
This is the unspecified version of the paper.

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

Permanent repository link: http://openaccess.city.ac.uk/2881/

Link to published version:

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.
Towards a Design-by-Contract Based Approach for Realizable Connector-Centric Software Architectures

Mert Ozkaya and Christos Kloukinas
Department of Computer Science, City University London, London, EC1V 0HB, UK
{mert.ozkaya.1, C.Kloukinas}@city.ac.uk

Keywords: Architecture Description Language, Design by Contract, Complex Interaction Protocols, Realizable Designs

Abstract: Despite being a widely-used language for specifying software systems, UML remains less than ideal for software architectures. Architecture description languages (ADLs) were developed to provide more comprehensive support. However, so far the application of ADLs in practice has been impeded by at least one of the following problems: (i) advanced formal notations, (ii) lack of support for complex connectors, and (iii) potentially unrealizable designs. In this paper we propose a new ADL that is based on Design-by-Contract (DbC) for specifying software architectures. While DbC promotes a formal and precise way of specifying system behaviours, it is more familiar to practising developers, thus allowing for a more comfortable way of specifying architectures than using process algebras. Furthermore, by granting connectors a first-class status, our ADL allows designers to specify not only simple interaction mechanisms as connectors but also complex interaction protocols. Finally, in order to ensure that architectural designs are always realizable we eliminate potentially unrealizable constructs in connector specifications (the connector “glue”).

1 Introduction

Since the early work on software architectures [Perry and Wolf, 1992, Garlan and Shaw, 1993], specialized architecture description languages (ADLs) [Medvidovic and Taylor, 2000] have been proposed for specifying software architectures. But UML has become a de facto design language for specifying and designing software systems – more practitioners use it than all other languages (e.g., AADL, ArchiMate, etc.) combined [Malavolta et al., 2013], even though it is less than ideal [Dashofy et al., 2002]. This is despite some ADLs having formally defined semantics, enabling not only formally precise models that help avoid confusion about what is meant by the specification but also early formal analysis of software architectures. Other ADLs offer automatic code generation, etc. Nevertheless, UML is still the first choice of practitioners for specifying software architectures, with most ADLs seemingly remaining of interest to the research community alone. In our view, there are three main problems that ADLs suffer from: (i) formal notations for behaviour specifications that require a steep learning curve, (ii) lack of support for complex connectors (i.e., interaction protocols), and (iii) potential for producing unrealizable designs. Indeed, to the best of our knowledge, there is no ADL that is easy to learn, treats connectors as first-class elements and ensures that architecture specifications are realizable. While (i) has been identified by practitioners as being a serious impediment to their adoption of current ADLs, (ii) is not an issue that they consider as crucial but we believe that it can substantially help in developing concise designs, as it promotes reuse-by-call instead of reuse-by-copy. Condition (iii) is in fact something that has not been identified so far at all to the best of our knowledge but we believe that it is crucial to identify and resolve, if a connector-centric ADL is to succeed.

Formal notations Many ADLs (e.g., Wright [Allen and Garlan, 1997], LEDA [Canal et al., 1999], SOFA [Plasil and Visnovsky, 2002], CONNECT [Issarny et al., 2011], etc.) adopt formal notations, e.g., process algebras [Bergstra et al., 2001], for specifying the behaviours of architectural elements. They do so in order to enable the architectural analysis of systems, which is extremely important in uncovering serious system design errors early on in the lifetime of

1Along with a number of researchers, as many ADLs do not support connectors.

2Perhaps due to the non-adoption of ADLs supporting connectors, as it is a side-effect of their connector specification structure.
a project. Indeed, if such an analysis is not possible, then there is no point in using a specialized language for software architectures – even simple drawings suffice. However, the aforementioned ADLs employ notations that practitioners view (with reason) as having a steep learning curve [Malavolta et al., 2013]. Thus, practitioners end up avoiding them and use instead simpler languages, even if that means that they lose the ability to properly describe and analyze their systems – better an informal description of a system that everybody understands than a formal description of a system that people struggle understanding.

