Design-by-Contract for Reusable Components and Realizable Architectures

Mert Ozkaya
City University London
School of Informatics
London EC1V 0HB, U.K.
mert.ozkaya.1@city.ac.uk

Christos Kloukinas
City University London
School of Informatics
London EC1V 0HB, U.K.
c.kloukinas@city.ac.uk

ABSTRACT

Architectural connectors can increase the modularity and reusability benefits of Component-based Software Engineering, as they allow one to specify the *general* case of an interaction pattern and reuse it from then on. At the same time they enable components to be *protocol-independent* – components do not need to know under which interaction patterns they will be used, as long as their minimal, local interaction constraints are satisfied. Without connectors one can specify only *specific* instances of such patterns and components need to specify themselves the interaction protocols that they will follow, thus reducing their reusability.

Connector frameworks so far allow designers to specify systems that are unrealizable in a decentralized manner, as they allow designers to impose global interaction constraints. These frameworks either ignore the realizability problem altogether, ignore connector behaviour when generating code, or introduce a centralized controller that enforces these global constraints but does so at the price of invalidating any decentralized properties of the architecture.

We show how the XCD ADL extends Design-by-Contract (DbC) for specifying *(i)* protocol-independent components, and *(ii)* arbitrary connectors that are always realizable in a decentralized manner as specified by an architecture – XCD connectors impose local constraints only. Use of DbC will hopefully make it easier for practitioners to use the language, compared to languages using process algebras. We show how XCD specifications can be translated to ProMeLa so as to verify that *(i)* provided services local interaction constraints are satisfied, *(ii)* provided services functional pre-conditions are complete, *(iii)* there are no race-conditions, *(iv)* event buffer sizes suffice, and *(v)* there is no global deadlock. Without formally analyzable architectures errors can remain undiscovered for a long time and cost too much to repair.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Languages*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Programming by contract*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Pre- and post-conditions*

CBSE'14, June 30–July 4, 2014, Marcq-en-Baroeul, France.

ACM 978-1-4503-2577-6/14/06 ...$15.00.
http://dx.doi.org/10.1145/2602458.2602463.

Keywords

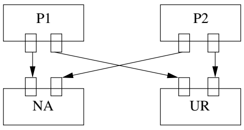
Modular specifications; Separation of functional and interaction behaviours; Connector realizability.

1. INTRODUCTION

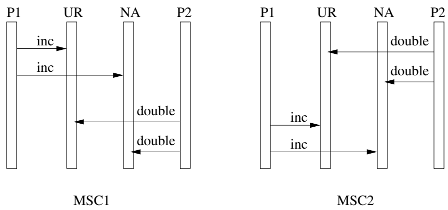
Component-based Software Engineering helps develop software systems out of largely reusable components, thus reducing development time and cost, and leading to a higher system quality. Reusable components end up having fewer design and implementation errors, as these are identified and corrected through their use by different systems. Researchers in software architectures [36, 19] have identified connectors as another important element for increasing modularity and reusability even further. Connectors allow the specification of arbitrary interaction patterns, thus allowing such patterns to be reused. At the same time, components no longer need to specify instances of such patterns themselves, thus increasing component reusability too. Indeed, designers can more easily explore alternative designs/protocols to meet the requirements of their specific system when components are separated from the possible interaction patterns (i.e., connectors) that they can be used with. This is similar to how we program in languages such as C++. We define a vector class (re-sizable array), specifying its basic operations and the minimal, local constraints on its use, e.g., that the vector should not be empty when retrieving an element. The vector does not specify anything about reverse, sort, etc. to be more reusable. These are instead specified by independent algorithms, among which one selects the most appropriate to their context, e.g., bubble or merge sort. Keeping the two separate increases the code modularity and reusability. Our data-structures/classes stay independent of specific usage patterns, which are described separately as algorithms. Indeed, the reusability of the algorithms themselves increases as well, as they can usually be applied at different classes. Specifying component becomes harder without support for connectors and sometimes specifiers avoid specifying the interaction patterns altogether, which leads to the architectural mismatch problem [17, 18].

1.1 Connector Realizability

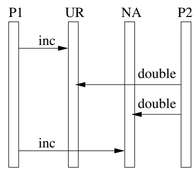
A formal framework for specifying connectors in the Wright language was presented in the seminal work of Allen and Garlan [4] and has been followed by almost all approaches that support connectors – a set of protocol role behaviours, that component participants should implement, and a "glue" element that choreographs them. However, connectors are not supported in the main languages used by practitioners [26], who complain about the complexity of ADLs (an orthogonal issue). This may have been a blessing in disguise, since the ADLs supporting connectors do so in a manner that is somewhat dangerous for general usage. This is because, following Wright [4], these languages allow architects to specify connectors that are potentially *unrealizable* in a distributed manner



(a) A decentralized architecture



(b) A nuclear power plant's (unrealizable) MSCs [5]



(c) An unavoidable bad behaviour [5]

$$
\begin{array}{l}
\textbf{connector}\ \texttt{Plant\_Connector} = \\
\textbf{role}\ \texttt{P1} = \overline{\texttt{ur}} \rightarrow \overline{\texttt{na}} \rightarrow \texttt{P1}. \\
\textbf{role}\ \texttt{P2} = \overline{\texttt{ur}} \rightarrow \overline{\texttt{na}} \rightarrow \texttt{P2}. \\
\textbf{role}\ \texttt{UR} = \texttt{inc} \rightarrow \texttt{UR} \,\Box\, \texttt{double} \rightarrow \texttt{UR}. \\
\textbf{role}\ \texttt{NA} = \texttt{inc} \rightarrow \texttt{NA} \,\Box\, \texttt{double} \rightarrow \texttt{NA}. \\
\textbf{glue} = \texttt{P1.ur} \rightarrow \texttt{UR.}\overline{\texttt{inc}} \rightarrow \texttt{P1.na} \rightarrow \texttt{NA.}\overline{\texttt{inc}} \\
\qquad \rightarrow \texttt{P2.ur} \rightarrow \texttt{UR.}\overline{\texttt{double}} \rightarrow \texttt{P2.na} \rightarrow \texttt{NA.}\overline{\texttt{double}} \rightarrow \textbf{glue} \\
\quad \sqcap\ \texttt{P2.ur} \rightarrow \texttt{UR.}\overline{\texttt{double}} \rightarrow \texttt{P2.na} \rightarrow \texttt{NA.}\overline{\texttt{double}} \\
\qquad \rightarrow \texttt{P1.ur} \rightarrow \texttt{UR.}\overline{\texttt{inc}} \rightarrow \texttt{P1.na} \rightarrow \texttt{NA.}\overline{\texttt{inc}} \rightarrow \textbf{glue}.
\end{array}
$$

(d) Wright's *(unrealizable)* connector for Alur's plant of (b)
Note: barred actions are initiated by the current process, → is the action sequence operator, and □/⊓ are external/internal choice.

Figure 1: An unrealizable protocol/connector

[33]. Realizability is defined as: "a set of MSCs *[i.e., a glue]* [is] realizable if there exist concurrent automata *[the connector roles]* which implement precisely the MSCs it *[the glue]* contains." [5]

Consider the nuclear power plant case study [5], shown in Figure 1a. In the plant, the quantities of Uranium (UR) and Nitric Acid (NA) need to be the same at all times. Two processes P1 and P2 respectively increase and double these quantities and to ensure the plant's safety they need to strictly follow the protocol described by the message sequence charts of Figure 1b. However the protocol in Figure 1b was proved to be unrealizable in a decentralized manner, since bad behaviours like in Figure 1c cannot be avoided [5].

