Better Abstractions for Reusable Components & Architectures

Christos Kloukinas
Department of Computing, City University London, Northampton Square, EC1V 0HB, U.K.

Abstract

Software architecture (SA) is a crucial component of Model Driven Engineering (MDE), since it eases the communication and reuse of designs and components. However, existing languages (e.g., UML, AADL, SysML) are lacking many needed features. In particular, they provide rudimentary support for connectors, a first-class element in the components and connectors (C&C) architectural view and one of the most reusable architectural elements. This is unfortunate, since the difficult properties that need to be guaranteed for complex systems are mainly the non-functional properties, like throughput, security and dependability, which are greatly influenced by the employed connectors.

This work reviews the basic abstractions of the C&C view of SA and examines extra architectural elements which can support the detailed, explicit and separate description of behaviour, interaction and control logic.

1. Introduction

Current standards for system architectures provide only rudimentary support for architectural connectors [1], thus impeding the description of the very basic C&C view [18, 8]. Since a UML 2.0 connector is just a UML association, architects must use modelling elements other than UML connectors to describe C&C connectors [11]. AADL [6] only supports certain specific, basic connector types and does not offer the possibility to define more complex connector types, while SysML [2] does not support connectors at all.

As such, designers either forgo describing connectors altogether or combine their description with the behavioural description of the components, thus producing unnecessarily complex models. Both these approaches obstruct architectural analysis and the early estimation of the non-functional properties of a system, such as throughput, security and dependability. Such properties are, however, crucial for a proper MDE-based architectural exploration and system development or for correctly supporting Service Level Agreements in the Service Oriented paradigm. One of the reasons for this situation is that CBSE has been advocated and used longer than SA and has a much better mapping to the constructs designers use routinely, e.g., modules, classes. Indeed, connectors are always implemented through code which is either part of or a single component, e.g., pipes. For the formal analysis of SAs, connectors are also mapped to the same structures as components, e.g., automata, which doesn't help distinguish them. Another possible reason could be the term connector itself; this aspect of SA might have been ignored less if the term used was *interaction protocol*. Finally, in certain cases we tend to use simple connectors [17, 10], e.g., procedure calls, to break down a more complex one. This hides the forest for its trees, making it difficult to see the usefulness of complex connectors and leading to system specifications which are at a very low level of abstraction, as is the case with AADL [4].

For these reasons we believe it is time to revisit the C&C view of SA and the support needed from modelling languages. Thus, we reexamine the basic elements of the C&C view and suggest ways to improve them, along with new architectural elements for easing high-level system specification. After a more or less standard treatise of components, we revisit connectors and some issues which are still problematic in them. Then, before concluding, we examine configurations and the new element of *control strategies*.

2. Specifying Components

Components are the best supported abstraction of the C&C view. As such, the definition provided herein is largely similar to what one would expect to see in other formalisms. They contain a set of *ports*, P, through which they are used, divided into *sockets*, P^s, and *plugs*, P^p. Plugs are the ports through which a component uses other components, while sockets are the ports through which it is used. Each port supports a specific *interface* of the computation/data related actions it can perform. Interfaces supported by plugs are the *required* interfaces, while those supported by sockets are the *provided* interfaces. Unlike UML, ports cannot both provide and require interfaces, since this complicates architectural descriptions unnecessarily. Note that the same interface may be both provided and required by a component, as for example is often the case with filters. Components also define *cardinality constraints* (Ca) on the number of ports which will be supporting each interface, $Ca = I \rightarrow (\mathbb{N} \cup \{0\}) \times (\mathbb{N} \cup \{\infty\})$. For example, a monitor has an explicit requirement to restrict the ports of the component to

$$
\begin{aligned}
< \quad & I = \{i_1 = \{\texttt{long read(),void write(long)}\}\},\\
& P^s = \{p_0^{i_1}\}, P^p = \emptyset,\\
& D = \{\texttt{long D}\}, B,\\
& \mathit{Pre} = \{(\texttt{D} = \bot)\, \mathcal{U}\, \texttt{write(X)}\},\\
& \mathit{Inv} = \emptyset,\\
& \mathit{Post} = \{\texttt{read()} = \texttt{D}, \texttt{write(X)} \Rightarrow (\texttt{D} = \texttt{X}\, \mathcal{U}\, \texttt{write(X')})\},\\
& \mathit{Ca} = \{(i_1, 1, 1)\}, F = \{\texttt{implemented\_as = software}\} >
\end{aligned}
$$

where B is the following:

```
long read(void) = { return D; },
void write(long d) = { D = d; }
```

