
This is the accepted version of the paper.

This version of the publication may differ from the final published version.

Permanent repository link: http://openaccess.city.ac.uk/4069/

Link to published version: http://dx.doi.org/10.1145/2602458.2602463

Copyright and reuse: City Research Online aims to make research outputs of City, University of London available to a wider audience. Copyright and Moral Rights remain with the author(s) and/or copyright holders. URLs from City Research Online may be freely distributed and linked to.
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 of Component-based Software Engineering, as they allow one to specify the protocol-independent components of a 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 analyzeable 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

@conference{CBSE14, title={Design-by-Contract for Reusable Components and Realizable Architectures}, author={Mert Ozkaya and Christos Kloukinas}, year={2014}, month={June 30–July 4}, publisher={City University London, School of Informatics, London EC1V 0HB, U.K.}, address={London EC1V 0HB, U.K.}, url={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 [5] 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


copyright@acm.org
problems, global interaction constraints allow architects to present interacting component actions together does not create any realizability.

Note: barred actions are initiated by the current process, → successor in existing ADLs. Indeed, Figure 1d shows the Wright [4] 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” [8]. 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 [53]. 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 [54].
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 functional or interaction port.

Figure 2: Simple components (SimpleType) grammar

documentation 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 pre-condition (requires 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 \( 0 \leq x + y + z \leq n \) means that we need to consider all possible combinations of \( x, y, z \) within the range \([0, n]\), i.e., \((n+1)\) states. Instead, architects write this as \( x \in [0, n], y \in [0, n-x], z \in [0, n-x-y], \) which has \((n+1)(n^2 + 5n + 6)\) states. 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 10pm (protocol) and offer an Italian menu (functional). Symmetrically, a customer may require a service between 9pm and 10pm (protocol) and offer an Italian menu (functional).

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

```java
Component Thread {
    boolean started := false; // component data.
    boolean died := false;
}

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

    @functional (ensures: \result := started && !died)
    void isAlive();

    @interaction {waits: died;
    void join();

    // ... other methods
};
```
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 waitForNotify 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:

\[(r \text{ waits } \rightarrow \text{r promises}) \rightarrow (p \text{ waits } \rightarrow \text{p accepts})\]

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 XTypeParam ( XTypeParam + )
2. { Role XInstance + }
3. | XTypeParam : ID XInstance + 
4. | XInstanceName XInstanceArg + ;
5. | XInstanceArg : ID ID + 
6. | XInstanceArg ID { ID + } | Expression ;
7. | Role : role ID RoleName
8. | { Variable + ([required|provided] PortVar) + } ;
9. | PortVar ID PortVarName
10. | { [ XProtocol MethodSignature + ] | } ;
11. | XProtocol : @interaction 
12. | { [ waits Expression ]? ensures Assignments 
13. | [ waits Expression ]? } ;
14. | XInstance : ID XInstanceName XInstanceArg + ;
15. | XInstanceArg : ID XInstanceArg ID XInstanceArg + | 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[2] 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 its 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 glue – 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 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[4]. The glue property states that UR and NA should always increase and double their quantities in tandem: UR.1 -> UR.1, UR.1 -> NA.1, UR.1 -> UR.1, UR.1 -> NA.1, where 1
and 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 4 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 2 and Figure 3, presents itself as UR and NA to P1 and P2 using its provided ports (lines 60-71 for 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 ? expr : 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 [42]. Compared to the Wright connector in Figure 1d, the XCD connector is longer – much more so. This is for two main reasons.
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 subcomponents. 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 (\(\alpha_{ci}^p\)), with a buffer size equal to the number \(N = \text{connectedTo}(c_i, p)\) of required ports that are 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 \text{var} is mapped to two variables (lines 27-29 of Figure 12). The first one (\text{var\_Pre\_State}) 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 (\text{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 \text{var\_Post\_State} may refer to some \text{var\_Pre\_State} values.
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_Copy has been modified var.Pre_State, 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 var_Pre_State_Copy to establish the value of some var.First_State and a write-write race when a post-condition attempts to establish a new value for var.First_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 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