One can check conditions implying a protocol's realizability [6, 9], attempt to identify implied scenarios from the protocol [43], or even attempt to repair it [23] by multi-casting messages to more recipients. However, there will always be cases where the protocol cannot be realized. Worse yet, there are cases where it cannot be decided whether a protocol is realizable in a distributed manner with only the specified roles or not – the general problem is undecidable [6] and relates to the undecidability of decentralized observation and control [41]. Connectors can use their "glue" to impose non-local interaction constraints on the participating components, just like service choreographies do. Such global interaction constraints cannot be realized always by the participating components, since the global system state is not always known. Nevertheless, such unrealizable protocols are very easy to specify in existing ADLs. Indeed, Figure 1d shows the Wright [4] connector specification of the unrealizable protocol of Figure 1b. It shows the four participating roles (`P1`, `P2`, `UR`, and `NA`), and the `glue` part of the connector. The glue element links role actions together (e.g., `P1.ur->UR.`$\overline{\texttt{inc}}$), establishing the communication channels between component ports. Unfortunately, the glue also imposes global interaction constraints – here that roles `UR` and `NA` follow the behaviour `inc`→`inc`→`double`→`double` ⊓ `double`→`double`→`inc`→`inc`. While linking component actions together does not create any realizability problems, global interaction constraints allow architects to present unrealizable specifications as architectural solutions. While a requirements language needs to be able to express something potentially unrealizable (as it is a wish), we believe that an ADL needs to be able to specify only realizable designs, as these are supposed to be solutions for the requirements: wishing for a building that is suspended in the air is acceptable but presenting a drawing of such a building as an architectural solution is not, unless it is made explicit how this can be achieved (builders cannot "refine" the architecture).

Some approaches follow Wright [4] in supporting connectors with a glue element but ignore the connector behavioural specification when generating code. This is for example the case with SOFA [13] and its code generation ConGen [16, 12]: "we are rather interested in rich functionality than formal proving that a connector has specific properties; thus, at this point we do not associate any formal behavior with a connector." [12, p. 14]. Without associated behaviour one cannot generate code for arbitrary, user-defined protocols. Only simple connectors like procedure call can be supported, which forces one to specify protocols inside components.

Finally, a third approach implements arbitrary, user-defined connectors by introducing additional centralized controllers for connectors. In Exogenous Connectors [22] these controllers are explicit and clearly visible – while this centralizes all behaviour, it avoids surprises. In BIP [8], an underlying distributed consensus protocol is employed instead, so that connector participants can know the exact global system state – essentially adding an implicit centralized controller. However, network overhead, reliability, scalability, etc. analyses (what practitioners *really* care about [26]) based on the decentralized architectural design are now invalid. BIP's implicit centralization changes the system structure and its behaviour with respect to these properties – it breaks what ArchJava calls "communication integrity" [1]. After all, if the architect wished for a centralized solution they should have specified it explicitly by introducing a controller component in the system in Figure 1d – that is the solution at the architectural level for the requirement. If they did not do so it was probably because they desired a decentralized solution, so as to get its benefits. But such a decentralized solution must be shown to satisfy the requirement, not simply repeat it, as the glue does in this specification.

1.2 Paper Contribution and Structure

Herein we present XCD, a formal ADL that, following Wright [4], supports arbitrary, user-defined, connectors. Unlike Wright and all other ADLs following it, XCD allows only local constraints to be defined in connectors, so as to ensure realizability by definition. Non-local interaction constraints are now only expressible as properties the architecture should satisfy. Our work builds on our earlier attempts at such an ADL [21, 32], and using FSP [25] to specify and verify architectures [35]. The differences from these are the following: *(i)* We have simplified the main notions, no longer having "control strategies"; strategies are connectors. *(ii)* We have extended the language to better support architects with: data arrays, enumerated types, interval values, helper functions, asynchronous interaction, and composite components that were not supported in our initial FSP encoding and tool. *(iii)* We have also replaced FSP with Spin's ProMeLa language [20], as encoding asynchronous interaction and method/event parameters in FSP required too much effort. Spin's code availability also helped us in better understanding the use of some constructs and optimizing our models.

XCD tries to resemble a programming language and follows a Design-by-Contract (DbC) based approach, as practitioners find process algebra-based ADLs to have a "steep learning curve" [26].

A brief, high-level introduction of the current version of the XCD language was presented in an earlier short position paper [34]. This

```
1 SimpleCType:component ID^CTypeName ( [DataType ID^ParamName]* )
2   { Variable* Port+ } ;
3
4 Port: RequiredPort | ProvidedPort ;
5 ProvidedPort: provided ID^PortName
6   { ProvidedPortMethod+ } ;
7 RequiredPort: required ID^PortName
8   { RequiredPortMethod+ } ;
9 ProvidedPortMethod:
10   [ ProtocolAwaits | ProtocolAccepts ]?
11       FunctionalReqEns? MethodSignature ;
12 RequiredPortMethod:
13   ProtocolAwaits?
14       FunctionalPromReqEns? MethodSignature ;
15
16 ProtocolAwaits:  @interaction { waits   Expression } ;
17 ProtocolAccepts: @interaction { accepts Expression } ;
18
19 FunctionalReqEns:
20   @functional { ReqEns [ otherwise ReqEns ]* };
21 ReqEns: [ requires Expression ]? ensures Assignments
22       | requires Expression ;
23 FunctionalPromReqEns:
24   @functional { PromReqEns [otherwise PromReqEns]* };
25 PromReqEns: [ promises Assignments ]? ReqEns
26           | promises Assignments ;
27
28 Variable: DataType ID^VarName := Expression ;
29 DataType: bool | byte | short | int | ID^DataTypeName ;
30 Assignments: Assignment [ Assignment ]* ;
31 Assignment: ID^VarName := Expression ;
32       | ID^VarName in '[' Expression , Expression ']' ;
```

Note: Rules are of the form `symbol: expression;`. Keywords are in bold, "`(){}=,`" are part of the input, "`[]`" are used for grouping (unless quoted), and "`?+*`" stand for optional, at least once, and zero or more repetitions respectively. Superscripts refer to the meaning of an `ID`, e.g., `ID`$^{\text{CTypeName}}$ is a component type name.

Figure 2: Simple components (`SimpleCType`) grammar

paper describes in detail the XCD notions, its grammar, and the language mappings to Spin's ProMeLa language, so as to enable formal verification of architectural designs. It identifies the five properties that can be verified without any further input from designers, and shows how designers can modify the ProMeLa models to verify more properties. It demonstrates most of the new features (enums in Figure 6 line 1, helper functions in Figure 8 lines 55-58) and discusses others (intervals in section 2.1). It shows how global constraints can be supported by an *explicit* centralized controller component when decentralized control is impossible. The paper also includes an extensive experimental evaluation using a number of well-known architectural case studies (all available at [40]), and some further related work before the final discussion.

2. CONTRACTS FOR ARCHITECTURES

In XCD we follow a Design-by-Contract (DbC) [28] approach to specify the behaviours of components, extending it in two ways so as to support software component frameworks like CORBA [30] and OSGi [31] better. We extend DbC so as to be able to specify contracts not only for the component provided services but for its required services too. This is because, unlike object classes for which DbC was initially designed, components also have required services in their public interfaces. At the same time, we propose a different contract structure so as to better distinguish between the functional and interaction component constraints, which are usually mixed together in most DbC approaches. Finally, we use DbC to specify connectors/protocols as well as components.