Figure 1: A component for a simple monitor

exactly one, so as to enforce serial, mutual-exclusive access. The lower bound can be zero to disallow certain actions, e.g., in an element of a product family. Along with the component's behaviour, B and a set of *private* data variables, D, for specifying the preconditions (Pre), invariants (Inv) and postconditions (Post), a minimal description of a component type is $< I, P^p, P^s, D, B, \mathit{Pre}, \mathit{Inv}, \mathit{Post}, \mathit{Ca}, F >$, where F is a list of additional features (e.g., whether it should be a hardware or software component). Figure 1 shows a simple monitor component. This example shows a basic problem with components - they almost always assume a particular interaction protocol with their environment [7]. Indeed, the interface of this component declares two procedures, which assume a request-response interaction protocol. As a consequence of this, each interface must offer actions of only a single type, i.e., either procedures or notifications, and each port must support interfaces of a similar type only.

Apart from the cardinality constraints and the requirement that all procedures of an interface should be of the same type, the definition of components herein is more or less standard and well supported by the various specification languages and tools. Indeed, it is the connector element which is not supported so well.

3. Connectors - Interaction Protocols

As aforementioned, a connector represents an interaction protocol. As such, its foremost characteristic is a description of the different *roles* participating in the interaction. Roles are finite - in a protocol instantiation however there can be many, even infinite, *instances* of them. Each role defines the *interaction* primitives that components assuming that role are allowed to perform and their acceptable sequences. Roles, just like the behaviour of the components, can be modelled using different formalisms, depending on the particular interaction semantics one wishes to enforce and the analyses that need to be performed, e.g., CSP as in [1] or BIP [3]. In any case, structurally the (inter-)actions used in the definition of a role are parametric ones, e.g., `send#asynch(server, id, f(arguments))`, whose parameters are given specific values by the components that use them, e.g., `send#asynch(server, ID1, add(1, 2, 3))`. Role behavioural descriptions may also comprise private data, which effectively model the local knowledge of the roles concerning the global state of the protocol.

Another very important (and ignored) defining characteristic of a protocol is its *goal*, G, that it tries to achieve, which usually can be expressed in temporal logic. This goal, defines what should be achieved at the end of the protocol or the invariant of the protocol, if it is not to ever terminate. In complex protocols one may wish to describe a separate goal for each of the participating roles, G^R. For example, in a bus communication protocol, each sender role may have a goal to eventually transmit a message, which is different to saying that eventually a message will be transmitted, since the latter is not necessarily fair to all senders. In the request-response protocol, the goal is the reception of the response by the client, i.e., $\Box\Diamond$ `client:receive(response(id, r))`, while the roles do not need to specify their own goals. Again, the choice of logic for describing the protocol and roles goals depends on the specific protocol; some may require support for metric time or probabilities in order to express their goal, such as "the message will be transmitted within x time units, with a probability higher than p." Others may need epistemic operators to specify security properties as well. Explicit, formal goals can substantially help in the documentation, design and testing of roles, in the synthesis of control strategies or in run-time monitoring [16, 15, 19, 5].

