Realizable, Connector-Driven Software Architectures for Practising Engineers

Mert Ozkaya and Christos Kloukinas

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

Abstract. Despite being a widely-used language for specifying software systems, UML remains less than ideal for software architectures. Architecture description languages (ADLs) were developed to provide more comprehensive support. However, so far the application of ADLs in practice has been impeded by at least one of the following problems: (i) advanced formal notations requiring a steep learning curve, (ii) lack of support for user-defined, complex connectors, and (iii) potentially unrealizable architectural designs.
This paper proposes XCD, a new ADL that aims at supporting user-defined, complex connectors to help increase architectural modularity. It also aims to help increase the degree of reusability, as now components need not specify interaction protocols, as these can be specified independently by connectors (which increases protocol reusability too).
Connector support requires to ensure that architectural designs are always realizable, as it is currently extremely easy to obtain unrealizable ones. XCD eliminates potentially unrealizable constructs in connector specifications.
Finally, XCD employs a notation and notions from Design-by-Contract (DbC) for specifying software architecture behaviour. While DbC promotes a formal and precise way of specifying system behaviours, it is not as challenging for practising developers as process algebras that are usually employed by ADLs.

Keywords: Component Based Software Engineering; Software architecture; Modular specifications; Connector realizability; Separation of functional and interaction behaviours; Design-by-Contract.

1 Introduction

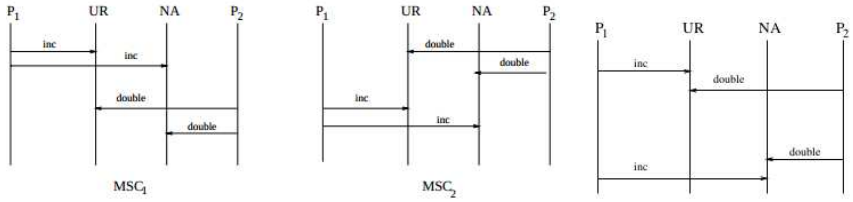
A number of specialized architecture description languages (ADLs) have been proposed for specifying software architectures [20], since the early work on software architectures [12, 23]. Currently UML has become a de facto design language for specifying and designing software systems – more practitioners use it than all other languages (e.g., AADL, ArchiMate, etc.) combined [19], even though it is less than ideal [16]. This is despite its lack of support for formal architectural analysis, unlike many ADLs that have formally defined semantics. In our view, there are three main problems that ADLs suffer from: (i) formal notations for behaviour specifications that require a steep learning curve, (ii) lack of support for complex connectors (i.e., interaction protocols),

and (iii) potential for producing unrealizable designs. Indeed, to the best of our knowledge, there is no ADL that is easy to learn, treats connectors as first-class elements and ensures that architecture specifications are realizable.

While condition (i) has been identified by practitioners as being a serious impediment to their adoption of current ADLs [19], condition (ii) is not an issue that they consider as crucial, as does a number of researchers since many ADLs do not support complex connectors. Nevertheless, we believe that it can substantially help in developing concise designs, as it increases modularity and reusability by allowing designers to reuse not only components but interaction protocols as well, thus facilitating architectural exploration and avoiding reuse-by-copy. Condition (iii) is in fact something that has not been identified at all so far to the best of our knowledge but we believe that it is crucial to identify and resolve, if a connector-centric ADL is to succeed among practitioners. Below we briefly examine each of these issues.

Formal notations Many ADLs (e.g., Wright [1], LEDA [6], SOFA [24], CONNECT [15], etc.) adopt formal notations, e.g., process algebras [4], for specifying the behaviours of architectural elements. They do so in order to enable the architectural analysis of systems, which is extremely important in uncovering serious system design errors early on in the lifetime of a project. Indeed, if such an analysis is not possible, then there is no point in using a specialized language for software architectures – even simple drawings suffice. However, ADLs employ notations that practitioners view (with reason) as having a steep learning curve [19]. Thus, practitioners end up avoiding them and use instead simpler languages, even if that means that they lose the ability to properly describe and analyse their systems – better an informal description of a system that everybody understands than a formal description of a system that people struggle understanding.

Limited support for complex connectors Another problem with many ADLs (e.g., Darwin [18], Rapide [17], LEDA [6], and AADL [10]) is that they provide limited or no support for complex connectors, treating them instead as simple connections. This is unfortunate because connectors represent the interaction patterns between components, i.e., the interaction *protocols* that are employed to achieve the system goals using the system components, such as reliability. By instead offering support only for components, architects end up with two alternatives. One is to ignore protocols, which inhibits the analysis of crucial system properties, such as deadlock-freedom, and also can lead to architectural mismatch [11], i.e., the inability to compose seemingly compatible components due to wrong assumptions these make about their interaction. The other is to incorporate the protocol behaviour inside the components themselves, which leads to complicated component behaviour that is neither easy to understand nor to analyse and makes it difficult to reuse components with different protocols, as well as to find errors in specific protocol instances. Incorporating protocol behaviour inside components is essentially following a reuse-by-copy approach, whereby each component has its own copy of the protocol constraints. On the other hand, support for protocols through first-class connectors promotes a reuse-by-call approach. There is only one instance of the protocol constraints and these are simply called wherever they are needed, making it easier to keep them correct and to replace them with those of another protocol if needed.