2.1 Structure of Simple Components

A simple (non-composite) component has data variables and a set of ports for interacting with its environment. We ignore ports supporting events due to lack of space. Each port can be either a

```
component Thread {
 bool started := false; // component data.
 bool died := false;

 provided p {
  @interaction   { accepts: ! started; }
  @functional { ensures: started := true;}
  void start();

  @functional {
    ensures: \result := started && ! died; }
  bool isAlive();

  @interaction { waits: died; }
  void join();
  // ... other methods
 };
};
```

Figure 3: Java Thread as an XCD component

provided one, offering a number of methods to the environment, or a required one, which uses methods provided by the environment. XCD component ports execute concurrently to each other and operate as a monitor, i.e., at most one method of a port can be active at any time. Interaction between ports is asynchronous, as we target mainstream software components. Figure 2 shows the high-level grammar for simple components, abstracting over a number of language details, e.g., helper functions, for simplicity. Figure 3 shows a small component example, described in more detail later.

As aforementioned, provided port methods (`ProvidedPortMethod`, at line 9) resemble object methods and their constraints can essentially be described through classic DbC. Ignoring the interaction contract, whenever a method is called and the method precondition (**requires** of `FunctionalReqEns`, at line 21) on the parameter and component data values is satisfied, the method post-condition (**ensures**) should be satisfied as well. It should be noted that while pre-conditions are expressions, post-conditions in XCD are in fact assignments. In assignments (lines 31-32) we also allow a variable to be set to a value within a range, for non-deterministic specifications. The use of assignments instead of post-conditions is to make our models easier to formally analyze. Trying to ensure a post-condition like $0 \le x+y+z \le n$ means that we need to consider all possible combinations of x,y,z within the range $[0,n]$, i.e., $(n+1)^3$ states. Instead, architects write this as $x \in [0,n]; y \in [0,n-x]; z \in [0,n-x-y]$;, which has $(n+1)(n^2+5n+6)/6$ states [1]. For $n = 255$, i.e., a byte, we need explore 2.8 M instead of 16.7 M states. A provided port method is atomic – testing its required pre-condition and performing its ensured assignment is done as one action.

Required port methods (`RequiredPortMethod`, at lines 12-14) do not have an equivalent in object class definitions and, as such, classic DbC does not consider them. These are actions that the component enacts itself, instead of actions that it reacts to (in its provided ports). A restaurant may provide a service between 7pm and 11pm (protocol) and offer an Italian menu (functional). Symmetrically, a customer may require a service between 9pm and 10pm (protocol) and desire to have a pizza (functional). A required port method is non-atomic (race-conditions are considered later). At the first state it selects parameter values (i.e., affects its **promises** at lines 25-26) and makes the method call. At the second state it receives the method call results and updates the component data, according to the required/ensures pair establishing appropriate assignments given conditions on the component data and the method results.

2.1.1 Functional and Interaction Contracts

As shown in Figure 2, along with functional contracts methods in XCD can have protocol contracts too. The latter can be of a

[1] Wolfram Alpha: `https://www.wolframalpha.com/input/?i=sum_x=0^n+sum_y=0^(n-x)+sum_z=0^(n-x-y)+1,n=255`

waits (line 16) or an **accepts** type (line 17). Provided port methods can use either type. The former indicates that the action will be delayed until some predicate on the component data and the method parameters is satisfied. The latter indicates that the action will be processed immediately when received and either it will be accepted or it will be rejected – whereby rejection leads to chaotic behaviour (caught as a violated assertion in our models). So a data queue may use a waits constraint to specify that a request for an element will be delayed till the queue is not empty. Alternatively, an object lock can use an accepts constraint to specify that attempts to unlock it cause undefined behaviour when it is already unlocked.

Examples of such protocol contracts abound in everyday life. A washing machine manufacturer can warn users against opening the door while the machine is operating (**accepts**: `! operating`) or add a safety mechanism that delays the door opening (**waits**: `! operating`). The former protocol contract makes no guarantees whatsoever if someone attempts to open the door during operation – water may be spilt outside and the user can even be electrocuted because of it. In fact, such bad behaviour due to a component's protocol contract violation appears in the standard libraries of mainstream languages already. In Java, `RuntimeExceptions` are used extensively to represent such situations. Unlike other exceptions, they are not supposed to be caught by code. In fact, they are not even supposed to be declared by the methods that may throw them – Java calls them "unchecked exceptions". The method `Thread.start()` can throw such an exception when called on a thread that has already started. Using XCD protocol contracts this can be specified as in line 6 of Figure 3. Note that a method may have no protocol contract, e.g., `isAlive` (lines 10-12). Sometimes it may have no functional contract instead, like `join` that can be specified entirely through a protocol contract (lines 14-15).

Another example of protocol contract violations in Java is `SocketException`, thrown when a socket's `setSocketFactory` is called more than once. Exception `InternalError` as well, thrown by `wait`/`notify` when the thread is not the current owner of the object's monitor. And of course, a `NullPointerException`, which is thrown when an object reference has not been initialized properly. All these are examples of erroneous protocol usage. All of them terminate a program immediately. By introducing the separate protocol contract (**@interaction**) construct, such interface protocols become easier to express and their importance is highlighted. Functional contracts also become easier to express. Indeed, in the functional contract of method `start` at lines 6-8 of Figure 3, the requires clause does not consider the state of variable `started`. It assumes that the call has already been accepted, at which point it has no functional constraint to impose. It should be noted that component protocol contracts do not modify the component state – there is no ensures clause in them. State updates in XCD components are instead the sole responsibility of functional contracts, so as to keep contracts simpler.

User obligations. When a required port `r` makes a request on a provided port `p`, it needs to ensure that `p.`**accepts** is satisfied, if the provided method has an accepts protocol (so `p.`**waits** is **true**), otherwise (`p.`**accepts** being **true**) that `p.`**waits** is satisfied. So in general:

$$(\texttt{r.}\mathbf{waits} \rightarrow \texttt{r.}\mathbf{promises}) \rightarrow (\texttt{p.}\mathbf{waits} \rightarrow \texttt{p.}\mathbf{accepts})$$

Interestingly, the user does not need to satisfy the functional requirements of the provided service (`p.`**requires**), since these must be complete when the service's interaction constraint is satisfied – the call has been accepted already, so it must be honoured.

Simple component types define the data a component has and its ports with their methods and protocol/functional contracts. However in order to produce formal models of the component *instances* we need to consider also the protocol/connector roles these are assuming within an architecture, as roles constrain their behaviour.

```
1 XType:connector ID^XTypeName ( XTypeParam+ )
2   { Role+ XInstance+ } ;
3 XTypeParam: ID^RoleName { ID^PortVarName+ }
4            | DataType ID^ParamName;
5
6 Role:role ID^RoleName
7   { Variable+ [[required|provided] PortVar]+ } ;
8
9 PortVar: ID^PortVarName
10   { [ [XProtocol]? MethodSignature]+ } ;
11 XProtocol: @interaction {
12    [ waits Expression ]? ensures Assignments
13    | waits Expression } ;
14
15 XInstance: ID^XTypeName ID^XInstanceName ( XInstanceArg+ );
16 XInstanceArg: ID^RoleName { ID^PortVarName+ } | Expression ;
```

Figure 4: High-level XCD grammar – Connectors (`XType`)

While a component type may have just accepts conditions, its instances may also get waits conditions from their roles.