Along with these two main characteristics of a protocol, i.e., its roles and goals, other useful *structural* characteristics can be defined as well. These have to do with extra *constraints* one may wish to impose upon the *instances* of the roles. So one can define *compatibility constraints* (Co) concerning which roles can be assumed by the same component, $\mathit{Co} = R \times R$. For example, in order to disallow recursion, role `client` can be rendered incompatible with `server`. Or one can define *cardinality constraints* (Ca) on the number of role instances that can participate in the same instantiation of a protocol, e.g., `#client = #server = 1`, $\mathit{Ca} = R \rightarrow (\mathbb{N} \cup \{0\}) \times (\mathbb{N} \cup \{\infty\})$. Finally, one can define constraints on the min/max *number of instances* of a role (IC) that a single component can assume, $IC = R \rightarrow (\mathbb{N} \cup \{0\}) \times (\mathbb{N} \cup \{\infty\})$, e.g., to state that a replica must have a single vote. In other protocols, however, a single entity can assume many instances of the same role, e.g., in Blackjack, where players can split their hand and start playing as two players. It should be noted here that the aforementioned constraints are orthogonal to architectural style constraints, such as these of ACME [12]. The latter are global constraints enforcing a particular style, while the ones presented herein are local, part of the definition of a connector, required to define when a protocol is adhered to or not. So there can be cases where the connector constraints are respected but the style ones are not. An example of this is a pipe-and-filter style which requires linear sequences of pipes and filters - this cannot be described by the local connector constraints proposed herein, which can only constrain

$$
\begin{aligned}
< \quad & R = \{\texttt{client}, \texttt{server}\}, \\
& G = \Box\Diamond \texttt{client:receive(response(id, r))}, \\
& G^R = \emptyset, D, \\
& Ca = \{(\texttt{client}, 1, 1), (\texttt{server}, 1, 1)\} \\
& Co = \{(\texttt{client}, \texttt{client}), (\texttt{server}, \texttt{server})\} \\
& IC = \{(\texttt{client}, 1, 1), (\texttt{server}, 1, 1)\} \\
> &
\end{aligned}
$$

Where D is the following:

```
client = { id = new nonce
  ; send#asynch(server,id,f(args))
  ; receive(server,id,RESULT) ; client },
server = { receive(client,ID,F(ARGS))
  ; r = F(ARGS)
  ; send#asynch(client,ID,r) ; server }
```

Figure 2: A simple request-response protocol

the system so that no local feedback loops exist (from a filter back to itself).

More formally, $< R, G, G^R, D, Ca, Co, IC >$ is the septuple defining a protocol, where G is the goal of the protocol, R the set of role names, G^R are the different role goals, D the set of role descriptions and Ca, Co, IC the sets of the cardinality, compatibility and instance constraints respectively. Figure 2 shows a simple request-response connector, where messages cannot be lost and therefore acknowledgements or replays of messages are not required.

Connector Glue is Dangerous. Unlike [1], here a "glue" does not link actions of different roles. This can be achieved with other, simpler means, such as action renaming or the use of send/receive primitives on channels, etc. In fact, glues are *dangerous*, since they can introduce errors in the protocol description by ignoring the distributed nature of the protocol and requiring unimplementable behaviours [14]. This occurs when the glue requires roles to perform an action when the roles are unsure about the real global protocol state, i.e., the glue's state, due to their partial knowledge of its global state. However, unimplementable protocols are *impossible* to specify when composing role descriptions directly without using a glue, since the role actions in this case will depend necessarily only upon their local knowledge.

4. Architectural Configuration

In most approaches on software architectures, the configuration is more or less an assignment of component ports to protocol roles. However, we believe that the architectural *configuration* needs to specify a lot more than simple port-role mappings. Indeed, for each port-role mapping there needs to be defined a *control strategy* for it. This is the strategy that a component will follow when participating in the particular protocol, to achieve the role/protocol goals. These strategies are associated with the port-role mappings since they must be local (to be implementable) and they may need information from the specific mapping, e.g., the ID of the current port/component instance. Some of the protocol strategies employed by components will be trivial ones, when a protocol has no choices, like the server of the request-response connector of Figure 2. In other cases these

```
Phil(N) = sit
  ; f[N].take
   ; f[N+1].take
  ; eat
  ; f[N+1].release
   ; f[N].release
  ; think ; Phil
```

(a) Dining Philosopher

```
Phil(N) = sit
  ; (f[N].take)
   || f[N+1].take)
  ; eat
  ; (f[N+1].release)
   || f[N].release)
  ; think ; Phil
```

