Highly Analysable, Reusable, and Realisable Architectural Designs with XCD

Mert Ozkaya and Christos Kloukinas

School of Informatics, City University London
London EC1V 0HB, UK
{mert.ozkaya.1,C.Kloukinas}@city.ac.uk

Abstract. Connector-Centric Design (XCD) is a new approach to specifying software architectures. XCD views complex connectors as highly significant in architectural designs, as it is the complex connectors that non-functional quality properties in systems can emanate from. So, XCD promotes in designs a clean separation of connectors (interaction behaviours) from components (functional behaviours). Designers can then specify connectors in detail explicitly thus easing the analysis of system designs for quality properties. Furthermore, XCD separates control behaviour from connectors as control strategies. Architectural designs in XCD thus become highly modular with re-usable components, connectors, and control strategies (representing design solutions for quality properties). The end result is the eased architectural experimentation with different design solutions by re-using components/connectors and formal analysis of these solutions to find out the optimal ones.

1 Introduction

Architectural description of systems is described in terms of two main elements – *components* and *connectors* [1]. However, current design languages (e.g., SysML [2] and AADL [3]) do not support connectors as first-class elements. That is, they only make available simple connector types (e.g., procedure call and event broadcast) and high-level *complex* connectors (i.e., interaction protocols) are not supported. This leads to architectural designs being more like low-level specifications [4]. In such cases, complex connectors would be either not specified or at best integrated into components. However, omitting the specification of complex connectors results in architectural mismatch [5], i.e., the inability to compose independent components to a whole system due to wrong assumptions they make about their interaction. Furthermore, formal analysis of architectural designs w.r.t. quality properties is hindered too; it is the complex connectors from which system-level global issues emanate from. When the behaviour of a complex connector is instead integrated into components, the analysis becomes more difficult as designs become less modular. It is almost as trying to analyse a program where procedures have been replaced by *goto* statements.

Complex Connectors. are essentially *interaction protocols* specifying at a *high-level* how interacting system components are to be composed into an entire

T.-h. Kim et al. (Eds.): ASEA/DRBC 2012, CCIS 340, pp. 72–79, 2012.

system. Let us consider n trains, t_1,..., t_n, which operate on a single station (as is the case in London Underground). They interact with each other through a signaling interaction protocol that instructs trains how to behave in particular cases. For instance, trains on their way to a station have to reduce their speed if the present train in the station experiences a delay. Or the trains might have to stop right on the track depending on an issue that might break out on the station. So, the interaction protocol herein is the one that determines how trains can interact properly without resulting in safety issues (e.g. collision) and thereby is the key part for analyses against non-functional quality properties (e.g., safety). On the other hand, the trains themselves are unaware of each other and operate independently. Therefore, if there were no interaction protocols coordinating their behaviour, they would possibly collide leading to safety issues.

In this paper, we introduce our new *connector-centric* approach (XcD) to specifying software architectures. Inspiring from Wright ADL [6], XcD separates *connectors* (representing interaction protocols) from *components* (representing functional behaviour). Thus, architectural designs can be easier to (i) understand, (ii) develop and (iii) more importantly analyse. Indeed, the complex connectors, where system-level quality issues emanate from, now become explicit in designs. Connectors in XcD, unlike Wright connectors enforcing centralised glue, are *decentralised* thus rendering distributed system designs realisable. Furthermore, to maximise modularity XcD separates control behaviour from connectors as *control strategies*. Design solutions for quality properties can then be specified externally to connectors. This eases the architectural experimentation with different alternative design solutions without modifying components/connectors. Thus, architectural design with alternative solutions for quality properties can be formally analysed easily and the optimal solutions can be explored early on.

2 Component Specification in XCD

The functional units in systems, components are specified in XcD with (i) ports ($P^{e,r,s,p}$) representing the points of interaction with their environment, (ii) data (D) representing the component state, and (iii) functional and (minimal) interaction constraints (FC and IC respectively).

Ports are similar to those of CORBA [7] – emitter ports (P^e) and recipient ports (P^r) that emit and receive *events* respectively; socket ports (P^s) and plug ports (P^p) that provide and require *methods* respectively. Here events and methods are grouped into interfaces (e.g., $i_{get,set}$ comprising *get* and *set* events) which are then supported by ports (e.g., $p^{i_{get,set}}_{user\_emits}$).