2.2 Connector/Protocol Structure

As shown in Figure 4, XCD connectors have a set of roles (each assumed by some component) and instances of other connectors that they are using. A basic connector is provided by the language to specify a simple asynchronous method call, linking the required port of one component to the provided port of another, without imposing any further constraints on their actions. There is no glue element in XCD connectors, nor any other way to specify global state or constraints – everything is *local* and so *directly realizable*. Each role consists of role data, that keep track of the protocol's local state, and a set of provided/required port variables, to be assumed by the role component's ports. Role port variables have actions like component ports do. These are the actions that the role requires its component to have and that the role will constrain. The behaviour of port variable actions is again specified through contracts, only now all contracts have the same form, i.e., a pair of a **waits** precondition and an **ensures** assignment, as shown in lines 11-13. This is because roles can only delay some component port action, until the point where it is acceptable by the protocol/connector they are a part of. Role actions have no functional contracts, as they cannot influence the component's action parameters, or its result or the manipulation of the component's private data. Instead, the protocol contracts of role actions use their **ensures** assignments, to update the role's local protocol state after the action.

A component instance is provided with all the roles it assumes in an architecture, just like actors are provided with the roles and corresponding scripts they play in a movie. Component instances use the role(s) port method contracts to further constrain their own contracts and are responsible for updating the role variables along with their own. Here again we diverge from Wright [4]. In Wright, components should refine/implement the roles they assume; the final system is the composition of components and connector glues only – roles are ignored. This restricts the reusability of components – they need to know beforehand all protocols under which they may be used, something that one would never require of actors. Instead, in XCD components do not need to refine their roles. On the contrary, their behaviour can be much richer. For this to work, XCD components need to be presented with their role constraints – XCD components are *interpreters* of connector roles.

2.2.1 A Centralized Nuclear Plant Xcd Connector

Figures 6–9 specify a centralized XCD connector that ensures the required glue property of the nuclear plant example in Figure 1 – the architecture is shown in Figure 5. The glue property states that UR and NA should always increase and double their quantities in tandem: `UR.i->NA.i->UR.d->NA.d|UR.d->NA.d->UR.i->NA.i`, where `i`

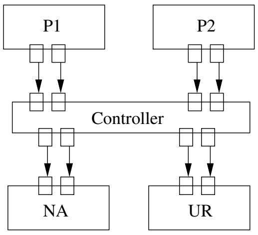


Figure 5: A centralized architecture

```
1 enum order := {none, incFirst, dblFirst}; // New type.
2
3 connector centralised(roleP1{toUR, toNA},
4                       roleP2{toUR, toNA},
5                       roleUR{inc,  double},
6                       roleNA{inc,  double},
7 /*extra role*/  roleController{P1toUR,   P1toNA,
8                             P2toUR,   P2toNA,
9                             CtoURinc, CtoURdouble,
10                            CtoNAinc, CtoNAdouble}) {
// P1/UR/NA/P2 and Controller from Fig. 7 and Fig.8-9 respectively.

134   // Controller appears to P1 & P2 as UR & NA.
135   connector async link1(roleP1{toUR},
136                         roleController{P1toUR});
137   connector async link2(roleP1{toNA},
138                         roleController{P1toNA});
139   connector async link3(roleP2{toUR},
140                         roleController{P2toUR});
141   connector async link4(roleP2{toNA},
142                         roleController{P2toNA});
143   // Controller appears to UR & NA as P1 & P2.
144   connector async link5(roleUR{inc},
145                         roleController{CtoURinc});
146   connector async link6(roleUR{double},
147                         roleController{CtoURdouble});
148   connector async link7(roleNA{inc},
149                         roleController{CtoNAinc});
150   connector async link8(roleNA{double},
151                         roleController{CtoNAdouble});
152 }
```

Figure 6: Centralized nuclear plant connector in XCD

```
12   role roleP1 {
13     bool urFirst:=false;
14     required port_variable toUR {
15       @interaction{ waits: !urFirst;
16             ensures: urFirst := true; }
17       void incUR(); }
18     required port_variable toNA {
19       @interaction{ waits: urFirst;
20             ensures: urFirst := false; }
21       void incNA(); }
22   }
23   role roleUR {
24     provided port_variable inc { void incUR(); }
25     provided port_variable double { void doubleUR(); }
26   }
```

Figure 7: Original nuclear plant roles in XCD (part of Figure 6)

and d are the increase and double actions. The connector employs five roles instead of the four roles in the decentralized connector, as it has an explicit centralized controller (lines 7-10). Without a controller it is impossible to ensure the glue property (indeed, the decentralized connector violates it). Figure 7 shows the P1 and UR roles of the decentralized connector (omitted roles P2 and NA are similar). These roles behave as in the Wright specification of Figure 1d. Roles UR (lines 23-26) and NA have no constraints, as they can receive requests to increase or double their amount of fuel anytime. Roles P1 (lines 12-22) and P2 impose that increase/doubling requests are sent first to UR and then to NA. The controller role, shown in Figure 8 and Figure 9, presents itself as UR and NA to P1 and P2 using its provided ports (lines 60-71 for

```
43  role roleController {
44    order corder := none;
45    bool p1_incNARcvd :=false;
46    bool p1_incURRcvd :=false;
47    bool p2_dblNARcvd :=false;
48    bool p2_dblURRcvd :=false;
49
50    bool ur_incUREmtd := false;
51    bool na_incNAEmtd := false;
52    bool ur_dblUREmtd := false;
53    bool na_dblNAEmtd := false;
54
55 all_received(){return p1_incURRcvd && p1_incNARcvd
56 /*helper functions*/ && p2_dblURRcvd && p2_dblNARcvd;}
57 inc_emitted(){return ur_incUREmtd && na_incNAEmtd;}
58 dbl_emitted(){return ur_dblUREmtd && na_dblNAEmtd;}
59
60    provided port_variable P1toUR {
61      @interaction{
62      waits: !p1_incURRcvd;
63      ensures: p1_incURRcvd :=true;
64        corder := pre(corder) == none
65          ? incFirst : pre(corder); }
66      void incUR(); }
67    provided port_variable P1toNA {
68      @interaction{
69      waits:  !p1_incNARcvd;
70      ensures: p1_incNARcvd :=true; }
71      void incNA(); }
```

Figure 8: Nuclear plant controller role in XCD – provided ports

```
85    required port_variable CtoURinc {
86      @interaction{
87      waits: all_received() && !ur_incUREmtd
88        && ( (corder==incFirst)
89          || (corder==dblFirst && dbl_emitted()) );
90      ensures: ur_incUREmtd := true; }
91      void incUR(); }
92    required port_variable CtoNAinc {
93      @interaction{
94      waits: ur_incUREmtd && !na_incNAEmtd;
95      ensures: // clear flags if dblFirst
96        p1_incURRcvd := !(pre(corder) == dblFirst);
97        p1_incNARcvd := !(pre(corder) == dblFirst);
98        ur_incUREmtd := !(pre(corder) == dblFirst);
99        na_incNAEmtd := !(pre(corder) == dblFirst);
100       p2_dblURRcvd := !(pre(corder) == dblFirst);
101       p2_dblNARcvd := !(pre(corder) == dblFirst);
102       ur_dblUREmtd := pre(corder) == dblFirst
103         ? false : pre(ur_dblUREmtd);
104       na_dblNAEmtd := pre(corder) == dblFirst
105         ? false : pre(na_dblNAEmtd);
106       corder := pre(corder) == dblFirst
107         ? none : pre(corder); }
108     void incNA(); }
```

Figure 9: Nuclear plant controller role in XCD – required ports

ports related to increase). Using its required ports, it presents itself as P1 and P2 to UR and NA (lines 85-108 for ports related to increase). The provided ports note which commands have been received by P1 and P2, and which of increase or double was received first in each round, using the `corder` variable (an enumerated type). The expression on lines 64-65 uses the *if-then-else* operator "`guard ? exp1 : exp2`", and the operator `pre` to access the value of variable `corder` when the action started. Once all commands have been received, the required ports in Figure 9 start requesting from UR and NA to update their fuel amounts. This behaviour uses helper functions `all_received`, `inc_emitted`, and `dbl_emitted` (defined in Figure 8, lines 55-58). Depending on whether it was the increments or the doubles that were received last, action `incNA` (or `doubleNA` respectively) reset all role variables, to enable the next round. The full models for both the decentralized and centralized protocols are available at the XCD website [40].

