new connector-centric approach for designing systems, which facilitates 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 further modularizes 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 very easily experiment 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 the three types of elements.

Inspired by JML, XcD follows a ***Design by Contract (DbC)*** specification approach so that it is easier to use. **XcD *extends DbC*** in two ways. First XcD introduces a ***new structure for contracts*** so as 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"***.

Apart from developing tools to support the XcD approach, we are currently considering extensions of it so that it can deal with events (i.e., asynchronous 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] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Softw. Eng. Notes*, vol. 17, no. 4, pp. 40–52, Oct. 1992.

[2] D. Garlan and M. Shaw, "An introduction to software architecture," in *Adv. in SW Eng. and Knowledge Eng.* Singapore: World Scientific Publishing Company, 1993, pp. 1–39.

[3] P. H. Feiler, B. A. Lewis, and S. Vestal, "The SAE architecture analysis & design language," in *IEEE Intl Symp. on Intell. Control*, Oct. 2006, pp. 1206–1211, www.aadl.info.

[4] L. Balmelli, "An overview of the systems modeling language for products and systems development," *J. of Obj. Tech.*, vol. 6, no. 6, pp. 149–177, Jul.–Aug. 2007, www.sysml.org.

[5] D. Delanote, S. Van Baelen, W. Joosen, and Y. Berbers, "Using AADL to model a protocol stack," in *ICECCS*, Apr. 2008, pp. 277–281.

[6] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural mismatch or why it's hard to build systems out of existing parts," in *ICSE*, Apr. 1995, pp. 179–185.

[7] C. Kloukinas, "Better abstractions for reusable components & architectures," in *ICSE-NIER – ICSE Companion.* Vancouver, Canada: IEEE Press, May 2009, pp. 199–202.

[8] S. Bliudze and J. Sifakis, "The algebra of connectors – Structuring interaction in BIP," in *EmSoft*, Oct. 2007, pp. 11–20.

[9] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll, "Beyond assertions: Advanced specification and verification with JML and ESC/Java2," in *FMCO'05 – Formal Methods for Comp. and Obj.*, ser. LNCS, vol. 4111. Springer, 2006, pp. 342–363.

[10] B. Meyer, "Applying "design by contract"," *IEEE Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[11] E. Rodríguez, M. B. Dwyer, C. Flanagan, J. Hatcliff, G. T. Leavens, and Robby, "Extending JML for modular specification and verification of multi-threaded programs," in *ECOOP*, ser. LNCS, vol. 3586. Springer, 2005, pp. 551–576.

[12] R. Allen and D. Garlan, "A formal basis for architectural connection," *ACM TOSEM*, vol. 6, no. 3, pp. 213–249, Jul. 1997.

[13] R. Alur, K. Etessami, and M. Yannakakis, "Inference of message sequence charts," *IEEE Trans. Software Eng.*, vol. 29, no. 7, pp. 623–633, 2003.

[14] ——, "Realizability and verification of MSC graphs," *Theor. Comput. Sci.*, vol. 331, no. 1, pp. 97–114, 2005.

[15] F. Di Giandomenico, M. Z. Kwiatkowska, M. Martinucci, P. Masci, and H. Qu, "Dependability analysis and verification for connected systems," ser. LNCS, vol. 6416, 2010, pp. 263–277.

[16] J. Magee and J. Kramer, *Concurrency – state models and Java programs*, 2nd ed. Wiley, 2006.

[17] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice.* John Wiley & Sons, 2010, ISBN-13: 978-0470167748.

[18] V. Issarny, A. Bennaceur, and Y.-D. Bromberg, "Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability," ser. LNCS, vol. 6659, 2011, pp. 217–255.

[19] N. R. Mehta, N. Medvidovic, and S. Phadke, "Towards a taxonomy of SW connectors," in *ICSE*, 2000, pp. 178–187.

[20] D. Hirsch, S. Uchitel, and D. Yankelevich, "Towards a periodic table of connectors," in *COORDINATION*, ser. LNCS, vol. 1594, 1999, p. 418.

[21] J. Ivers, P. Clements, D. Garlan, R. Nord, B. Schmerl, and J. R. O. Silva, "Documenting component and connector views with UML 2.0," TR CMU/SEI-2004-TR-008, 2004.

[22] D. Bálek and F. Plasil, "Software connectors and their role in component deployment," ser. IFIP Conf. Proc., vol. 198. Kluwer, 2001, pp. 69–84.

[23] F. Plasil, M. Besta, and S. Visnovsky, "Bounding component behavior via protocols," in *TOOLS (30).* IEEE, 1999, pp. 387–398.

[24] F. Plasil and S. Visnovsky, "Behavior protocols for software components," *IEEE Trans. Software Eng.*, vol. 28, no. 11, pp. 1056–1076, 2002.

[25] J. S. Kim and D. Garlan, "Analyzing architectural styles with Alloy," in *ROSATEA*, Jul. 2006.