Functional and interaction constraints represent the functional and the *minimal* interaction behaviours of a component respectively. The former allows for specifying the acceptable arguments for methods/events accessible via ports; the latter, if desired, for specifying (i) the particular manner in which the component wants to behave (i.e., the order of actions), or (ii) the conditions under which it does not know how to behave thus leading to interaction exception. XcD constraints are specified following the well-known Design by Contract (DbC) approach

[8], inspired by JML [9] too. The syntax for XCD constraints is thus: *(port, method-/event, pre-condition, post-condition)*, stating that when a method/event action occurs via a port, if the pre-condition is met, then the post-condition is to be met.

In Fig. 1 and Fig. 2 the *user* and *memory* components interacting via shared-data connector are specified in XCD. The user specification does not have interaction constraints (i.e., $\mathtt{IC}_{\mathtt{user}} = \emptyset$), meaning that its instances emit or receive events in any order. Whereas in the memory, $\mathtt{IC}_{\mathtt{mem}}$ states that (c_1) upon receiving event *set* via the port $p_{mem\_receives}$, the component state is to be updated, setting $initialised_m$ to $True$, (c_2) upon receiving event *get* from the users, if $initialised_m$ is $True$, then the event is received successfully (i.e., post-condition is $True$), else, as stated in the last constraint (c_3), an *Interaction Exception* (i.e., `Int_EX`) is to be thrown. Thus, the memory does not know what to do in case it receives event *get* before event *set* causing access to uninitialised data. As for the functional constraints, neither component has any (i.e., $\mathtt{FC}_{\mathtt{user/mem}} = \emptyset$), as the events (i.e., *get* and *set*) emitted or received via the ports do not have parameters.

$$\begin{bmatrix} \mathtt{P^e} : \{p^{i_{get,set}}_{user\_emits}\}, \mathtt{P^r} : \{p^{i_{get,set}}_{user\_receives}\},\ \mathtt{D} : \{\mathtt{Bool}\ initialised_u = \text{False}\}, \\ \mathtt{FC}_{\mathtt{user}} : \emptyset,\ \mathtt{IC}_{\mathtt{user}} = \emptyset \end{bmatrix}$$

Fig. 1. User Component Specification

$$\begin{bmatrix} \mathtt{P^e} : \emptyset,\ \mathtt{P^r} : \{p^{i_{get,set}}_{mem\_receives}\},\ \mathtt{D} : \{\mathtt{Bool}\ initialised_m = \text{False}\}, \\ \mathtt{FC}_{\mathtt{mem}} : \emptyset,\ \mathtt{IC}_{\mathtt{mem}} \end{bmatrix}$$

$$IC_{mem} : \left\{ \begin{array}{l} c_1 : \big(p_{mem\_receives}, set, True, initialised_m\big) \\ c_2 : \big(p_{mem\_receives}, get, initialised_m, True\big) \\ c_3 : \big(p_{mem\_receives}, get, \neg initialised_m, Int\_EX\big) \end{array} \right\}$$

Fig. 2. Memory Component Specification

3 Connector Specification in XCD

The *high-level interaction protocols* among components, connectors are specified in XCD with *roles* and *channels*. Depicted in Fig. 3, shared-data connector, coordinating access to a (shared) memory by users, is specified in XCD. Connector roles are described in terms of *data-variables*, *port-variables*, and *interaction constraints*. Roles essentially represent the interaction behaviour of components interacting via the connector. Indeed, the user components specified in Fig. 1 will assume the user and initialiser roles in Fig. 3a and Fig. 3c respectively; the memory components in Fig. 2 assume the memory role in Fig. 3b. Connector channels represent the communication links between interacting roles; each is specified with a *pair of port-variables* and a *communication type*, e.g., synchronous and lossy. In Fig. 3d, the channels specify which recipient port-variable receives events of which emitter port-variable, e.g., ch_3 stating that the recipient port-variable of the memory role receives the events emitted by the emitter port-variable of the initialiser role through synchronisation (i.e., *sync*) of events.

r_{user} :
$\begin{bmatrix} \mathtt{P^e} : \{pv^{i_{get,set}}_{user\_emits}\}, \mathtt{P^r} : \{pv^{i_{get,set}}_{user\_receives}\}, \\ \mathtt{D} : \{\mathtt{Bool}\ initialised_u = \text{False}\}, \mathtt{IC_{user}} : \emptyset \end{bmatrix}$