Compared to the Wright connector in Figure 1d, the XCD connector is longer – much more so. This is for two main reasons.

```
CompositeCType:component ID^CTypeName (
                              [DataType ID^ParName]* )
                { [ CInstance | XInstance ]+ } ;
CInstance: ID^CTypeName ID^CInstanceName ( Expression* ) ;
```

Figure 10: High-level XCD grammar – Composite components (CompositeCType)

```
component NuclearPlant() {
  component P1 p1inst();  component P2 p2inst();
  component NA nainst();  component UR urinst();
  component controller controllerinst();

  connector centralised  centrins(
         p1inst{incUR,    incNA},
         p2inst{doubleUR, doubleNA},
         urinst{incUR,    doubleUR},
         nainst{incNA,    doubleNA},
         controllerinst{P1_incUR,    P1_incNA,
                        P2_doubleUR, P2_doubleNA,
                        UR_incUR,    UR_doubleUR,
                        NA_incNA,    NA_doubleNA}
         );
}
```

Figure 11: Nuclear plant composite component in XCD

Firstly, it does not employ a process algebra but uses a language similar to a programming one, e.g., Java, which is more verbose but also more familiar. Secondly, and more importantly, the XCD connector specifies a *solution*. Indeed, it does not simply repeat the requirement about the behaviour of the UR and NA roles but it guarantees it. It should be noted that this solution increases the number of messages per round, from four to eight. It also changes the structure of the system – if one of P1 or P2 fails, no interactions are possible any more, unlike in the original architecture. Both the number of messages and system structure are crucial for a proper architectural system analysis. Lower-level designs should not modify them, since then the architecture is compromised – what ArchJava calls (lack of) "communication integrity" [1]. XCD aims at facilitating the expression of architectures that can be realized without compromising their communication integrity. If a Wright connector is realizable then XCD can also represent it.

2.3 Structure of Composite Xcd Components

The grammar for specifying composite components is shown in Figure 10. A composite component declares a set of component instances (which can be either simple or composite) and a set of connector instances. The connector instances are initialized with the component instances that will assume their roles. In this way, a composite component defines the configuration of its sub-components. The ports of sub-components that are not connected through the connectors employed in the composite component become ports of the composite component. An architecture is a composite component where all the ports of the sub-components are connected, as the composite component in Figure 11.

3. TRANSLATING XCD INTO PROMELA

We translate XCD models into Spin's ProMeLa [20], in order to formally verify architectures. Each component *instance* becomes a separate ProMeLa process. The number of component instances is fixed in each architecture (we consider only static architectures). ProMeLa processes are concurrent automata that are composed together through synchronous or buffered asynchronous channels. We use asynchronous channels in our models, as we target software systems, where asynchronous interaction is the mainstream. For each simple component (c) instance's (i) provided port (p) we introduce one asynchronous channel ($ch_p^{c_i}$), with a buffer size equal to the number $N = \text{connectedTo}(c_i, p)$ of required ports that are

```
1  SimpleComponent2Promela(SimpleCInstance comp)
2
3  FORALL port ∈ comp.SimpleCType.ProvidedPortSet
4    // Requests
5    chan cReq.comp.port = [port.Connections] ...
6    // Responses
7    chan cRes.comp.port = [1] ...
8
9  proctype comp.InstanceID (params...) {
10  LET
11    RoleVars = { role.VarSet | role ∈ comp.RoleSet };
12      VarSet = RoleVars ⋃ comp.SimpleCType.VarSet;
13    RoleReqPorts = { r ∈ role.RequiredPortSet
14        | role ∈ comp.RoleSet };
15    RoleReqMethods = { m ∈ r.Methods
16        | r ∈ RoleReqPorts };
17    RoleVarsRace = { v ∈ m.Methods.Ensures.VarSet
18        | m ∈ RoleReqMethods };
19    compReqPorts = { r
20        | r ∈ comp.SimpleCType.RequiredPortSet};
21    compReqMethods = { m ∈ r.Methods
22        | r ∈ compReqPorts };
23    compVarsRace = { v ∈ m.Methods.Ensures.VarSet
24        | m ∈ compReqMethods };
25      VarSetRace = RoleVarsRace ⋃ compVarsRace;
26  IN
27  FORALL var ∈ VarSet
28    var.DataType var.Pre_State  = var.InitialValue;
29    var.DataType var.Post_State = var.InitialValue;
30  FORALL var ∈ VarSetRace
31    var.DataType var.Pre_State_Copy=var.InitialValue;
32  FORALL port ∈ comp.SimpleCType.RequiredPortSet
33    short port.Lock=0; // One lock per required port.
34
35  Start:
36  do
37   FORALL port ∈ comp.SimpleCType.ProvidedPortSet
38     Port2Promela_Provided(comp, port);
39   FORALL port ∈ comp.SimpleCType.RequiredPortSet
40     Port2Promela_Required(comp, port);
41  od
42 }
```

Figure 12: Translating a simple component instance to ProMeLa

connected to port p of component instance c_i, as in lines 4-5 of Figure 12. This is because in the worst case there will be N concurrent service requests to port p from these N required ports. No more service requests can be initiated by them, as required component ports, just like provided ports, act as a monitor and therefore allow at most one method request to be active each time. We also introduce a channel to carry the response back to the required port (lines 6-7). Due to lack of space we omit here the discussion of XCD support for events (emitter and consumer ports) or for non-atomic provided methods, which are needed when a provided method has to call another method to obtain partial results.

3.1 Translating Simple Component Instances

Figure 12 shows the top-structure of the ProMeLa translation for a simple XCD component instance. The translation goes through the instance's assumed roles, collecting their variables and noting which ones of them are used in **ensures** clauses in methods of required port variables (to check for race-conditions later). It declares corresponding variables for each variable of the component and its roles. It then translates the provided and required ports themselves. All port actions are inside a do/od loop of guarded actions [14].

Each component and role data var is mapped to two variables (lines 27-29 of Figure 12). The first one (var.Pre_State on line 28) is the current data value, i.e., the value right before a call, used to evaluate the protocol constraints and the pre-conditions. The second one (var.Post_State) is the data value immediately after a call, i.e., where we have just established the post-conditions. The two variables are needed because an assignment of some $\texttt{var}_i\texttt{.Post\_State}$ may refer to some $\texttt{var}_j\texttt{.Pre\_State}$ values.

```
1 Port2Promela_Provided(SimpleCInstance comp, Port port)
2  FORALL method ∈ port.MethodSet
3  LET
4   roles = { method.roleMethod(r) | r ∈ port.RoleSet };
5   roleAwaits = { r.method.Awaits | r ∈ roles };
6   rolePostEnsures = { r.method.Ensures | r ∈ roles };
7
8   compPos = roleAwaits ⋀ method.Awaits
9             ⋀ method.Accepts;
10  compNeg = roleAwaits ⋀ ¬ method.Accepts;
11
12  compFCReq = method.FCRequiresEnsures.Requires ;
13  compFCEns = {method.FCRequiresEnsures.Ensures}
14               ⋃ rolePostEnsures;
15 IN
16 ::atomic {
17    port.Channel_req ? method.Args : compPos ->
18       assert(compFCReq); // Ensure functional completeness
19       calcData(comp.SimpleCType.VarSet, compFCReq,
20                compFCEns);
21       port.Channel_res ! method.Args;
22    }
23 ::atomic {
24    port.Channel_req ? method.Args : compNeg ->
25       assert(! compNeg); // Request rejected - CHAOS
26    }
```