(a) The nuclear power plant's requirements

(b) An unavoidable bad behaviour

Fig. 1: A nuclear power plant [2]

connector `Plant_Connector` =
role P_1 = ur→na→P_1.
role P_2 = ur→na→P_2.
role NA = increment→ NA ⊓ double→NA.
role UR = increment→ UR ⊓ double→UR.
glue =P_1.ur→UR.increment→P_1.na→NA.increment
→P_2.ur→UR.double→P_2.na→NA.double→ **glue**
□P_2.ur→UR.double→P_2.na→NA.double→P_1.ur
→UR.increment→P_1.na→NA.increment → **glue**.

Fig. 2: Wright's *(unrealizable)* connector for Alur's nuclear power plant

Potentially unrealizable designs The third problem of existing ADLs is that when they do support user-defined, complex connectors, they do so in a way that can lead to unrealizable designs. All ADLs in this category follow the approach initiated by Wright [1] and require connectors to include a glue element. In Wright [1], a connector role specifies the "*obligations of [a] component participating in the interaction*" and a glue specifies "*how the activities of the [. . .] roles are coordinated.*" – a connector glue is supposed to be more than simple definition/use relationships. The fact that the glue can introduce inter-role interaction constraints is deeply problematic because these constraints cannot always be implemented in a decentralized manner by the components that assume the connector roles, as these can only observe their local state [27, 28]. In fact, it has been shown that the general problem of deciding whether a glue is realizable is undecidable [2, 3, 27, 28], so there is no general algorithm that can be implemented to warn architects that the glue they are specifying is not realizable by the existing roles. The only easy solution to realize a protocol then is to introduce yet another component that will assume the role of the glue, thus transforming all protocols into centralized ones and potentially invalidating architectural analyses concerning scalability, performance, reliability, information flows, etc.

An example of such an unrealizable protocol is the simplified nuclear power plant [2], shown in Fig. 1a. The interaction therein involves two client roles (P_1 and P_2) updating the amounts of the Uranium fuel (UR) and Nitric Acid (NA) server processes in a nuclear reactor. After the update operations, the amounts of UR and NA must be equal

to avoid nuclear accidents, for which reason we wish to allow only the sequences shown in Fig. 1a. The interaction of the two clients with the NA and UR variables, can easily be specified in Wright as in Fig. 2. Note that this glue specification does two things. First it establishes bindings between clients and servers (e.g., P_1.ur $\rightarrow$ UR.increment). Then it constraints interactions by requiring that we only allow UR.increment $\rightarrow$ NA.increment or UR.double $\rightarrow$ NA.double. This specification is however unrealizable [2] because it is impossible to implement it in a decentralized manner in a way that avoids behaviours excluded by the glue, e.g., the one depicted in Fig. 1b. The only way to achieve the desired behaviour is to introduce another role, for a centralized controller G. Roles P_1 and P_2 then need to inform G when they wish to interact with UR and NA and have G perform the interactions with UR and NA in their place.

2 Our Approach

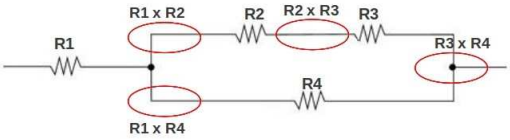
The ADL we are developing, called XCD, tries to overcome the problems identified in the previous section and offer: *(i)* first-class support for user-defined, complex connectors; *(ii)* realizable software architectures by construction; and *(iii)* a simple to understand, yet formal, language for specifying behaviour, based on design-by-contract (DbC).

2.1 Support for Complex Connectors

XCD grants connectors in software architectures first-class status, allowing designers to specify both simple interaction mechanisms and complex protocols. These can then be instantiated as many times as needed, allowing architects to simplify the specifications of their components and easily reuse the specification of complex protocols.

To illustrate how important this is for both architectural understandability and also analysis, we will use a simple example from electrical engineering. Let us consider k concrete electrical resistors, $r_1, \cdots, r_k$, i.e., our system components. When using a sequential connector ($\rightarrow$), the overall resistance is computed as $R^{\rightarrow}(N, \{R_i\}_{i=1}^{N}) = \sum_{i=1}^{N} R_i$, where N, R_i are variables (R_i correspond to connector roles), to be assigned eventually some concrete values k, r_j. If using a parallel connector ($\|$) instead, it is computed as $R^{\|}(N, \{R_i\}_{i=1}^{N}) = 1/\sum_{i=1}^{N} 1/R_i$. So the interaction protocol (connector) used is the one that gives us the formula we need to use to analyse it – if it does not do so, then we are probably using the wrong connector abstraction. The components (r_j) are simply providing some numerical values to use in the formula, while the system configuration tells us which specific value (k, r_j) we should assign to each variable (N, R_i) of the connector-derived formula. By simply enumerating the wires/connections between resistors/components, we miss the forest for the trees. This leads to architectural designs at a very low level that is not easy to communicate and develop – as [8] found the case to be with AADL.