**Limited support for complex connectors** Another problem with many ADLs (e.g., Darwin [Magee and Kramer, 1996], Rapide [Luckham, 1996], LEDA [Canal et al., 1999], and AADL [Feiler et al., 2006]), as well as with UML, is that they provide limited or no support for complex connectors, treating them instead as simple connections. Other ADLs, such as UniCon [Shaw et al., 1995], provide only partial support through a limited number of basic connectors but do not allow designers to freely specify their own (complex) types. This is unfortunate because connectors represent the interaction patterns between components, i.e., the interaction protocols that are employed to achieve the system goals using the system components, such as reliability. By instead offering support only for components, architects end up with two alternatives. One is to ignore protocols, which inhibits the analysis of crucial system properties, such as deadlock-freedom, and also leads to architectural mismatch [Garlan et al., 1995], i.e., the inability to compose seemingly compatible components due to wrong assumptions these make about their interaction. The other is to incorporate the protocol behaviour inside the components themselves, which leads to complicated component behaviour that is neither easy to understand nor to analyze and makes it difficult to reuse components with different protocols, as well as to find errors in specific protocol instances. Incorporating protocol behaviour inside components is essentially following a reuse-by-copy approach, whereby each component has its own copy of the protocol constraints. On the other hand, support for protocols through first-class connectors promotes a reuse-by-call approach, whereby there is only one instance of the protocol constraints and these are simply called wherever they are needed.

**Potentially unrealizable designs** The third problem of existing ADLs is that they do support user-defined, complex connectors, they do so in a way that can lead to unrealistic designs. This is because,
rithm that can be implemented to warn architects that
the glue they are specifying is not realizable by the
existing roles. The only easy solution to realize a pro-
tocol then is to introduce yet another component that
will assume the role of the glue, thus transforming all
protocols into centralized ones.

An example of such an unrealizable protocol from
a simplified nuclear power plant is described in [Alur
et al., 2003] and reproduced in Figure 1. The in-
teraction therein involves an Incrementor \((P_1)\) and a
Doubler \((P_2)\) client roles updating the amounts of
the Uranium fuel \((UR)\) and Nitric Acid \((NA)\) server pro-
cesses in a nuclear reactor. After the update opera-
tions, the amounts of UR and NA must be equal to
avoid nuclear accidents, for which reason we wish to
allow only the sequences shown in Figure 1. The in-
teraction of the two clients with the NA and UR vari-
ables, can easily be specified in Wright as in Figure 2.
Note that this glue specification does two things. First
it establishes bindings between clients and servers
(e.g., Incrementor.ur \(\rightarrow\) UR.increment). Then it con-
straints interactions by requiring that we only al-
low \(UR.increment \rightarrow NA.increment\) or \(UR.double \rightarrow
NA.double\). This specification is however unrealiz-
able as [Alur et al., 2003] have shown because it is
impossible to implement it in a decentralized manner
in a way that avoids behaviours excluded by the glue,
e.g., the one depicted in Figure 3. The only way to
achieve the desired behaviour is to introduce another
role, for a centralized controller \(G\). Roles Incrementor
and Doubler then need to inform \(G\) when they wish to
interact with UR and NA and have \(G\) perform the in-
teractions with \(UR\) and \(NA\) in their place.

2 Our Approach

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

2.1 Support for Complex Connectors

Our ADL grants connectors in software architectures
first-class status, allowing designers to specify both
simple interaction mechanisms and complex proto-
cols. These can then be instantiated as many times
as needed, allowing architects to simplify the specifi-
cations of their components and easily reuse the spec-
ification of complex protocols.