Figure 13: Translating a provided port to ProMeLa

In order to identify race-conditions that may arise due to the non-atomicity of required method requests, we also introduce another variable `var.Pre_State_Copy` for each data `var` appearing in an **ensures** clause (lines 30-31). This variable keeps a copy of the data's pre-value (`var.Pre_State`) at the point the request was started at the port. For required port methods, we have that `var.Pre_State` = `var.Pre_State_Copy` before and immediately after enacting the method request. But when the response is received we may find that `var.Pre_State` ≠ `var.Pre_State_Copy`, because some other component port has modified `var.Pre_State` (the current variable value) in between. This is a write-read race when a post-condition attempts to use the value $\mathtt{var}_i$`.Pre_State` to establish the value of some $\mathtt{var}_j$`.Post_State` and a write-write race when a post-condition attempts to establish a new value for $\mathtt{var}_i$`.Post_State` itself. We check for such conflicts separately, as architects may be interested in the particular type of race-conditions in their system.

3.2 Translating Provided Ports

Figure 13 shows the translation to ProMeLa of provided ports. Their methods are translated as a pair of mutually exclusive atomic actions (lines 16–22 and 23-26). Both are guarded by the delaying guards of the role port variables that have been assumed by the port (`roleAwaits` in line 5, which is part of both `compPos` and `compNeg` defined in lines 8-10). When both role and method protocol guards are satisfied, service requests are processed by the first atomic block of actions (lines 16-22), which computes the next values of the component and role variables and sends back a response to the caller. On line 18 we check the completeness of the required conditions, when the interaction constraints (`compPos`) are satisfied. If the role guards are satisfied and the negation of the method's accepts guard is also satisfied, then the service request is rejected (lines 23-26) and the model fails explicitly, so as to indicate that a service user has violated the protocol constraints of the provided service. Both atomic blocks use the extended (non-)ProMeLa expression `chanX ? msg : pred` to receive `msg` from `chanX` only when `msg` satisfies `pred` – we have implemented this ourselves.

As can be seen, the role constraints are *injected* in the corresponding port (see usage of `roleAwaits` in lines 8-10 of Figure 13). The same behaviour could have been achieved by using a wrapper around provided ports, in which case ports would not need to know about their role constraints. Wrappers however cannot constrain

```
1 Port2Promela_Required(SimpleCInstance comp, Port port)
2  FORALL method ∈ port.MethodSet
3   LET
4   roles = { method.roleMethod(r) | r ∈ port.RoleSet };
5   roleAwaits = { r.method.Awaits | r ∈ roles };
6   rolePostEnsures = { r.method.Ensures | r ∈ roles };
7
8   compPos = rAwaits ⋀ method.Awaits;
9
10  compFCProm =
11   {method.FCPromisesRequiresEnsures.Promises} ;
12  compFCReq = method.FCPromisesRequiresEnsures.Requires;
13  compFCEns = {method.FCPromisesRequiresEnsures.Ensures}
14              ⋃ rolePostEnsures;
15
16  RoleVarsRace = { v ∈ e.VarSet | e ∈ rolePostEnsures };
17  compVarsRace = { v ∈ method.Ensures.VarSet };
18    VarSetRace = RoleVarsRace ⋃ compVarsRace;
19 IN
20 ::atomic { // sending a request
21    selectParams(method.Args,
22                 comPos ⋀ !port.Lock,compFCProm)→
23      port.Lock = method;
24      FORALL var ∈ VarSetRace
25         var.Pre_State_Copy = var.Pre_State;
26      port.Channel_req
27          ! method.Args;
28    }
29 ::atomic { // receiving a response
30    port.Channel_res ? method.Args
31                     : port.Lock == method →
32      raceCheck(compVarsRace,compFCEns); // Check race-conditions
33      calcData(comp.SimpleCType.VarSet,compFCReq,
34               compFCEns);
35      port.Lock = 0;
36    }
```

Figure 14: Translating a required port to ProMeLa

required ports, as these can make requests whenever their protocol constraints allow them to do so. A wrapper of a required port could only delay such a request but it could not undo it – the request would still exist. For this reason we have opted for the injection of the role constraints directly into the components. This is similar to how human actors work – they are given the script of their roles to read, as, unlike marionettes, they are active entities which need to know when they should perform an action. Directors do not attempt to delay actions initiated by actors during a play.

3.3 Translating Required Ports

Required ports are translated to ProMeLa as shown in Figure 14. Now actions are translated into a pair of co-dependent atomic actions (lines 20-28 and 29-36). The first block initiates a service request to a provided port; the second treats the response.

If each port was a separate process then they would be specified as two sequential (non-atomic) steps – the port process would block after sending a request, until it would receive the response. In our translation however all ports are part of the same component process, so as to decrease the overall number of active processes (Spin has an upper limit). This is why we use a lock (**port**.`Lock`) per each required port, to hold the currently active method. When none is active, a request can be made, as long as we can also select appropriate method parameter values that meet the promise of the method and satisfy its protocol constraints (lines 21-22). In this case we keep copies of the variables that might suffer a race-condition, so as to identify these later, and emit the request message, updating the lock to indicate which method made the request.

Once a response can be received (line 29-31), we check for race-conditions among the variables (line 32), use the ensures clause of the method contract to compute new values for the component data (lines 33-34), and free the lock on this required port.

Table 1: Memory and time required for verifying architectural specifications

Case Study		Issues ‡	State-vector (Bytes)	States Stored	States Matched	Memory (MB)	Time (sec)
[5]	Centralized Nuclear Plant		424	168349	407776	186	1.21
	Decentralized Nuclear Plant	1	240	137	73	130	0.00
[39, 7, 27]	Lunar Lander v. 1	4	372	118	78	131	0.01
	Lunar Lander v. 2		392	4223125	8072166	3793	15.50
[29]	Gas Station (1 customer)		188	1003	1401	130	0.00
	Gas Station (2 customers)		288	1136214	2793961	382	3.23
	Gas Station (3 customers)		368	25056808	89254880	7024†	78.00
	BITSTATE Gas Station (3 customers)		368	62792292	207452380	24	242.00
	BITSTATE Gas Station (4 customers)		456	66989014	289982810	25	321.00
	BITSTATE Gas Station (5 customers)		544	69607515	356984080	26	365.00
[3]	Aegis v. 1	2	620	13834057	71301546	7024†	52.00
	BITSTATE Aegis v. 1	2	620	64408848	266469200	37	330.00
	BITSTATE Aegis v. 2		548	63568962	268078040	35	304.00
[15]	English auction v. 1 (1 participant)	3, 4	140	296	295	130	0.00
	English auction v. 2 (1 participant)	4	144	776	1642	130	0.00
	English auction v. 2 (2 participants)	4	232	1293488	3732650	367	5.00
	English auction v. 2 (3 participants)	4	312	27315867	96797687	7024†	134.00
	BITSTATE English auction v. 2 (3 participants)	4	312	57105380	189090640	20	310.00

‡ Issues: The model fails to satisfy a property. 1: glue, 2: local deadlock, 3: global deadlock, 4: buffer overflow.
Column "States Stored" shows the number of unique global system states stored in the state-space, while column "States Matched" the number of states that were revisited during the search – see: `spinroot.com/spin/Man/Pan.html#L10`
† Cases marked with † in the Memory column run out of memory.
Using a 64bit Intel Xeon CPU (W3503 @ 2.40GHz × 2), 11.7GB of RAM, and Linux version 3.5.0-39-generic.
Spin (version 6.2.4) and gcc (version 4.7.2) used, with up to 7024MB of RAM and a search depth of 50,000:

```
spin -a configuration.pml
gcc -DMEMLIM=7024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan pan.c
./pan -m50000 -c1
```

For bit-state verification, the `-DBITSTATE` option needs to be passed to gcc. All case studies available at the XCD web site [40].