Fig. 3a shows the number of simple connectors (identified with ellipses) that are needed in our system. It is easy to see that there are many of them and it is not so easy to identify the protocol logic, especially as the system size increases – this is the equivalent of spaghetti code. By making interaction protocols implicit in designs, analysis



(a) Simple connectors, i.e., wires

$$\rightarrow \left(r_1, \| \begin{pmatrix} \rightarrow (r_2, r_3), \\ r_4 \end{pmatrix} \right)$$

(b) Complex connectors

Fig. 3: Connectors in circuits

also becomes difficult and architectural errors can go undetected until later development phases. Indeed, we are essentially forced to reverse-engineer the architect's intent in order to analyse our system – after all, the architect did not select the specific wire connections by chance but because they form a specific complex connector. When complex connectors are employed instead as in Fig. 3b then the number of connectors to be considered is reduced substantially. This makes it much easier to understand the system and to analyse its overall resistance by taking advantage of the connector properties as:

$$R_{\rightarrow(r_1,\|(\rightarrow(r_2,r_3),r_4))} = r_1 + R_{\|(\rightarrow(r_2,r_3),r_4)} = r_1 + \frac{1}{\frac{1}{R_{\rightarrow(r_2,r_3)}} + \frac{1}{r_4}} = r_1 + \frac{1}{\frac{1}{r_2+r_3} + \frac{1}{r_4}}$$

The use of connectors allows us to separate interaction patterns/protocols from components and renders components independent from these – resistors do not need to know if they will be connected in series or in parallel. Both modularity and reuse (for both components and protocols) are increased. Unlike physical systems, where configuration patterns are enough to specify connectors as interaction in them is governed by known physical laws, software systems connectors also need to specify role interaction.

2.2 Realizable Software Architectures

Connectors in our ADL are not specified with glue-like elements. Instead, we consider connectors as a simple composition of roles, which represent the interaction behaviour of participating components, and built-in sub-connectors (i.e., links) that allow actions of one role to reach another. Coordination is now the responsibility of roles alone. If a particular property is desired then it must be shown that the roles satisfy it. But this is a problem that is decidable for finite state systems – model-checking. Thus an architect can easily specify a protocol and be sure that it has the required properties. Designers can also feel reassured that the architectural protocols are indeed realizable in principle, without the need to transform them into centralized ones, which might invalidate architectural analyses concerning scalability, performance, reliability, information flows, etc., as aforementioned.

So in the case of Fig. 1a, the architect should quickly realize that the desired property is not satisfied by the roles and opt for a centralized protocol instead, by adding a centralized controller. Thus, surprises are avoided – it becomes clear early on whether something can be made to work in a decentralized manner or not, as it is tested by

the more experienced architect. The less experienced designers do not have to waste their time trying to achieve the impossible or take the easy (and dangerous) way out and turn a decentralized protocol into a centralized one. We essentially turn the glue from constraints to be imposed, to a property that needs to be verified, thus turning an undecidable problem that the less experienced designers have to deal with, into a decidable one for them (and pushing the responsibility to resolve the issue to the more experienced architect).

2.3 Design-by-Contract for Architecture Specifications

The Java Modelling Language (JML) [7] seems to be gaining popularity among developers, as they use it for "test-driven development" and even for static analysis in some instances. XCD attempts to follow this trend so as to maximize adoption by practitioners. Thus, it departs from the ADLs that adopt process algebras, and instead follows a Design by Contract (DbC) [21] approach like JML, specifying behavioural aspects of systems through simple *pairs of method pre-/post-conditions*, in a syntax reminiscent of JML. DbC allows for a formal specification of systems, as it is based on Hoare's logic [13] and VDM's [5] rely-guarantee specification approach. DbC has so far been mainly considered for programming languages (e.g., Java through JML), which is why contracts have been restricted to provided services (i.e., class methods).

There are very few ADLs that employ DbC. The work of Schreiner et al. [26] along with the TrustME ADL [25] are some of the very few examples applying DbC at the level of software architecture. Schreiner et al.'s work transforms connectors into components themselves, which we believe loses many of the connector benefits, as these are needed to essentially drive the component interactions. Doing so through wrapper-like components [26] makes it difficult to control component required ports, i.e., the ones initiating calls. This is because a wrapper-like explicit connector can delay a call request, while a proper connector can ensure that it never gets triggered at all. TrustME does not provide support for user-specified, complex connectors at all, as it essentially follows the approach of Darwin [18], enriching it with contracts.

Our approach attempts to apply DbC in a more comprehensive manner, covering component methods and events. As we view connectors as first-class elements, we use DbC to specify their behaviour as well. XCD further extends DbC by structuring component port action contracts into separate functional and interaction parts.

3 DbC-based Specifications with XCD

Fig. 4 gives the meta-model of the XCD language. There are two main elements for specifying software architectures with XCD: components (primitive and composite ones), used to specify abstractions of computational units in a system, and connectors, that specify the complex interaction protocols of components.