<table>
<thead>
<tr>
<th>Case Study</th>
<th>Issues ‡</th>
<th>State-vector States</th>
<th>States Matched</th>
<th>Memory (MB)</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized Nuclear Plant</td>
<td>1</td>
<td>424</td>
<td>168349</td>
<td>407776</td>
<td>186</td>
</tr>
<tr>
<td>Decentralized Nuclear Plant</td>
<td>2</td>
<td>240</td>
<td>137</td>
<td>73</td>
<td>130</td>
</tr>
<tr>
<td>Lunar Lander v. 1</td>
<td>3</td>
<td>372</td>
<td>118</td>
<td>78</td>
<td>131</td>
</tr>
<tr>
<td>Gas Station (1 customer)</td>
<td>4</td>
<td>288</td>
<td>1003</td>
<td>1401</td>
<td>139</td>
</tr>
<tr>
<td>Gas Station (2 customers)</td>
<td>5</td>
<td>368</td>
<td>1136214</td>
<td>2793961</td>
<td>392</td>
</tr>
<tr>
<td>Gas Station (3 customers)</td>
<td>6</td>
<td>456</td>
<td>25056808</td>
<td>89254880</td>
<td>70241</td>
</tr>
<tr>
<td>BITSTATE Gas Station (3 customers)</td>
<td>7</td>
<td>544</td>
<td>66989014</td>
<td>289982810</td>
<td>25</td>
</tr>
<tr>
<td>BITSTATE Gas Station (4 customers)</td>
<td>8</td>
<td>638</td>
<td>46907515</td>
<td>35698480</td>
<td>26</td>
</tr>
<tr>
<td>BITSTATE Gas Station (5 customers)</td>
<td>9</td>
<td>732</td>
<td>64408884</td>
<td>266469200</td>
<td>37</td>
</tr>
<tr>
<td>Aegis v. 1</td>
<td>10</td>
<td>820</td>
<td>13834057</td>
<td>71301546</td>
<td>10241</td>
</tr>
<tr>
<td>BITSTATE Aegis v. 1</td>
<td>11</td>
<td>920</td>
<td>64808848</td>
<td>266469200</td>
<td>37</td>
</tr>
<tr>
<td>English auction v. 1 (1 participant)</td>
<td>12</td>
<td>1020</td>
<td>45689628</td>
<td>268078040</td>
<td>35</td>
</tr>
<tr>
<td>English auction v. 2 (1 participant)</td>
<td>13</td>
<td>1120</td>
<td>45689628</td>
<td>268078040</td>
<td>35</td>
</tr>
<tr>
<td>English auction v. 2 (2 participants)</td>
<td>14</td>
<td>1220</td>
<td>45689628</td>
<td>268078040</td>
<td>35</td>
</tr>
<tr>
<td>English auction v. 2 (3 participants)</td>
<td>15</td>
<td>1320</td>
<td>45689628</td>
<td>268078040</td>
<td>35</td>
</tr>
</tbody>
</table>

‡ Issues: The model fails to satisfy a property. 1: glue, 2: local deadlock, 3: global deadlock, 4: buffer overflow.

Using a 64bit Intel Xeon CPU (W3503 @ 2.40GHz × 2), 11.7GB of RAM, and Linux version 3.5.0-39-generic.

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 [29][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] is 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 sa 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 sa, 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) 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 sup-
port 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 [12] 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 [35] 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 [22] 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 type 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:

\[
\text{architecture aObserverPattern} \{ \\
\text{class Subject implements cSubject;} \ \\
\text{class Observer implements cObserver;} \}
\]

Types \(c\text{Subject}\) and \(c\text{Observer}\) are role interfaces used by a connector called \(c\text{ObserverPattern}\). 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 \(c\text{Observer}\) component instance without stating that it implements \(c\text{Observer}\). 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 [13] 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 “sleep-learning curve”, as these require the use of process algebras. Compared to languages like \(\pi\)-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