4. TOOL EVALUATION

We have evaluated our language and translation tool by considering a number of well-known case studies, apart from the nuclear plant used so far. The Lunar Lander [39, 7, 27] has been considered extensively in the software architecture community. A number of sensors and actuators are controlled by a single controller that attempts to safely land a spacecraft on the moon. The Gas Station [29], another classic case study in software architectures, consists of a number of gas pumps and customers that need to pay a cashier before a pump is released for them. The Aegis Weapons System [3] is a Command-and-Control system developed by the US Navy using a client-server approach, containing a number of sensors to establish the environment a ship is in and components that analyze this context in order to react to potential threats. Finally, FIPA's English Auction [15], describes a marketplace with an auctioneer who uses the English auction variant to sell an item.

The XCD models of these systems (available at the XCD web site [40]) were translated into ProMeLa so as to verify various properties that are encoded in our translation. First, we verify that users respect the protocol constraints of provided services, i.e., no chaotic behaviours are possible. Second, we verify that provided services functional pre-conditions are complete when their protocol constraints are satisfied. Third, we verify against race-conditions, write-read and write-write ones. Fourth, we verify that when using events, then the finite size of the asynchronous channel buffers suffices. Finally, Spin itself verifies deadlock-freedom.

Currently our language and tool do not allow the specification in XCD of other, more general properties, e.g., like the nuclear plant glue property. For these one needs to edit the produced ProMeLa model. We added an extra ProMeLa process (`glueP`), which receives messages through an additional channel from the `UR` and `NA` processes whenever they act on a request and checks if the glue property sequence is respected. Most importantly, we had to modify the produced ProMeLa models of `UR` and `NA`, so that they notify `glueP`. These modifications added another message emission (to `glueP`) in their code, right before they emit the `inc`/`double` method response in the atomic block, i.e., between lines 20 and 21 of Figure 13. This should be done carefully and only after having verified the general properties, as these message emissions render provided methods non-atomic – they terminate the atomic block in Spin (since emission is blocked by a full channel buffer). This is how we verified that while the decentralized version of the plant does not satisfy the property, the centralized version of it (in Figure 6-9) satisfies it.

Table 1 shows the obtained experimental results. These case studies can be analyzed extremely quickly in most cases, with a reasonable amount of memory. When memory is insufficient (marked with a †), one can use Spin's bit-state hashing mode, which reduces memory drastically through Bloom filters [10].

We view these results as extremely promising – they indicate that a formal architectural analysis of systems is far from unrealistic, even when these are described with such detailed models (e.g., modelling method parameters). We believe that having widespread support for this is something that can improve software systems quality substantially, as architectural errors that are not identified early are extremely costly to correct at later development stages and can easily lead to project failure. At the same time, powerful architectural analysis greatly facilitates architectural design exploration, thus helping designers to consider many more alternatives when designing their systems, without increasing their workload or the overall cost unreasonably.

The downside of our approach is that components and connectors cannot be analyzed in isolation, as Spin requires a closed system. For each component one wishes to analyze, a corresponding testing component is needed. Similarly, for each connector one wishes to analyze, a testing component is needed for each role.

5. RELATED WORK

All the ADLs supporting connectors that we have studied permit the specification of unrealizable connectors [33], since they all support an element like Wright's glue. Of the ADLs we studied that do not support connectors, all of them are realizable apart from Rapide [24], which allows the specification of global constraints.

ArchJava [1, 2] supports connectors but targets code generation, not formal analysis. It uses reflection to type-check that connector roles are associated with appropriate component ports but this considers just their interfaces. Connector roles function as wrappers to component ports, thus we cannot see how required port methods can be (temporarily or permanently) deactivated (not just delayed), as XCD can do by strengthening their protocol constraints.

Trust-By-Contract [38] uses DbC to describe component port protocols but does not support connectors. XCD also follows a more programming-like approach in the description of interfaces and contracts, like JML [11, 37], so that it looks more familiar to practitioners than the usual formal notations used in ADLs. Unlike JML that allows it but does not insist on it, XCD imposes the separation of protocol and functional constraints – we believe that this can make both easier to understand. XCD also extends DbC to support required methods too, that JML does not consider (as they are not part of a class's interface – only provided methods are).

Archface [42] is geared more towards code generation and design/code bidirectional traceability. In Archface connector roles are specified through interfaces (called component interfaces) that also contain predicates on aspect-oriented "pointcuts", such as "call (method call), execution (method execution), and cflow (control flow)". These seem to be able to describe a local role behaviour like in XCD, though the use of interfaces means that Archface cannot have as fine control as XCD – one cannot declare role variables. The connector element itself specifies how some role interface ports (i.e., methods) are connected to each other, adding further interaction constraints. These constraints can only be applied at the provided method side "A connector interface represents connections among ports. The types of advice that can be applied to [a provided port] are declared in an in statement." [42, p. 80]. We could not see any global constraints in the provided examples, so it seems that Archface specifications are realizable. The formal ProMeLa models produced are far simpler than those for XCD, not modelling component data, method parameters, race-conditions, etc. Indeed, constraints on these cannot be specified in Archface. Its input language requires that users know AOP, while XCD does not require so. We failed to understand how connector usage integrity is achieved – consider the following architecture [42, List 3, lines 01-04, p. 79]:

```
architecture aObserverPattern {
 class Subject  implements cSubject;
 class Observer implements cObserver;}
```

Types `cSubject` and `cObserver` are role interfaces used by a connector called `cObserverPattern`. But the latter does not appear in the architecture (nor did we find a rule that makes it impossible for another connector to use the same role interfaces). We cannot see what would happen if designers forgot to instantiate one of the roles or added an `Observer` component instance without stating that it implements `cObserver`. In XCD a connector is instantiated explicitly and the components that use it are passed as parameters to the connector constructor, so there is no doubt of which connector is being used or which component has assumed which role.

6. DISCUSSION AND CONCLUSIONS

The XCD formal architectural description language (ADL) supports arbitrary, user-defined connectors/protocols that are guaranteed to be realizable. It does so without requiring underlying mechanisms that introduce extra, unspecified information flows, e.g., distributed consensus protocols, which break communication integrity. XCD guarantees connector realizability by not allowing the expression of any global interaction constraints. All constraints in XCD are expressed using local state and therefore each interacting party in a protocol knows at any time what it needs to do. All other ADLs we have studied [33] fail in this respect because they allow architects to impose any kind of global constraint through what they call connector glue – XCD has no connector glue.

We believe that support for user-defined connectors is crucial if we are ever to achieve the goal of CBSE for modular, reusable component specifications that we can easily adapt through our connectors when exploring different architectural solutions. Without support for connectors, one needs to restrict component specifications to specific protocol interactions, thus reducing their reusability, while substantially increasing their complexity at the same time.