We use the shared-data case study [1] to facilitate the presentation of the XCD language. In this system, user components retrieve and update some shared data stored in a memory component. The memory component accepts requests for data retrieval only if

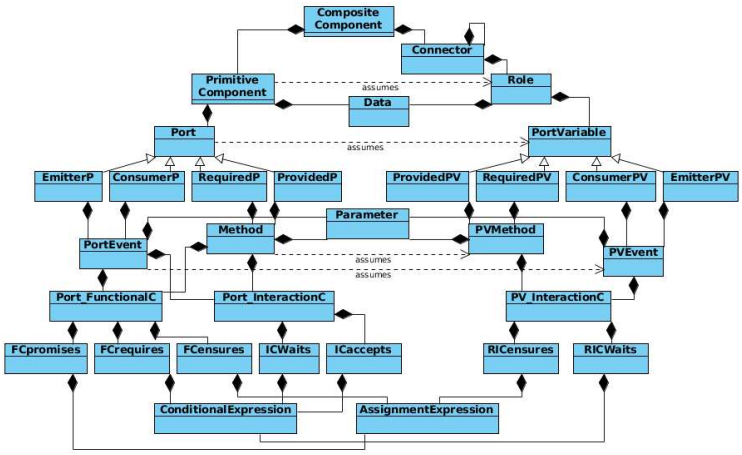


Fig. 4: Meta-model of XCD

the data has been initialized – otherwise, it rejects the request and commences a chaotic behaviour.

The XCD specification of the shared-data access is given in Fig. 5. Two primitive component types are specified, `user` in lines 1–13 and `memory` in lines 15–31. Both the user and the memory comprise data-variables (line 2 and line 16 respectively) representing their states and ports that are the points of interaction with their environment. There is also a connector type `memory2user` specified (lines 1–32 at the right side), which represents the interaction between a memory and a user. Connector memory2user uses some other connectors (here built-in ones) to establish the communication links between its role ports (lines 26–31). Its roles (`userRole` at line 6 and `memoryRole` at line 14) constrain the behaviour of the components that assume them. Finally, we specify a composite component type `sharedData` (lines 34–44 at the right side), which includes component and connector instances and represents their configuration.

Primitive Component Types Component `user` has a required port `puser_r` (lines 3–7) through which it makes method calls to its environment (i.e., the memory) to retrieve the value of some data. Port `puser_r` has a single method `get`, whose functional contract **ensures** post-assignment clause (lines 4–5) assigns the method's result to the component data – it has no pre-condition (i.e., a **requires** clause). Component `user` also has an emitter port `puser_e` (lines 8–12) to emit events. Port `puser_e` declares a single event `set`, whose functional contract **promises** clause assigns its parameter to 7 – the event has no pre-condition (i.e., a **requires** clause) or post-assignment (i.e., an **ensures** clause) . It should be noted here that while method and event **requires** clauses are conditions, method and event **promises** and **ensures** clauses are assignment sequences, not conditions. A **requires** clause specifies the functional requirements for a method call (or event) to be acceptable, while an **ensures** clause states how the state should be modified by the call. Finally, a **promises** clause states what values the parameters of a call request will have.

```
1 component user(){
2  int data:=0;
3  required port puser_r {
4   @Functional{
5    ensures: data := \result; }
6   int get();
7  }
8  emitter port puser_e {
9   @Functional{
10   promises: data_arg := 7; }
11   set(int data_arg);
12  }
13 }
14
15 component memory(int numOfUsers) {
16  bool initialized_m := false;
17  int sh_data := 0;
18  provided port pmem_p[numOfUsers] {
19   @Interaction{
20    accepts: initialized_m; }
21   @Functional{
22    ensures: \result := sh_data; }
23   int get();
24  }
25  consumer port pmem_c[numOfUsers] {
26   @Functional{
27    ensures: intialised_m := true;
28             sh_data := data_arg; }
29   set(int data_arg);
30  }
31 }
```

```
1 connector memory2user(
2                  userRole{pvuser_r,
3                           pvuser_e},
4              memoryRole{pvmem_p,
5                           pvmem_c}) {
6  role userRole {
7   required port pvuser_r {
8    int get();
9   }
10  emitter port pvuser_e {
11   set(int data_arg);
12  }
13 }
14 role memoryRole {
15  bool initialized := false;
16  provided port pvmem_p {
17   @Interaction{ waits: initialized; }
18   int get();
19  }
20  consumer port pvmem_c {
21   @Interaction{
22    ensures: initialized := true; }
23   set(int data_arg);
24  }
25 }
26 channel async
27  user2memory_m(userRole{pvuser_r},
28                memoryRole{pvmem_p});
29 channel async
30  user2memory_e(userRole{pvuser_e},
31                memoryRole{pvmem_c});
32 };
33
34 component sharedData() {// Composite
35  component user user1();
36  component user user2();
37  component memory mem(2);
38  connector memory2user
39   x1(user1{puser_r,puser_e},
40      mem{pmem_p[0],pmem_c[0]});
41  connector memory2user
42   x2(user2{puser_r,puser_e},
43      mem{pmem_p[1],pmem_c[1]});
44 }
```

Fig. 5: Shared-data access in the XCD ADL

Component `memory` has an array of provided ports `pmem_p` (lines 18–24). It uses each of these ports to provide method `get` to a different `user` component instance. Unlike the contracts of component `user`, the contract of these ports have an additional **@Interaction** part (lines 19–20). This states that a `pmem_p` port will accept a `get` method-call only if the component data `initialized_m` is true. Otherwise, the call is rejected and the component starts behaving in a chaotic manner. If a call is accepted, then the functional contract (lines 21–22) is considered, which sets the result of the method call to be the value of the component `sh_data` variable. The array of consumer ports `pmem_c` (lines 25–30) serves to receive `set` events. Reception of such an event modifies the component state.