(b) Reusable Dining Philosopher

```
CS(N) = ( when (N%2=0) f[N+1].take ; f[N].take
        | when (N%2=1) f[N].take ; f[N+1].take )
```

(c) A control strategy for (b)

Figure 3: Two versions of the Dining Philosopher

strategies can be quite complicated, e.g., in real-time systems [16, 13]. There a connector describes the protocol used by components to request resources with specific deadlines from the platform and synchronise with each other. Therefore, the employed strategy will be crucial in achieving the system goals - meeting deadlines, minimising jitter, etc. In cases where it is difficult to express the component strategies in this distributed fashion, it may well be necessary to redesign the connector in question with an explicit role of a centralised *controller*. Then, an extra component needs to be used which will assume the role of the controller and implement a centralised strategy for all the protocol participants. However, it should be noted that this is not always possible - if the components are distributed then the control aspect itself may need to be distributed. In certain cases, both distributed and centralised control will be needed, sometimes for better structuring the system design. For example, the distributed strategies may define local goals of the particular components, while the central one defines the overall, system goals. In either case, we consider the explicit description of component strategies to be crucial in order to accurately specify highly complex systems in a well structured manner and be able to evaluate their correctness. Indeed, through their use we may hope to disentangle the current spaghetti-like combination of function, interaction and control behaviour within components, leading to a more structured approach, where the functional component behaviour is separated from the behaviour describing their interaction with and control of their environment.

Component strategies themselves may be structured [9], e.g., as a stack, in order to specify generic strategies first and then specialised ones. Thus, a designer can define a strategy to guard against deadlocks, another one for deadlines and so on, as in [16, 13]. This will make the strategies easier to understand/optimise/validate and also make the system easier to analyse and more robust. For example it will remove obscure dependencies of different system properties with each other, such as deadlocks being hidden by temporal relations which can easily change during implementation.

Control Strategies Increase Reuse. Figure 3a shows a typical model for the philosopher component of the classic dining philosophers problem. A problem with this model is that the component has considered a specific interaction strategy with its environment, specifying the order in which it obtains and releases fork resources. By using control strategies this can be delayed, as in Figure 3b, where the specific order is to be specified by the port-role strategies. This is what will allow to model a deadlock-free system by setting the strategy of even philosophers to obtain the right fork first while odd philosophers obtain the left fork first, as in Figure 3c. This solution is not applicable on the first model, which combines its behaviour with the control strategy for its communication. It should be noted that this way of specifying the components does not change their functional behaviour. It simply identifies their interaction needs and specifies them in a general manner, without superfluous constraints. This increases the re-usability of the component and decreases the adaptation cost of the component in a new environment. More general strategies, applied at the role level, would usually need to work at a grosser grain than the ones applied at the configuration level chosen here. Thus they would unnecessarily over-constrain the system, with negative effects on other non-functional properties. However, when applying control strategies at the configuration level one can take advantage of the extra knowledge. For example, if there were fewer instances of philosophers than forks then a safe control strategy against deadlocks could very well be the empty, i.e., non-deterministic, one.

5. Conclusions

While Software Architecture is crucial for an MDE approach to the development of high quality complex systems, current modelling languages do not provide adequate support for it. The situation is particularly bad for connectors, thus one cannot easily describe and reason about interaction protocols. Yet these are crucial for analysing and meeting system-wide, non-functional properties and for disentangling the spaghetti-like component descriptions of today. This luck of support makes it even more difficult to develop secure and dependable systems. It also hinders the analysis of these systems, which is needed to support Service Level Agreements in the Service Oriented paradigm.

This work revisits the basic notions of Software Architectures (components, connectors and configuration) and discusses how these can be better supported. It introduces *constraint relations* upon both the component ports and the connector roles, assigns *goals* to roles and requires that port-role configuration bindings have associated *control strategies*, which describe how the port will behave when assuming that role. The constraints help with better describing the structural properties of an element, while the strategies help in separating the interaction and control logic from the computation one, thus leading to designs which are easier to understand and analyse. Future work includes the provision of tool support and the consideration of dynamic architectures.