To illustrate how important this is for both archi-
tectural understandability and also analysis, we will
use a simple example from electrical engineering. Let
us consider \(k\) concrete electrical resistors, \(r_1, \ldots, r_k\),
i.e., our system components. When using a sequential
connector \(\rightarrow\), the overall resistance is computed
as \(R(N, \{R_i\}_{i=1}^N) = \sum_{i=1}^{N} R_i\), where \(N, R_i\) are vari-
ables \((R_i\) correspond to connector roles), to be as-
signed eventually some concrete values \(k, r_j\). If us-
ing a parallel connector \(\|\) instead, it is computed
as \(R(N, \{R_i\}_{i=1}^N) = 1/\sum_{i=1}^{N} R_i\). So the interaction
protocol (connector) used is the one that gives us the
formula we need to use to analyze it – if it does not
do so, then we are probably using the wrong connec-
tor abstraction. The components \((r_j)\) are simply pro-
viding some numerical values to use in the formula,
while the system configuration tells us which specific
value \((k, r_j)\) we should assign to each variable \((N, R_i)\)
of the connector-derived formula. By simply enumer-
ating the wires/connections between resistors/components,
we miss the forest for the trees. This leads to ar-
chitectural designs at a very low level that is not easy
to communicate and develop – as [Delanote et al.,
2008] found the case to be with AADL.

Figure 4 shows the number of simple connectors
(identified with red ellipses) that are needed in our
system. It is easy to see that there are many of them
and it is not so easy to identify the protocol logic,
especially as the system size increases – this is the
equivalent of spaghetti code. By making interaction
protocols implicit in designs, analysis also becomes
difficult and architectural errors can go undetected un-
til later development phases. Indeed, we are essen-
tially forced to reverse-engineer the architect’s intent
in order to analyze our system – after all, the architect
did not select the specific wire connections by chance
but because they form a specific complex connector.
When complex connectors are employed instead as in
Figure 5 then the number of connectors to be consid-
ered is reduced substantially. This makes it much eas-

\[
\rightarrow (r_1, |(| \rightarrow (r_2, r_3), r_4))
\]

Figure 4: Simple connectors in circuits

Figure 5: Complex connectors in circuits
ier to understand the system and to analyze its overall resistance by taking advantage of the connector properties as:

\[
\begin{align*}
  r_1 &\rightarrow (r_1 || (\neg (r_2 \land r_3), r_4)) \\
  &= r_1 + r_1 || (\neg (r_2 \land r_3), r_4) \\
  &= r_1 + \frac{1}{1 + \frac{1}{r_2 + r_3} + \frac{1}{r_4}} \\
  &= r_1 + \frac{1}{r_2 + r_3 + \frac{1}{r_4}}
\end{align*}
\]

2.2 Realizable Software Architectures

Connectors in our ADL are not specified with glue-like elements. Instead, we consider connectors as a simple composition of roles, which represent the interaction behaviour of participating components, and channels that allow actions of one role to reach another. Coordination is now the responsibility of roles alone. If a particular property is desired then it must be shown that the roles satisfy it. But this is a problem that is decidable for finite state systems – model-checking. Thus an architect can easily specify a protocol and be sure that it has the required properties. Designers can also feel reassured that the architectural protocols are indeed realizable in principle.

So in the case of Figure 1, the architect should quickly realize that the desired property is not satisfied by the roles and opt for a centralized protocol instead, by adding a centralized controller. Thus, surprises are avoided – it becomes clear early on whether something can be made to work in a decentralized manner or not, as it is tested by the more experienced architect. The less experienced designers do not have to waste their time trying to achieve the impossible or take the easy way out and turn a decentralized protocol into a centralized one. We essentially turn the glue from constraints to be imposed, to a property that needs to be verified, thus turning an undecidable problem that the implementers have to deal with, into a decidable one for them (and pushing the responsibility to resolve the issue to the more experienced architect).