Complex Connector Types Connector type `memory2user` (lines 1–32 at the right of Fig. 5) specifies the protocol used in the system between the memory and the users. It serves to ensure that the memory will not behave chaotically. The connector has two roles, `userRole` (lines 6–13) and *memoryRole* (lines 14–25). Role `userRole` has a required port-variable `pvuser_r` (lines 7–8), reflecting port `puser_r` of component `user`, and an emitter port-variable `pvuser_e` (lines 10–11), reflecting port `puser_e`. These port-variables do not impose any interaction constraints on the role.

Role `memoryRole` has a provided port-variable `pvmem_p` (lines 16–19) reflecting port `pmem_p` of component `memory`. Unlike the port-variables of `userRole`, this port-variable introduces extra interaction constraints on the behaviour of its methods. It requires that calls to method `get` are considered only when the role's `initialized` data is true, thus delaying them while this condition is not satisfied.

The role's consumer port-variable `pvmem_c` (lines 20–24) reflects port `pmem_c` of component `memory`. It uses its interaction contract to note that the memory has been set, through its **ensures** clause. The combination of the contracts of the two ports means that the memory cannot start behaving chaotically, as requests at non-accepting states are delayed until they are safe.

Composite Component Types The `sharedData` component type (lines 34–44 at the right of Fig. 5) includes two instances of the `user` component and a single instance of the `memory` component. The component instances are passed as arguments to the two connector instances, in lines 38–43, to bind them together and constrain their interactions.

XCD *Notation and Expressiveness* As can be seen by Fig. 5, the notation used for DbC in the XCD ADL follows a JML-like syntax, which should prove much easier for practitioners to understand and use effectively than formal languages such as process algebras. Indeed, we would expect one to be able to use XCD with minimal training. At the same time, XCD introduces connector constructs that are essentially (decentralized) algorithms (i.e., protocols) – configuration patterns of component variables, on which we have imposed additional interaction constraints. Apart from π calculus' ability to send channels as messages, which XCD does not support, the XCD ADL should allow architects to express the static architectures that one can express now with ADLs based on process algebras. However, XCD does not support dynamic architectures currently.

4 XCD Semantics

Fig. 6 shows the general behaviour of a (primitive) component described using Dijkstra's guarded command language [9]. Each instance is a concurrent process, that initializes its data and then enters a loop, executing the actions of its ports (lines 6–12) or performing a skip action (line 14). The behaviour of port actions is shown in Fig. 7 for the four different port types.

Provided and required ports (Fig. 7d and 7b) employ a pair of channels (`request` and `response`) to realize the method call interaction protocol, while emitter and consumer ports (Fig. 7c and 7a) employ a single channel (`stream`). Channels are essentially (finite) buffers of messages and a `send` action adds another message into them. A `read`

```
FORALL c ∈ Model.Components
process c ... {
// initialization of data
Start:
do
 FORALL p ∈ c.EmitterPorts
  FORALL e ∈ p.Events
   // see Fig. 7a
 FORALL p ∈ c.RequiredPorts
  FORALL m ∈ p.Methods
   // see Fig. 7b
 FORALL p ∈ c.ConsumerPorts
  FORALL e ∈ p.Events
   // see Fig. 7c
 FORALL p ∈ c.ProvidedPorts
  FORALL m ∈ p.Methods
   // see Fig. 7d
[] true → skip; // do nothing
od
}
```

(a) Component

```
// all associated
// Role Interaction Constraints
RICs(port p, action a) {
 pvs = p.associatedPortVariables;
 return ⋃_{pv∈pvs} pv.a.RICs;
}
```

(b) `RICs` for action `a` of a port `p`

Fig. 6: Semantics of components

```
[] true →
   assign_params(e.FCPromises);
   if
   [] e.ICWaits
      ∧ ⋀_{re∈RICs(p,e)} re.RICWaits → skip
   [] else → goto Start
   fi;



   assign_data(e.FCensures);
   FORALL re ∈ RICs(p, e)
    assign_data(re.RICensures);
   send(p.stream, e, e.params);
```

(a) Emitter port `p`'s event `e`

```
[] p.activeM = NULL →
 assign_params(m.FCPromises);
 if
 [] m.ICWaits
    ∧ ⋀_{rm∈RICs(p,m)} rm.RICWaits → skip
 [] else → goto Start
 fi; p.activeM := m;
 send(p.request, m, m.params);

[] readCond(p.response, m, m.result,
    p.activeM = m) →
   if
   []m.FCrequires →
    assign_data(m.FCensures);
    FORALL rm ∈ RICs(p, m)
     assign_data(rm.RICensures);
    p.activeM := NULL;
   fi
```

(b) Required port `p`'s method `m`

```
[] readCond(p.stream, e,e.params,
    e.ICWaits ∧⋀_{re∈RICs(p,e)} re.RICWaits)
 → if
   [] e.ICaccepts ∧ e.FCrequires →
      assign_data(e.FCensures);
      FORALL re ∈ RICs(p, e)
       assign_data(re.RICensures);

   // [] ! e.ICaccepts → chaos
   fi
```

(c) Consumer port `p`'s event `e`

```
[] readCond(p.request, m, m.params,
    m.ICWaits ∧⋀_{rm∈RICs(p,m)} rm.RICWaits)
 → if
   [] m.ICaccepts ∧ m.FCrequires →
      assign_data(m.FCensures);
      FORALL rm ∈ RICs(p, m)
       assign_data(rm.RICensures);
      send(p.response, m, m.result);
   // [] ! m.ICaccepts → chaos
   fi
```

(d) Provided port `p`'s method `m`

Fig. 7: Semantics of a port `p`'s actions

action retrieves some message from a channel (in a non-deterministic order). Finally, a `readCond` action retrieves a message in a non-deterministic order, with the additional

constraint that its parameters satisfy a predicate, which is passed as the fourth parameter of the action (see lines 1–2 of Fig. 7d).