Acknowledgements. This work has been funded by the European Commission Information Society Technologies Programme, as part of the projects SERENITY (contract FP6-27587) and SLA@SOI (contract FP7-216556).

References

[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM TOSEM*, 6(3):213–249, July 1997.
[2] L. Balmelli. An overview of the systems modeling language for products and systems development. *J. of Obj. Tech.*, 6(6):149–177, July–Aug. 2007 `//sysml.org`.
[3] S. Bliudze and J. Sifakis. The algebra of connectors - structuring interaction in BIP. In *EmSoft*, pages 11–20, Oct. 2007.
[4] D. Delanote, S. Van Baelen, W. Joosen, and Y. Berbers. Using AADL to model a protocol stack. In *ICECCS*, pages 277–281, Apr. 2008.
[5] A. Dingwall-Smith and A. Finkelstein. Monitoring goals with aspects. UCL `//eprints.ucl.ac.uk/837`, 2003.
[6] P. H. Feiler, B. A. Lewis, and S. Vestal. The SAE architecture analysis & design language. In *IEEE Intl Symp. on Intell. Control*, pages 1206–1211, Oct. 2006 `//aadl.info`.
[7] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *ICSE*, pages 179–185, Apr. 1995.
[8] D. Garlan and M. Shaw. An introduction to software architecture. In *Adv. in SW Eng. and Knowledge Eng.*, pages 1–39, Singapore, 1993. World Scientific Publishing Company.
[9] I. J. Hayes, M. A. Jackson, and C. B. Jones. Determining the specification of a control system from that of its environment. In *FME*, volume 2805 of *LNCS*, pages 154–169, 2003.
[10] D. Hirsch, S. Uchitel, and D. Yankelevich. Towards a periodic table of connectors. In *COORDINATION*, volume 1594 of *LNCS*, page 418, 1999.
[11] 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.
[12] J. S. Kim and D. Garlan. Analyzing architectural styles with alloy. In *ROSATEA*, July 2006.
[13] C. Kloukinas. Thunderstriking constraints with Jupiter. In *MEMOCODE*, pages 211–220. IEEE Press, July 2005.
[14] C. Kloukinas, G. Lekeas, and K. Stathis. From agent game protocols to implementable roles. In *EUMAS*, pages 27–41, Dec. 2008.
[15] C. Kloukinas and G. Spanoudakis. A pattern-driven framework for monitoring security and dependability. In *TrustBus*, volume 4657 of *LNCS*, pages 210–218, Sept. 2007.
[16] C. Kloukinas and S. Yovine. Synthesis of safe, QoS extendible, application specific schedulers for heterogeneous real-time systems. In *ECRTS*, pages 287–294, July 2003.
[17] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of SW connectors. In *ICSE*, pages 178–187, 2000.
[18] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, Oct. 1992.
[19] G. Spanoudakis, C. Kloukinas, and K. Mahbub. The runtime monitoring framework of SERENITY. In G. Spanoudakis, A. Maña, and S. Kokolakis, editors, *Security and Dependability for Ambient Intelligence*, number 13 in Information Security Series, pages 190–214. Springer-Verlag, 2009.

Better Abstractions for Reusable Components & Architectures

XBSE Before CBSE

Christos Kloukinas
Department of Computing
City University London
http://www.soi.city.ac.uk/%7Ekloukin

CITY UNIVERSITY LONDON

Current modelling languages for MDE, e.g., UML, AADL, SysML, have been developed with the CBSE paradigm in mind. As such they offer very little support for connectors.
This is unfortunate, given how reusable connectors are and how important they are for deriving the non-functional, system properties that are of most interest at the architectural level - a level where components are usually barely defined at any detail.
This work attempts to review the situation and hopefully help refocus the architectural design support towards the use of connectors.

1 Introduction

Current CBSE languages do not support the definition of complex connectors and focus instead on specifying as many component characteristics as possible; given two resistors they will attempt to describe their impedance, weight, size, ... However, the overall impedance is a property of how these two resistors are connected and how they communicate, i.e., a property of the *connector* used.
Luck of full support for the *easy*, *explicit* definition of *complex* connectors hinders the understanding and analysis of our systems.