2.3 DBC-based Specifications

Java Modelling Language (JML) [Chalin et al., 2006] seems to be gaining popularity among developers, as they use it for “test-driven development” and even for static analysis in some instances. Our new ADL attempts to follow this trend so as to maximize adoption by practitioners. Thus, it departs from the ADLs that adopt process algebras, and instead follows a Design by Contract (DbC) [Meyer, 1992] approach like JML, specifying behavioural aspects of systems through simple pairs of method pre-/post-conditions, in a syntax reminiscent of JML. DbC allows for a formal specification of systems, as it is based on Hoare’s logic [Hoare, 1969] and VDM’s [Björner and Jones, 1978] rely-guarantee specification approach. DbC has so far been mainly considered for programming languages (e.g., Java through JML), which is why contracts have been restricted to provided services (i.e., class methods).

There are very few ADLs that employ DbC. The work of Schreiner et al. [Schreiner and Göschka, 2007] along with the TrustME ADL [Schmidt et al., 2001] are some of the very few examples applying DbC at the level of software architecture. Schreiner et al.’s work however does not support component interfaces emitting or receiving events. Moreover, they view connectors as simple interconnection mechanisms, providing no support for the contractual specification of complex interaction protocols. Likewise, TrustME supports only required and provided interfaces for components, again neglecting asynchronous events. Our approach attempts to apply DbC in a more comprehensive manner, covering both methods and events. As we view connectors as first-class elements, we use DbC to specify their behaviour as well. Our ADL further extends DbC by structuring contracts into functional and interaction parts.

2.3.1 Component Contracts

Components are used to specify at a high-level the functional units in software systems. Our ADL allows designers to describe freely as components whatever they deem appropriate in their system, not imposing, nor introducing, low-level notions (e.g., threads, processes) as AADL does. Component types are essentially specified in terms of (i) ports representing the points of interaction with their environment, and (ii) data representing the component state. Similarly to CORBA interfaces [OMG, 2006], component ports can be either event consumer/emitter ones, corresponding to CORBA sink and source events respectively, or provided/required method ones, corresponding to CORBA facets and receptacles.

The behaviour of components are specified in terms of contracts attached to event/method actions (e.g., method call and event emission) performed via the ports. Extending DbC, our ADL allows to attach contracts to required methods too, to specify usage patterns of the required service, and also to the events emitted/received by emitter/consumer ports. Another extension we applied is the separation of contract specification into functional and interaction contracts, allowing to distinguish between the functional and interaction behaviour. The former allows for relating component data, event/method parameters, and re-
sults, while the latter specify (i) the particular manner in which components want to behave (i.e., the order of actions), and (ii) the cases in which they do not know how to behave, thus leading to chaotic behaviour.

2.3.2 Connector Contracts

As aforementioned, connectors in our ADL are introduced to represent high-level complex interaction protocols, which the components interacting through connectors adhere to. Connector types are specified in terms of roles and channels. Each role can be viewed as a component wrapper as depicted in Figure 6, which further constrains the interaction behaviour of the component assuming it. A role is described with data and port-variables, mirroring a component – a role is essentially a component variable, with extra state. The port-variables of a role are bound to ports of the components assuming the role. Channels of a connector represent the communication links between interacting roles and can be synchronous or asynchronous. A channel is described with a pair of communicating role port-variables and its communication type.

The behaviour of connectors is specified with interaction contracts attached to the port-variable actions. These interaction contracts constrain port-variable actions so that the respective component ports behave in a particular manner (i.e., enforcing a certain action order), according to the desired interaction protocols. In doing so, components avoid undesired interactions with other components associated with the same connector. The end result is then a set of components interacting with their environments successfully to compose the whole system. All constraints imposed are local ones, ensuring realizability.

3 Case Study: Shared-Data Access

Listing 1 below illustrates our ADL’s DbC-based behavioural specification on shared-data access. An important thing to note is the overall syntax; it resembles JML, with expressions that shouldn’t feel alien to practitioners – a far step from process algebraic notations. Two component types are specified, user between lines 1–21 and memory between lines 22–44. User comprises a required port, puser_r (lines 4–12), through which its instances call method get from the memory and an emitter port, puser_e (lines 13–20), through which event set is emitted. Memory comprises a provided port, pmem_p (lines 25–34), through which its instances accept calls for method get from users and a consumer port, pmem_c (lines 35–43) through which event set is received.