XCD also attempts to increase the uptake of formal ADLs by practitioners, through a programming language-like syntax and use of Design-by-Contract (DbC) concepts. As reported recently [26], practitioners find that formal ADLs have a "steep-learning curve", as these require the use of process algebras. Compared to languages like π-Calculus or CSP, we believe that XCD specifications are easier to understand. We have extended DbC to better support components (and connectors), by splitting contracts into their protocol and functional parts and by providing contractual support for required services, along with that already existing for provided services.

Our experience so far with the tool [40] that translates XCD into ProMeLa models is quite encouraging. We can verify that *(i)* users of provided services respect their local protocol constraints, *(ii)* functional pre-conditions of provided services are complete (modulo their protocol constraints), *(iii)* there are no race-conditions, *(iv)* event buffer sizes suffice, and *(v)* there is no global deadlock.

We are working on improving the support for component/role arrays and recursive definitions, as well as the efficiency of our models. A user-friendly (sub-)language for expressing general properties, e.g., a glue, is an open issue.

7. REFERENCES

[1] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting software architecture to implementation. In W. Tracz, M. Young, and J. Magee, editors, *ICSE*, pages 187–197. ACM, 2002.

[2] J. Aldrich, V. Sazawal, C. Chambers, and D. Notkin. Language support for connector abstractions. In L. Cardelli, editor, *ECOOP*, volume 2743 of *LNCS*, pages 74–102, Darmstadt, Germany, July 2003. Springer-Verlag.

[3] R. Allen and D. Garlan. A case study in architectural modelling: The Aegis system. In *IWSSD-8*, pages 6–15, Paderborn, Germany, Mar. 1996.

[4] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM TOSEM*, 6(3):213–249, July 1997.

[5] R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. *IEEE TSE*, 29(7):623–633, 2003.

[6] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. *Theor. Comput. Sci.*, 331(1):97–114, 2005.

[7] H. Bagheri and K. J. Sullivan. Monarch: Model-based development of software architectures. In D. C. Petriu, N. Rouquette, and Ø. Haugen, editors, *MoDELS (2)*, volume 6395 of *LNCS*, pages 376–390. Springer, 2010.

[8] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis. Rigorous component-based system design using the BIP framework. *IEEE Software*, 28(3):41–48, 2011.

[9] S. Basu, T. Bultan, and M. Ouederni. Deciding choreography realizability. In J. Field and M. Hicks, editors, *POPL*, pages 191–202. ACM, 2012.

[10] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[11] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, 2005.

[12] T. Bures. Automated synthesis of connectors for heterogeneous deployment. Tech. report no. 2005/4, Dep. of SW Engineering, Charles University, Prague, Aug. 2005.

[13] T. Bures, P. Hnetynka, and F. Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *SERA*, pages 40–48. IEEE Computer Society, 2006.

[14] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[15] FIPA TC C. FIPA English auction interaction protocol specification. Technical Report XC00031F (Experimental), FIPA, Aug. 2001. `www.fipa.org/specs/fipa00031/`.

[16] O. Galik and T. Bures. Generating connectors for heterogeneous deployment. In E. D. Nitto and A. L. Murphy, editors, *SEM*, pages 54–61. ACM, 2005.

[17] 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.

[18] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is still so hard. *IEEE Software*, 26(4):66–69, 2009.

[19] 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.

[20] G. J. Holzmann. The Spin model checker. *IEEE TSE*, 23(5):279–295, May 1997.

[21] C. Kloukinas and M. Ozkaya. XCD - Modular, realizable software architectures. In C. S. Pasareanu and G. Salaün, editors, *FACS*, volume 7684 of *LNCS*, pages 152–169. Springer, 2012.

[22] K.-K. Lau, P. V. Elizondo, and Z. Wang. Exogenous connectors for software components. In G. T. Heineman, I. Crnkovic, H. W. Schmidt, J. A. Stafford, C. A. Szyperski, and K. C. Wallnau, editors, *CBSE*, volume 3489 of *LNCS*, pages 90–106. Springer, 2005.

[23] G. Lekeas, C. Kloukinas, and K. Stathis. Producing enactable protocols in artificial agent societies. In D. Kinny, J. Y. jen Hsu, G. Governatori, and A. K. Ghose, editors, *PRIMA*, volume 7047 of *LNCS*, pages 311–322. Springer, 2011.

[24] D. C. Luckham, J. Kenney, L. Augustin, J. Verra, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE TSE*, 21(4):336–355, Apr. 1995.

[25] J. Magee and J. Kramer. *Concurrency – state models and Java programs*. Wiley, 2 edition, 2006.

[26] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang. What industry needs from architectural languages: A survey. *IEEE TSE*, 39(6):869–891, 2013.

[27] S. Maoz, J. O. Ringert, and B. Rumpe. Synthesis of component and connector models from crosscutting structural views. In B. Meyer, L. Baresi, and M. Mezini, editors, *ESEC/SIGSOFT FSE*, pages 444–454. ACM, 2013.

[28] B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, 1992.

[29] G. Naumovich, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil. Applying static analysis to software architectures. In M. Jazayeri and H. Schauer, editors, *ESEC / SIGSOFT FSE*, volume 1301 of *LNCS*, pages 77–93. Springer, 1997.

[30] OMG. Common object request broker architecture (CORBA) specification, version 3.3 – Part 3: CORBA component model. Specification formal/2012-11-16, OMG, Nov. 2012. `omg.org/spec/CORBA/3.3/`.

[31] OSGi Alliance. OSGi core release 5. Specification, Mar. 2012. `osgi.org`.

[32] M. Ozkaya and C. Kloukinas. Highly analysable, reusable, and realisable architectural designs with XCD. In T.-h. Kim, C. Ramos, H.-k. Kim, A. Kiumi, S. Mohammed, and D. Slezak, editors, *Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity*, volume 340 of *CCIS*, pages 72–79. Springer Berlin Heidelberg, 2012.

[33] M. Ozkaya and C. Kloukinas. Are we there yet? Analyzing architecture description languages for formal analysis, usability, and realizability. In O. Demirors and O. Turetken, editors, *SEAA*, pages 177–184, Santander, Spain, Sept. 2013. IEEE.

[34] M. Ozkaya and C. Kloukinas. Towards a design-by-contract based approach for realizable connector-centric software architectures. In J. Cordeiro, D. A. Marca, and M. van Sinderen, editors, *ICSOFT*, pages 555–562. SciTePress, 2013.

[35] M. Ozkaya and C. Kloukinas. Towards design-by-contract based software architecture design. In *SoMeT*, pages 157–164. IEEE, 2013.

[36] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, Oct. 1992.

[37] E. Rodríguez, M. B. Dwyer, C. Flanagan, J. Hatcliff, G. T. Leavens, and Robby. Extending JML for modular specification and verification of multi-threaded programs. In A. P. Black, editor, *ECOOP*, volume 3586 of *LNCS*, pages 551–576. Springer, 2005.

[38] H. W. Schmidt, I. Poernomo, and R. H. Reussner. Trust-by-contract: Modelling, analysing and predicting behaviour in software architectures. *Journal of Integrated Design and Process Science*, 5(3):25–51, September 2001.

[39] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2010.

[40] XCD. Website, 2013. Maintained by Mert Ozkaya. URL: www.staff.city.ac.uk/c.kloukinas/Xcd/.

[41] S. Tripakis. Undecidable problems of decentralized observation and control on regular languages. *Inf. Process. Lett.*, 90(1):21–28, 2004.

[42] N. Ubayashi, J. Nomura, and T. Tamai. Archface: A contract place where architectural design and code meet together. In J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, editors, *ICSE*, pages 75–84. ACM, 2010.

[43] S. Uchitel, J. Kramer, and J. Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM TOSEM*, 13(1):37–85, 2004.