As can be seen from Fig. 7, all port actions correspond to a single atomic block of guarded actions, apart from required port method requests that correspond to two atomic blocks of guarded actions (separated by a single blank line at line 9). Event and method guarded action patterns have been aligned vertically so as to make it easier to establish their similarities and differences.

An emitter port event (Fig. 7a) attempts to assign the event parameters in a way that satisfies its own and its roles' interaction constraints. If successful, it assigns the component and role data and sends the event over the port event stream channel. If the parameter values do not satisfy the interaction constraints then it simply passes control back to the component (possibly retrying). The role interaction constraints `RICs(p, e)` are the delaying constraints imposed by the port-variables assumed by the event's port and associated with this event, as shown in Fig. 6b.

A required port method (Fig. 7b) is enabled if no method request is currently active on the port, in which case it assigns the parameters of this method request and verifies that they satisfy the method's interaction constraints. If they do, it notes that the method is currently active on this port and emits the method request over the channel `p.request`. A second atomic block is enabled when there is a response for this method. So, if the functional constract pre-condition (**requires** clause) is satisfied, then, it assigns the component data according to the **ensures** clause of the method functional contract (and similarly for its roles).

Consumer events and provided methods are the dual of these, with the difference that a provided method is a single atomic block instead of two. Another difference is that, unlike the former actions, these latter port actions can cause the component to exhibit chaotic behaviour, as seen in lines 9 of Fig. 7c and 7d. This occurs when the action's delaying interaction constraints (in line 2) imposed by its associated roles are satisfied but the component interaction constraints at line 4 are not satisfied.

Race Conditions Being atomic blocks of actions, emitter/consumer port events and producer port methods do not suffer from race conditions. Required port methods on the other hand are by necessity modelled as a pair of states – one initiating a method call and another receiving the method response. The post-assignments (**ensures** clause) at the latter can suffer from two types of race-conditions. First, an assignment may attempt to use the value of some data at the pre-state, i.e., when the request was being made. If another port has modified this value, then we have a *write-read* kind of race-condition. If an assignment tries to update the value of some data that has been updated in the meantime by another port, then we have a *write-write* kind of race-condition. In our semantics we employ extra variables (not shown in the presented semantics) to identify these conflicts and notify architects about them.

4.1 Data Assignments in Contracts

XCD contracts use *assignments* to establish values for action parameters and to update the data after these actions. This is done so as to render the resulting formal models more tractable. So XCD does not accept post-conditions like "**ensures**: $0 \leq x+y+z \leq 25$;".

In order to ensure that variables x, y, z receive values that meet such a condition we would need to consider all possible combinations of their values in the range $[0, 25]$, i.e., consider $26^3 = 17576$ cases. Instead, XCD requires that the specification is transformed to a sequence of assignments, such as "**ensures**: $x \in [0,25]; y \in [0, 25-x]; z \in [0, 25-x-y]$;". Through the use of a generalized form of assignment that also supports ranges as here, XCD permits non-deterministic choices but it requires that these choices are done sequentially and only depend on constants and variables that have been assigned already. So in this case, there would be at most $26 * 3 = 78$ cases to consider, which is a substantial reduction.

Assignments are treated differently for action parameters (`assign_params`) and data updates (`assign_data`), e.g., as seen in lines 2 and 12 of Fig. 7a. This is because missing parameter assignments are added implicitly by assigning unconstrained parameters some values from their domain. This is not however done for missing data updates. It is instead assumed that these data should not be updated and retain whatever value they have at that point. The other difference between assigning parameters and data has to do with how the well-definedness of an assignment sequence is done in each case.

Well-definedness of Assignment Sequences Let us consider an assignment sequence v_i `:=` e_i, where $1 \leq i \leq n$ and v_i and e_i are a variable and an expression respectively. For `assign_params`, an assignment expression sequence as a whole is well-defined *iff* the left hand side is a parameter and the right hand side e_i of each assignment expression is an expression constructed according to the following rules:

Expression:
1. a Formula f
2. a range, i.e., $[min, max]$, where min and max are Formulas and $min \leq max$.

Formula
1. a Formula f (e.g., $+, -, /, *$) of formulas $f_1, \cdots, f_n$.
2. a Term t

Term
1. a constant, e.g., some Boolean or integer value.
2. a (known) variable, i.e., one of:
   (a) a v_j, where $j < i$
   (b) a (pre-state) value of some data d_k

As aforementioned, if the set of v_i is a strict subset of the set of parameters used in the respective action, then the other parameters are assigned values in their domain in a non-deterministic manner.

For `assign_data`, an assignment expression sequence as a whole is well-defined *iff* the left hand side is a component or role data variable and the right hand side e_i of each assignment expression is constructed according to the same rules as previously. In this case though all parameters are variables with known values, so a term can also be a parameter p_m.