2 XBSE - *Changing* the Paradigm

$$\text{Algorithms} + \text{Data Structures} = \text{Programs} \quad (1)$$

$$\text{Connectors} + \text{Components} = \text{Architectures} \quad (2)$$

CBSE is great for lower level designs and development but at higher levels it is the connectors that matter most - so we need to develop an ***XBSE*** approach (Connector-Based...) to designing system architectures.
XBSE essentially follows Wirth's statement (1), turning it into (2) with connectors standing for algorithms and components for data structures. Indeed, in a top-down design approach, one starts with the connectors desired and then tries to select/develop the components that can be used with them. In a bottom-up design approach one starts with the components at hand and tries to select/develop connectors that can make use of these (and in reality, different parts of a system may well be designed following different approaches). Our insistence on the importance of connectors should, therefore, not be taken as an attempt to disparage the importance of components. It is only because current approaches have shown a quite imbalanced interest in the latter that we attempt to bring connectors back into the light. Both are needed equally - but connectors need highly more support at this point.

3 More Support for XBSE's Connectors

Connectors for XBSE need better support. They need *at least*†:

Roles

- Who participates in the connector's protocol?
- How should they interact?

If component's actions are first-order predicates, connector's are higher-order ones...

Goals

- What is the overall goal of the connector?
- What is the local goal of each role?

Structural constraints

- Who can assume which/how many role(s)?

†More details in the accompanying paper.

4 XBSE's Configuration & *Control Strategies*

Architectural *configuration* should go beyond simple port-role mappings; it needs to add support for the *control strategies* to be used by components in order to meet the goals of the roles they have assumed.

5 Control Strategies & Reusability - An Example

```
Phil(N) = sit
  ; f[N].take
   ; f[N+1].take
  ; eat
  ; f[N+1].release
   ; f[N].release
  ; think ; Phil
```

(a) Dining Philosopher

```
Phil(N) = sit
  ; (f[N].take)
   || f[N+1].take)
  ; eat
  ; (f[N+1].release)
   || f[N].release)
  ; think ; Phil
```

(b) Reusable Dining Philosopher

```
CS(N) = ( when (N%2=0) f[N+1].take ; f[N].take
        | when (N%2=1) f[N].take ; f[N+1].take )
```

(c) A control strategy for (b)

Combining control with behaviour as in (a) leads to problems and reduces reuse, while separating them as in (b) & (c) solves the problem and allows the component to be used in more contexts, e.g., without a butler component.

6 How Can All These Help?

- Better separate:
  - Behaviour (components), from
  - Interaction (connectors), and
  - Control (strategies).
- Increase component reuse.
- Reduce over-specification.
- Ease the system's control.
- Better connector description.
- Re-balance the architectural description of systems:
  - Ease communication - no more a million "wires" but a small set of well understood communication protocols.
  - Analysis depends on connectors - components sometimes abstracted to a small set of numbers, e.g., period, computation time, mean time between failure, etc.

Connectors *impose* analyses - Components *endure* them...

  - Explicitly identifying connectors makes the selection of analyses methods and their application simpler.

7 Future Work - What's Missing?

Almost everything... Some points in particular are:

- Should *channels* and *connectivity patterns* (star, bus, hypercube, etc.) be included? Seems so - but how best to do it?
- Provide a link with CBSE at the level below.
- Explore the connections with Aspect-Orientation - cross-cutting system properties, interaction patterns (might provide the link to CBSE).
- Model real systems - what are their connectors? Not the wires or basic communication technology like blackboard/bus but the real protocols, like Model-View-Controller (i.e., feedback control), Master-Slave, ...
- Good tool support (for more than just drawing boxes).

Acknowledgments

This work has been funded by the European Commission Information Society Technologies Programme, as part of the projects:

- SERENITY (contract FP6-27587); and
- SLA@SOI (contract FP7-216556).

The author would also like to thank the anonymous reviewers for their very helpful feedback.