(a) User Role specification r_{user}

r_{memory} :
$\begin{bmatrix} \mathtt{P^e} : \emptyset, \mathtt{P^r} : \{pv^{i_{get,set}}_{mem\_receives}\}, \mathtt{D} : \{ \\ \mathtt{Bool}\ initialised_m = False\}, \mathtt{IC_{mem}} : \emptyset \end{bmatrix}$

(b) Memory Role specification r_{memory}

r_{init} :
$\begin{bmatrix} \mathtt{P^e} : \{pv^{i_{get,set}}_{init\_emits}\}, \mathtt{P^r} : \{pv^{i_{get,set}}_{init\_receives}\}, \\ \mathtt{D} : \{\mathtt{Bool}\ initialised_i = \text{False}\}, \mathtt{IC_{init}} \end{bmatrix}$

IC_{init} :
$\begin{Bmatrix} c_1 : (pv_{init\_emits}, get, \text{True}, \text{True}) \\ c_2 : (pv_{init\_emits}, set, \text{True}, initialised_i) \end{Bmatrix}$

(c) Initialiser Role specification r_{init}

$ch_1 : [sync, (pv_{init\_emits}, pv_{user\_receives})]$ $ch_2 : [sync, (pv_{user\_emits}, pv_{init\_receives})]$
$ch_3 : [sync, (pv_{mem\_receives}, pv_{init\_emits})]$ $ch_4 : [sync, (pv_{mem\_receives}, pv_{user\_emits})]$

(d) Channel Specifications for the Shared-Data

Fig. 3. XcD Connector Specification for Shared-Data Connector

Interaction protocols are imposed through role interaction constraints on the component(s) assuming the roles. The interaction constraints of roles are intended for enforcing components to behave in a particular manner (i.e., through imposition of specific order on action execution). Components can thus be avoided from getting involved in unexpected interactions due mainly to actions (e.g., event listening and receipt) performed in wrong order. The end result is then a set of components interacting with their environments successfully to compose the whole system. IC_{init} in Fig. 3c, for instance, states that (c_1) the emission of event *get* via the port-variable pv_{init_emits} occurs with no pre- and post-condition (i.e., both are $True$), and (c_2) upon emission of the event *set* with no pre-condition, $inititalised_i$ is set to $True$. Effectively, the initialiser role= allows assuming user components to perform actions (i.e., the emission of event *get* or *set*) in any order.

connector $\mathtt{Shared\_Data_2}$ =
role Initializer=**let** A = set→ A ⊓ get→A ⊓ ℘
in set→A
role User = set→User ⊓ get→User ⊓ ℘
glue = **let** Continue=Initializer.set→Continue □ User.set→Continue □
Initializer.get→Continue □ User.get→Continue □ ℘
in Initializer.set→Continue ℘

Fig. 4. Wright Connector Specification for Shared-Data Connector, reprinted from Figure 4 of [6]

Decentralised Connectors are adopted in XcD, distinguishing it from the Wright ADL [6] which also focuses mainly on connectors. Unlike XcD, Wright enforces a *centralised glue* for connector specification which composes the behaviour

of contained roles into a whole system behaviour. Indeed, as depicted in Fig. 4, Wright specification of the shared-data connector includes a glue coordinating the events of the user and initialiser roles. However, the glue, just like SOA choreographies [10], is problematic. As shown in [11, 12], realisation of choreography specifications is not always possible thus leading to systems that are impossible to implement.

4 Control Strategy Specification in XCD

XCD, unlike similar approaches, e.g., Wright [6] and Exogenous Connectors [13], introduces a new architectural abstraction for specifying design solutions.

XCD control strategy is specified by means of *external* interaction constraints that refer to a specific connector *role*. The constraints herein are intended for constraining the role behaviour further so that the assuming components obey an additional order of action execution. The end result is to be the satisfaction of the design solution (e.g., for quality properties) represented by the control strategy. For instance, memory in Fig. 2 might be accessed before its state-data is initialised thus leading to safety issues (e.g., deadlock, as the memory throws interaction exception where users expect successful termination). One can avoid this by specifying a control strategy for the memory role, as in Fig. 5. The $\texttt{IC}_{\texttt{str}}$ in Fig. 5 states that (c_1) event *get* is not allowed to be received by the memory role until *when* pre-condition, $initialised_m$ evaluating to $True$, is met. Thus, interacting with users memory always receives event *set* first to initialise its data which prevents it from throwing exception and thereby causing deadlock.