Unlike `assign_params` that assigns all parameters some value by choosing some non-deterministic value from their domain if not constrained otherwise, `assign_data` does not modify data variables that have not been assigned explicitly in the model.

Table 1: Verification results

Model Size	State-vector	States		Memory	Time
	(in Bytes)	Stored	Matched	(in MB)	(in sec)
1 user	140	1954	1511	128	0.00
2 users	220	364691	575897	195	0.95
3 users	312	27327216	68152656	7024†	97.80
4 users	392	21466341	69412168	7024†	69.60

Spin (v 6.2.4) and gcc (v 4.7.2) commands used, for up to 7024MB of RAM and a search depth of 500:

```
spin -a model.pml
gcc -DMEMLIM=7024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan pan.c
./pan -m500 -c1
```

Column "States Stored" refers to the number of unique global system states stored in the state-space, while column "States Matched" refers to the number of states that were revisited during the search - see: `spinroot.com/spin/Man/Pan.html#L10`

4.2 XCD and Architecture Realizability

All constraints in XCD are *local*, expressed on local component/role data and parameters. Indeed, components do not even synchronize on message emission – asynchronous channels are used to ensure that they are completely decoupled and independent.

Non-local interaction constraints, like those imposed by the glue in Fig. 2, cannot be expressed in XCD. This ensures that XCD connectors are always realizable in a way that respects the architecture, i.e., without transforming decentralized designs to centralized ones. When non-local interaction constraints are desired, they can be verified as properties of some connector or configuration.

Data themselves are encapsulated either by components or connector roles, so there are no aliasing problems, and concurrency is controlled through component ports. Each port is a concurrent unit (a monitor), thus ensuring that actions of a port are mutually exclusive to each other. As event emission/consumption and method servicing (at provided ports) are atomic, architects need only guarantee (and verify) that method calling (at required ports) will not lead to data race conditions.

5 Formal Verification Analysis

The semantics of XCD described in Section 4 are used to automatically transform XCD architectures into corresponding ProMeLa models, which can be analysed by the Spin model-checker [14]. Each component instance of an architecture becomes a ProMeLa process. Instances of primitive component types follow the patterns described in Fig. 6 and 7. For composite component instances we produce again ProMeLa processes that initiate the processes of their sub-components and establish the channels that these should be using. The transformation to ProMeLa models is done through a tool that is available from the XCD web page [22], along with other case studies and information about the XCD language.

We easily transformed the shared data specification in Section 3 into Promela and analysed the Promela codes using the model checker. The verification results are given

```
role userRole {
 required port pvuser_r {
  int get();
 }
 emitter port pvuser_e {
  set(int data_arg);
 }
}
```

(a) Original user role

```
role userRole {
 bool initialized := false;
 required port pvuser_r {
 @Interaction{ waits: initialized; }
  int get();
 }
 emitter port pvuser_e {
  @Interaction{
   ensures: initialized := true; }
  set(int data_arg);
 }
}
```

(b) Constrained user role

Fig. 8: Constraining role user of connector memory2user specified in Fig. 5

Table 2: Verification results for the constrained user role of Fig. 8

Model Size	State-vector	States		Memory	Time
	(in Bytes)	Stored	Matched	(in MB)	(in sec)
1 user	148	1744	1374	128	0.00
2 users	236	286735	479528	182	0.95
3 users	336	1998023	5594597	662	5.92
4 users	424	20477758	70199771	7024†	81.10

in Table 1. Its verification allowed us to quickly evaluate whether the system components behave compatibly without deadlocking. Although in some cases, the memory may go beyond the required amount for a full verification (indicated with a † in Table 1), designers can still obtain useful information about their system models and increase their confidence in their correctness.

In the rest of this section, we discuss some of the issues that we identified through the formal verification.

5.1 Avoiding Chaotic Behaviour through Connector Protocols

The `memory` component is specified in Listing 3 with an **accepts** guard stating that it will enter chaotic behaviour if it receives a call for method `get` when the data is not yet initialized. This is avoided through the `memory2user` connector that constrains memory such that it does not receive requests for method `get` before the event `set` that initializes its data. Indeed, when we remove this constraint from the memory role of the connector and re-run our verification, an assertion violation error occurs identifying that the memory component has entered a chaotic behaviour.

5.2 Reducing the State Space

When the number of user components in the system configuration becomes more than 2, the state space of the formal model increases and hinders a full verification. Therefore, design errors may be left uncaught. The state space can be reduced by further constraining the possible behaviour of components. To do so, we introduce further interaction

constraints on user components via the user role of the `memory2user` connector. When the user role is modified as shown in Fig. 8, user components cannot make requests for method `get` before they emit event `set`. When we re-run the verification the state space is reduced as shown in Table 2, enabling us to fully verify a system with three users.

6 Conclusions

The XCD ADL supports user-defined, complex connectors, that can recursively use other connectors to model protocols and sub-protocols, in the same way as components can have sub-components. Complex connectors allow architects to increase the modularity of their specifications, and produce component specifications that are agnostic to their usage contexts. This increases the re-usability of component specifications and can help CBSE by permitting the development of general component specifications. It also helps with the reuse of protocol specifications as these can be specified independently of specific usage instances. Finally, it aids architectural exploration, since architects can easily replace protocols and components without having to rewrite their specifications.