Ports are specified in terms of method/event signatures and contracts (i.e., @functional and @interaction). @interaction constraints on port actions (e.g., method call and event emission) have precedence over @functional, i.e., the evaluation of the latter depends on the satisfaction of the former.

User Component Type

User’s required port puser_r is used to call method get. These calls are delayed until the promised condition specified in @interaction is met. Since it is true in line 6, get can be called immediately. Next, the @functional in lines 7–10 is evaluated; since get has no parameters, there is no promised actual parameters (promises condition in line 8 is true). Upon receiving the response after calling get, the received result is ensured to be stored in variable data.

The user components emit event set through its emitter port puser_e. Note that events are specified without return types – they can only have an identifier and parameters. The emission of event set is delayed until the promised condition of @interaction in lines 14–15 is met. Since it is specified as true, emission can be made at any instance, using the promised actual parameters specified in @functional (line 17). This then ensures that initialised_u is set to true. Also note that events are emitted/received asynchronously.

Memory Component Type

Memory’s provided port pmem_p receives calls for method get. These calls are accepted when the accepts condition in @interaction is met, initialised_m evaluating to true (line 27). Otherwise, the call is rejected when initialised_m is false (line 29) leading to chaotic behaviour. If the call is accepted, then @functional in lines 30–32 is evaluated. There, it is ensured that the result value to be emitted is equal to the sh_data.

The memory components receive event set via the consumer port pmem_c. The reception of event set is delayed until the promised condition is met in @interaction. Since it is specified as true in line 37, the set event can be received at any instance. Upon reception of the event set, @functional in lines 38–41 is evaluated. As the functional pre-condition (requires) is true, the event’s post-condition initialised_m is ensured and sh_data is set to data_arg.

```plaintext
1 component user
2  bool initialised_u = false;
```
int data;
required port puser_r {
  @interaction{
    promises: true; }
  @functional{
    promises: true;
    requires: true;
    ensures: data = \result; }
  int get();
}

emitter: port puser_e {
  @interaction{
    promises: true; }
  @functional{
    promises: data_arg = 7;
    requires: !initialised_m; }
  functional
    set(int data_arg);
  } // component ends

component memory {
  bool initialised_m = false;
  int sh_data = 0;
  provided port pmem_p{
    @interaction{
      accepts: initialised_m;
      also:
        rejects: !initialised_m; }
    @functional{
      requires: true;
      ensures: \result = sh_data;}
    int get();
    } // port ends
}

consumer port pmem_c{
  @interaction{
    accepts: true; }
  @functional{
    requires: true;
    ensures: initialised_m = true
    && sh_data = data_arg; }
  set(int data_arg);
  } // port ends
}

connector sharedData{
  userRole{pv_user},memoryRole{pv_mem}]
  role userRole{
    bool initialised_u = true;
    required port_variable pv_user_r{
      @interaction{
        promises: when(initialised_u); }
      int get();
    };
    emitter port_variable pv_user_e{
      @interaction{
        promises: true;
        ensures: initialised_u = true; }
      set(int data_arg);
    } // port-variable ends
  }
  role memoryRole{
    bool initialised_m = true;
    provided port_variable pv_mem_p{
      @interaction{
        accepts: when(initialised_m);}
      int get();
    };
    consumer port_variable pv_mem_c{
      @interaction{
        accepts: true;
        ensures: initialised_m = true; }
      set(int data_arg);
    } // port-variable ends
  }
}

channel async user2memory_m(pv_user_r, pv_mem_p);
channel async user2memory_e(pv_user_e, pv_mem_c);

Listing 1: Shared-data Access in our ADL

Shared-Data Connector Type The connector type sharedData, specified in lines 45–79, serves as a mediator between users and memory. It essentially avoids memory from behaving chaotically. In the sharedData, two roles are specified, userRole in lines 47–60 and memoryRole in lines 61–74, where the former is assumed by the user component instances and the latter by the memory instances.