Unlike other approaches supporting design solutions (e.g., aspects in AO-ADL [14]), control strategies, e.g., in Fig. 5, are specified *externally* to connector specifications. The end result of such highly modular architectural designs is, as shown in the evaluation section, the eased architectural experimentation with different design solutions and their analysis for quality properties.

$$\begin{bmatrix}\texttt{r\_memory},\\ \texttt{IC}_{\texttt{str}}\end{bmatrix} IC_{str} : \left\{ c_1 : \begin{pmatrix} pv_{mem\_receives},\ get,\\ \textbf{when}(initialised_m),\ \text{True}\end{pmatrix}\right\}$$

Fig. 5. XCD Control Strategy Specification for the Memory Role

5 Evaluation

We have developed the first version of a toolset, used as plugin to Eclipse [15], that allows designers (i) to specify their system architectures in XCD and (ii) to *automatically* translate their models into formal specification in FSP formalism [16]. FSP formal specifications can then be automatically analysed through model checkers e.g., LTSA. Through our toolset, we have automatically encoded the architectural specifications, described in Fig. 1, Fig. 2 Fig. 3, and Fig. 5, in FSP. Our goal herein was to show how it helps in design to improve modularity

(i.e., separate functional, interaction, and control behaviours), which is introduced with XCD. For simplicity, herein we considered 2 different configurations of the shared-data system: one (*MemoryInitialised*) with the control strategy specified in Fig. 5 and another (*NoStrategies*) with no strategy. Interestingly, both configurations comprise the same component and connectors; it is only the strategy employed on the former that distinguishes it from the latter.

Table 1 shows the formal analysis results for these 2 configurations with one user [1]. Table 2, Table 3, and Table 4 show the results for 3, 5, and 7 users respectively. Unsurprisingly, the strategy is successful in avoiding deadlocks, resulting from users accessing un-initialised data. Moreover, modelling formal specifications as state-machines, LTSA ends up with the same number of states (i.e., 5) for the shared-data's formal specification, regardless of which configuration is applied. This shows that the design solution represented by the strategy *MemoryInitialised* not only avoids deadlock but also maintains state-space efficiency of the formal model. Hence, designers can better utilise the state space during the formal analysis by, e.g., analysing their systems with more users.

With the external control strategy we easily experimented with a design solution, without modifying components/connectors, and further analysed our system design for safety property (i.e., deadlock freedom). If the experiment results were not satisfactory, we could so easily employ different strategies and analyse the system design with them. Hence, highly modular XCD lets designers easily analyse their designs with alternative solutions for quality properties.

Table 1. Configurations with 1 Users

Strategies	**State**	**Deadlock**
NoStrategies	5	Yes
MemoryInitialised	5	No

Table 2. Configurations with 3 Users

Strategies	**State**	**Deadlock**
NoStrategies	5	Yes
MemoryInitialised	5	No

Table 3. Configurations with 5 Users

Strategies	**State**	**Deadlock**
NoStrategies	5	Yes
MemoryInitialised	5	No

Table 4. Configurations with 7 Users

Strategies	**State**	**Deadlock**
NoStrategies	5	Yes
MemoryInitialised	5	No

6 Related Work

Widely used modelling languages, e.g., UML [17] and SysML [2], unlike XCD, neglect connectors and offer mere associations that can expose only wire-like connections between components. ADLs, e.g., Darwin [18] and Rapide [19] suffer from the same problem too; while Darwin views connectors as bindings between

[1] We used LTSA version 2.2. with 2000 MB of RAM.

required and provided service of components, Rapide as connections between input events and output events released by components. Better yet, AADL [3] introduces a set of pre-defined connector types. However, it does not allow specifying any different complex connector types.

The lack of interest to connectors has been spotted earlier by Garlan [5], and subsequently Wright ADL [6] has been developed, which formalises connectors. XcD aims at enhancing Wright by going further and separating also the control behaviours from connectors. Moreover, XcD, unlike Wright, does not enforce (centralised) glue specification for connectors and thus render distributed system designs realisable. Viewing connectors as first-class entities, Plasil et all.'s [20] work is also similar to XcD. However, they allow for compound connectors that can encapsulate components too. This is avoided in XcD to maximise the understandability, re-usability and analysability of architectural designs. Exogenous connectors [13], like XcD, promotes clean separation of components from connectors. However, control behaviour is scattered inside the exogenous connectors, which by contrast in XcD is specified externally as control strategies.

BIP [21], like XcD, separates control behaviours from connectors. However, it neglects connectors, viewed as first-class elements by XcD, and supports only rendezvous and broadcast interaction methods. Similarly, AO-ADL [14] introduces aspect for design solutions, which are, unlike control strategies, specified inside connector specifications and thus the re-usability of connectors and the experimentation with different aspects get hindered.