Many ADLs have supported connectors so far, with Wright [1] being the first one to provide formal support for them. Unfortunately, the connector structure proposed by Wright, and all those inspired from Wright ever since, permits the specification of unrealizable architectures. We showed how this can occur and presented XCD's approach for avoiding this issue and guaranteeing that connectors will always be realizable.

The paper also presented how XCD uses and extends Design-by-Contract so as to hopefully make it easier for practitioners to use it for specifying the architectures of their systems and for communicating these architectures to others. The transformation of the XCD language constructs was shown with the use of patterns of Dijkstra's guarded commands that can be easily modelled with the Spin model-checker's language ProMeLa.

Preliminary verification results showed promise, though the current tool support needs to be improved. In the future we plan to apply a number of patterns to reduce the state space of the models produced and explore ways to perform other optimizations, e.g., to reduce the state size itself.

References

1. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Trans. Softw. Eng. Methodol. 6(3), 213–249 (1997)
2. Alur, R., Etessami, K., Yannakakis, M.: Inference of message sequence charts. IEEE Trans. Software Eng. 29(7), 623–633 (2003)
3. Alur, R., Etessami, K., Yannakakis, M.: Realizability and verification of MSC graphs. Theor. Comput. Sci. 331(1), 97–114 (2005)
4. Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.): Handbook of Process Algebra. Elsevier (Mar 2001)
5. Bjørner, D., Jones, C.B. (eds.): The Vienna Development Method: The Meta-Language, Lecture Notes in Computer Science, vol. 61. Springer (1978)
6. Canal, C., Pimentel, E., Troya, J.M.: Specification and refinement of dynamic software architectures. In: Donohoe, P. (ed.) WICSA. IFIP Conference Proceedings, vol. 140, pp. 107–126. Kluwer (1999)

7. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In: FMCO'05 – Formal Methods for Comp. and Obj. LNCS, vol. 4111, pp. 342–363. Springer (2006)
8. 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)
9. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM 18(8), 453–457 (1975)
10. Feiler, P.H., Gluch, D.P., Hudak, J.J.: The Architecture Analysis & Design Language (AADL): An Introduction. Tech. rep., Software Engineering Institute (2006)
11. 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)
12. Garlan, D., Shaw, M.: An introduction to software architecture. In: Ambriola, V., Tortora, G. (eds.) Advances in Software Engineering and Knowledge Engineering. pp. 1–39. World Scientific Publishing Company, Singapore (1993), also appears as SCS and SEI technical reports: CMU-CS-94-166, CMU/SEI-94-TR-21, ESC-TR-94-021.
13. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969)
14. Holzmann, G.J.: The SPIN Model Checker - Primer and reference manual. Addison-Wesley (2004)
15. Issarny, V., Bennaceur, A., Bromberg, Y.D.: Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability. In: Bernardo, M., Issarny, V. (eds.) SFM. Lecture Notes in Computer Science, vol. 6659, pp. 217–255. Springer (2011)
16. Ivers, J., Clements, P., Garlan, D., Nord, R., Schmerl, B., Silva, J.R.O.: Documenting component and connector views with UML 2.0. Tech. Rep. CMU/SEI-2004-TR-008, Software Engineering Institute (Carnegie Mellon University) (2004)
17. Luckham, D.C.: Rapide: A language and toolset for simulation of distributed systems by partial orderings of events. Tech. rep., Stanford University, Stanford, CA, USA (1996)
18. Magee, J., Kramer, J.: Dynamic structure in software architectures. In: SIGSOFT FSE. pp. 3–14 (1996)
19. Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A.: What industry needs from architectural languages: A survey. IEEE Trans. Software Eng. 39(6), 869–891 (2013)
20. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. Software Eng. 26(1), 70–93 (2000)
21. Meyer, B.: Applying "Design by Contract". IEEE Computer 25(10), 40–51 (1992)
22. Ozkaya, M.: XCD website. `http://www.soi.city.ac.uk/~abdz276/xcd.html` (2013)
23. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. SIGSOFT Softw. Eng. Notes 17(4), 40–52 (Oct 1992)
24. Plasil, F., Visnovsky, S.: Behavior protocols for software components. IEEE Trans. Software Eng. 28(11), 1056–1076 (2002)
25. Schmidt, H., Poernomo, I., Reussner, R.: Trust-by-contract: Modelling, analysing and predicting behaviour of software architectures. J. Integr. Des. Process Sci. 5(3), 25–51 (Aug 2001)
26. Schreiner, D., Göschka, K.M.: Explicit connectors in component based software engineering for distributed embedded systems. In: Proceedings of the 33rd conference on Current Trends in Theory and Practice of Computer Science. pp. 923–934. SOFSEM '07, Springer-Verlag, Berlin, Heidelberg (2007)
27. Tripakis, S.: Undecidable problems of decentralized observation and control. In: Proc. of the 40th IEEE Conf. on Decision and Control. vol. 5, pp. 4104–4109. IEEE, Orlando, FL, USA (Dec 2001)
28. Tripakis, S.: Undecidable problems of decentralized observation and control on regular languages. Inf. Process. Lett. 90(1), 21–28 (2004)