
Permanent City Research Online URL: http://openaccess.city.ac.uk/4069/

Copyright & reuse
City University London has developed City Research Online so that its users may access the research outputs of City University London's staff. Copyright © and Moral Rights for this paper are retained by the individual author(s) and/or other copyright holders. All material in City Research Online is checked for eligibility for copyright before being made available in the live archive. URLs from City Research Online may be freely distributed and linked to from other web pages.

Versions of research
The version in City Research Online may differ from the final published version. Users are advised to check the Permanent City Research Online URL above for the status of the paper.

Enquiries
If you have any enquiries about any aspect of City Research Online, or if you wish to make contact with the author(s) of this paper, please email the team at publications@city.ac.uk.
ABSTRACT

Architectural connectors can increase the modularity and reusability 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.2.4 [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.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.
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.
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.”

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.

One can check conditions implying a protocol’s realizability [6, 9] to identify implied scenarios from the protocol [13], 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 [31]. 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.\text{UR}→P2.\text{NA}→NA.\text{NA}→\text{glue}) 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 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]. 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) 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” [4]. 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, 22], and using FSP [23] to specify and verify architectures [13]. 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].
and discusses others (intervals in section 2.1). It shows how global
to verify more properties. It demonstrates most of the new features
signers, and shows how designers can modify the ProMeLa models
language mappings to Spin’s ProMeLa language, so as to enable
2. CONTRACTS FOR ARCHITECTURES
of well-known architectural case studies (all available at [40]), and
also includes an extensive experimental evaluation using a number
component when decentralized control is impossible. The paper
supporting events due to lack of space. Each port can be either a
set of ports for interacting with its environment. We ignore ports
to specify connectors/protocols as well as components.

Figure 2: Simple components (SimpleType) 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 des-
igners, and shows how designers can modify the ProMeLa models
to verify more properties. It demonstrates most of the new features
(enumeration 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 spec-
ify 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 contract components, 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

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 oper-
ate 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 3 shows the high-level
grammar for simple components, abstracting over a number of lan-
guage 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 es-
entially be described through classic DBC. Ignoring the interac-
tion contract, whenever a method is called and the method pre-
condition (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 specifi-
cations. 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 ≤ x + y + z ≤ 2 means that we need to consider all
possible combinations of x,y,z within the range [0,n], i.e., (n+1)2
states. Instead, we write this as x ∈ [0,n], y ∈ [0,n−x], z ∈
[0,n−x−y], which has (n+1)(n2+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, clas-
sic DBC does not consider them. These are actions that the com-
ponent enacts itself, instead of actions that it reacts to (in its pro-
vided ports). A restaurant may provide a service between 7pm and
11pm (protocol) and offer an Italian menu (functional). Symmet-
ically, 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/
?q= sum x=0^n+n^2+y=0-(n-x)+sum z=0^n-(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 process 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.waits → r.promises) → (p.waits → 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.
and 4 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 is longer – much more so. This is for two main reasons. The expression on lines 64-65 uses the helper functions from UR and NA to update their fuel amounts. This behaviour uses 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 10. 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 11 and Figure 12, 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 first in each round, using the variable all_received() and the operator || (lines 55-58). Depending on whether it was the increase or the double that was received last, action inc_emitted() or dbl_emitted() is used (lines 59-62).
For each simple component (ware systems, where asynchronous interaction is the mainstream. We use asynchronous channels in our models, as we target soft- is fixed in each architecture (we consider only static architectures). a separate ProMeLa process. The number of component instances formally verify architectures. Each component 
composing component where all the ports of the sub-components are connected to the component itself (i.e., knowing that at most one method request to be active each time. We also intro- 
duce 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 re-quired 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 sec- ond 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 var. Post_State may refer to some 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 \( \text{var.Pre\_State} \) for each data var appearing in an \( \text{ensures} \) clause (lines 30-31). This variable keeps a copy of the data’s pre-value (\( \text{var.Pre\_State} \)) at the point the request was started at the port. For required port methods, we have that \( \text{var.Pre\_State} = \text{var.Pre\_State\_Copy} \) before and immediately after enacting the method request. But when the response is received we may find that \( \text{var.Pre\_State} \neq \text{var.Pre\_State\_Copy} \), because some other component port has modified \( \text{var.Pre\_State} \) (the current variable value) in between. This is a write-read race when a post-condition attempts to use the value \( \text{var.Pre\_State} \) at the point the interaction was started at the port. We 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.

### 3.2 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 (\( \text{role\_awaits} \) in line 5, which is part of both \( \text{compPos} \) and \( \text{compNeg} \) 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 (\( \text{compPos} \)) 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 \( \text{chanX } \Rightarrow \text{msg } \Rightarrow \text{pred} \) to receive \text{msg} from \text{chanX} only when \text{msg} satisfies \text{pred} – we have implemented this ourselves.

As can be seen, the role constraints are injected in the corresponding port (see usage of \( \text{role\_awaits} \) 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 (\( \text{port\_Lock} \)) 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.
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], 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 [13], 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 un 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 un and sa, so that they notify glueP.

These modifications added another message emission (to glueP) in their code, right before they emit the un/double method response in the atomic block, i.e., between lines 20 and 21 of Figure [29]. 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 [29]) 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 [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 [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 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]:

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

Types `Subject` and `Observer` 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 [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 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 [50] 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