7 Conclusion and Further Work

Connector-Centric Design (XcD) introduces a new architectural modelling language that aims to revive the complex connectors in architectural designs. To this end, XcD, inspired from Wright ADL, cleanly separates in designs connectors (interaction behaviour) from components (functional behaviour). However, unlike Wright, XcD adopts decentralised connectors that do not allow for glue-like specifications which lead to un-realisable system designs. Furthermore, like BIP language, XcD separates design solutions (control behaviour) from connectors and introduces them as control strategies. Hence, designers can specify realisable system architectures in a highly modular way through re-usable components, connectors, and control strategies. This significantly eases the experimentation with different combination of components/connectors/control-strategies and also the formal analysis w.r.t. quality properties. Indeed, with external control strategies, designers can easily conduct formal analysis of their system with different design solutions by re-using components/connectors, and decide early on the optimal design solutions that best meet quality properties.

Currently, we are working on extending XcD so that it supports channel types and also extending the XcD language too to improve its expressiveness.

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

References

1. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. SIGSOFT Softw. Eng. Notes 17(4), 40–52 (1992)
2. Balmelli, L.: The systems modeling language for products and systems development. Journal of Object Technology 6(6), 149–177 (2007)
3. Feiler, P.H., Gluch, D.P., Hudak, J.J.: The Architecture Analysis & Design Language (AADL): An Introduction. Technical report, Software Engineering Institute (2006)
4. Delanote, D., Baelen, S.V., Joosen, W., Berbers, Y.: Using aadl to model a protocol stack. In: ICECCS, pp. 277–281. IEEE Computer Society (2008)
5. 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 (1995)
6. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Trans. Softw. Eng. Methodol. 6(3), 213–249 (1997)
7. Wang, N., Schmidt, D.C., O'Ryan, C.: Component-based software engineering, pp. 557–571. Addison-Wesley, Longman Publishing Co., Inc., Boston (2001)
8. Meyer, B.: Applying "design by contract". IEEE Computer 25(10), 40–51 (1992)
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. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-oriented computing: a research roadmap. Int. J. Cooperative Inf. Syst. 17(2), 223–255 (2008)
11. Alur, R., Etessami, K., Yannakakis, M.: Inference of message sequence charts. IEEE Trans. Software Eng. 29(7), 623–633 (2003)
12. Alur, R., Etessami, K., Yannakakis, M.: Realizability and verification of msc graphs. Theor. Comput. Sci. 331(1), 97–114 (2005)
13. Lau, K.K., Elizondo, P.V., Wang, Z.: Exogenous Connectors for Software Components. In: Heineman, G.T., Crnkovic, I., Schmidt, H.W., Stafford, J.A., Szyperski, C.A., Wallnau, K.C. (eds.) CBSE 2005. LNCS, vol. 3489, pp. 90–106. Springer, Heidelberg (2005)
14. Pinto, M., Fuentes, L., Linero, J.M.T.: Specifying aspect-oriented architectures in ao-adl. Information & Software Technology 53(11), 1165–1182 (2011)
15. International, O.T.: Eclipse platform technical overview. Technical report (2003), `http://www.eclipse.org/whitepapers/eclipse-overview.pdf`
16. Magee, J., Kramer, J.: Concurrency - state models and Java programs, 2nd edn. Wiley (2006)
17. Ivers, J., Clements, P., Garlan, D., Nord, R., Schmerl, B., Silva, J.R.O.: Documenting component and connector views with uml 2.0. Technical Report CMU/SEI-2004-TR-008, Software Engineering Institute (Carnegie Mellon University) (2004)
18. Magee, J., Kramer, J.: Dynamic structure in software architectures. In: SIGSOFT FSE, pp. 3–14 (1996)
19. Luckham, D.C.: Rapide: A language and toolset for simulation of distributed systems by partial orderings of events. Technical report, Stanford, CA, USA (1996)
20. Bálek, D., Plasil, F.: Software connectors and their role in component deployment. In: Zielinski, K., Geihs, K., Laurentowski, A. (eds.) DAIS. IFIP Conference Proceedings, vol. 198, pp. 69–84. Kluwer (2001)
21. Bliudze, S., Sifakis, J.: The algebra of connectors - structuring interaction in bip. IEEE Trans. Computers 57(10), 1315–1330 (2008)