The interaction behaviour of the event ports is constrained via the respective port-variables (i.e., pv_user_e of the userRole and pv_mem_c of the memoryRole) so that emission/receipt of event set ensures that the respective role data initialised_u/m is set to true. Furthermore, the pv_mem_p in lines 63–67 and pv_user_r in lines 49–53 constrain the memory provided port, so that it delays calls for method get until the shared data is initialized, and the user required port so that it does not make calls for get until the data is initialized. Therefore, the corresponding ports of the user and memory components playing the roles interact in a way that an event set is always executed before method get is called. This attempts to access uninitialized data and the resulting chaotic behaviour, as well as deadlocks.

In lines 75–78, there are two channels specified to indicate which role port-variable communicates with which other role port-variable.

Finally, the matching between connector roles and components is performed via the connector parameters, as specified in line 46. At configuration time, when sharedData is instantiated, the user and memory component instances and their ports are passed as its parameters.

4 Discussion

As can be seen by Listing 1, the notation used for DbC in our ADL follows a JML-like syntax, which is
familiar to practitioners. This should help facilitate its adoption as it does not require a steep learning curve – one should be able to use it with minimal training. At the same time, it introduces connector constructs that are essentially (decentralized) algorithms (as protocols are). So practitioners who have experienced the distinction between say C++ classes and C++ generic algorithms should feel comfortable with the distinction between components and connectors as made here. Apart from π calculus’ ability to send channels as messages, which we do not support, our ADL should allow architects to express what they can express now with ADLs based on process algebras.

All constraints in our ADL are local, expressed on local component/role data and parameters. This ensures realizability and makes reasoning about the effects of actions easier. Data are encapsulated, so there are no aliasing problems, and concurrency is controlled through the ports – each port is a concurrent unit, thus ensuring that actions of a port are mutually exclusive. As event emission/consumption and method servicing (at provided ports) are atomic, architects need only guarantee (and verify) that method calling (at required ports) will not lead to data race conditions.

The type of constraints an architect uses depends on the tool support they have. If for example they express something like $0 \leq x + y + z \leq 25$ then their tool would probably need to use an SMT solver or theorem prover and be helped by the architect whenever it cannot prove or disprove a claim. However, we believe that in practice one can constrain the type of expressions used, so as to enable automated verification through a model-checker, without losing much. The aforementioned constraint can easily be expressed as $x \in [0, 25] \land y \in [0, 25 - x] \land z \in [0, 25 - x - y]$, which makes it easy for a model-checker to establish, as it introduces an order for selecting values for variables. By constraining the form of expressions we enable better tool support for practitioners and make it possible for these tools to be fully automated.

5 Conclusions

This paper presents a new ADL that aims at resolving three main problems in current ADLs: (i) advanced formal notations used for specifying behaviours, (ii) lack of support for complex connectors, and (iii) potentially unrealizable designs.

In response to these problems, our ADL follows a Design-by-Contract approach for specifying behaviour, using a JML-like syntax as much as possible. Thus, practitioners should be able to use it with minimal training and find it not much different to other languages that they might have used, such as JML. Furthermore, architectural connectors in our ADL can be used to specify either simple interconnection mechanisms or complex interaction protocols. So, large and complex systems can be specified at a high level as components interacting via complex interaction protocols, like someone may use C++ classes and C++ generic algorithms to structure a system so as to avoid re-coding generic algorithm fragments inside the classes. Finally, connectors are specified as a set of decentralized, independent roles assumed by the participating components. There is no glue-like centralized element in connector specifications or elsewhere – all constraints are expressed on local state. This leads to architectural designs that are realizable by construction and whose verification is decidable (when restricted to a finite number of elements) through model-checking.

Acknowledgements

This work has been partially supported by the EU project FP7-257367 IoT@Work – “Internet of Things at Work”.

REFERENCES


