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 [1] 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 [2] 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

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

[2] Perhaps 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 when they do support user-defined, complex connectors, they do so in a way that can lead to unrealizable designs. This is because,

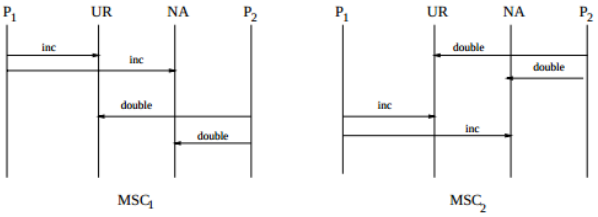

Figure 1: A nuclear power plant's unrealizable MSCs [Alur et al., 2003]

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

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

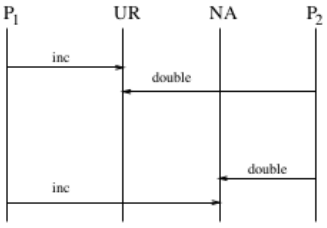

Figure 3: An implied bad scenario in Alur's plant [Alur et al., 2003] (*hidden* by Wright's glue specification)

ADLs of this category are following the approach initiated by Wright [Allen and Garlan, 1997] and require connectors to include a glue element. As stated in [Allen and Garlan, 1997], a connector role specifies the "*obligations of [a] component participating in the interaction*" and a glue specifies "*how the activities of the [. . .] roles are coordinated.*" – a connector glue is supposed to be more than simple definition/use relationships. The fact that the glue can introduce inter-role interaction constraints is deeply problematic however, because these constraints cannot always be implemented in a decentralized manner by the components that assume the connector roles, as these can only observe their local state [Tripakis, 2001, Tripakis, 2004]. In fact, it has been shown that the problem of deciding whether a glue can be realized is undecidable [Tripakis, 2001, Tripakis, 2004, Alur et al., 2003, Alur et al., 2005], so there is no general algo-

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 protocol 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 interaction 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 processes in a nuclear reactor. After the update operations, the amounts of UR and NA must be equal to avoid nuclear accidents, for which reason we wish to allow only the sequences shown in Figure 1. The interaction of the two clients with the NA and UR variables, 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 constraints interactions by requiring that we only allow $UR.increment \rightarrow NA.increment$ or $UR.double \rightarrow NA.double$. This specification is however unrealizable 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 interactions 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 connectors;
- 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 protocols. These can then be instantiated as many times as needed, allowing architects to simplify the specifications of their components and easily reuse the specification of complex protocols.

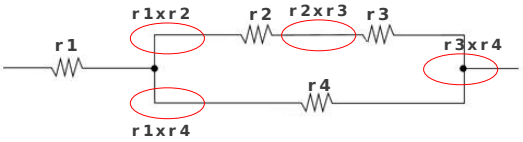

Figure 4: Simple connectors in circuits

$$\rightarrow (r_1, ||(\rightarrow (r_2, r_3), r_4))$$

Figure 5: Complex connectors in circuits

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

ier to understand the system and to analyze its overall resistance by taking advantage of the connector properties as:

$$\begin{aligned} r_{\rightarrow(r_1,\|(\rightarrow(r_2,r_3),r_4))} &= r_1 + r_{\|(\rightarrow(r_2,r_3),r_4)} \\ &= r_1 + \frac{1}{\frac{1}{\rightarrow(r_2,r_3)} + \frac{1}{r_4}} \\ &= r_1 + \frac{1}{\frac{1}{r_2+r_3} + \frac{1}{r_4}} \end{aligned}$$

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 [Bjørner 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

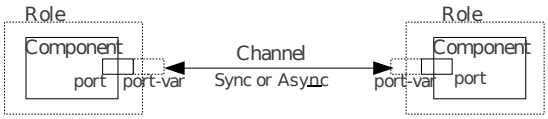

Figure 6: Connectors with roles wrapping components

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*.

```
1 component user{
2   bool initialised_u = false;
```

```
3    int data;
4    required port puser_r {
5       @interaction{
6        promises: true; }
7       @functional{
8        promises: true;
9         requires: true;
10        ensures: data = \result; }
11      int get();
12   };
13   emitter: port puser_e {
14      @interaction{
15       promises: true; }
16      @functional{
17       promises: data_arg = 7;
18       ensures: intialised_u = true;  }
19      set(int data_arg);
20   }; //port ends
21 }; // component ends
22 component memory {
23   bool initialised_m = false;
24   int sh_data = 0;
25   provided port pmem_p{
26      @interaction{
27       accepts: initialised_m;
28      also:
29       rejects: !initialised_m; }
30      @functional{
31       requires: true;
32       ensures: \result = sh_data;}
33      int get();
34   }; // port ends
35   consumer port pmem_c{
36      @interaction{
37       accepts: true; }
38      @functional{
39       requires: true;
40       ensures: intialised_m = true
41                && sh_data = data_arg; }
42      set(int data_arg);
43   };  // port ends
44 }; // component ends
45 connector sharedData(
46      userRole{pv_user},memoryRole{pv_mem})
47   role userRole{
48      bool initialised_u = true;
49      required port_variable pv_user_r{
50         @interaction{
51          promises: when(intialised_u); }
52         int get();
53      };
54      emitter port_variable pv_user_e{
55         @interaction{
56          promises: true;
57          ensures: initialised_u = true; }
58         set(int data_arg);
59      };  // port-variable ends
60   };  // role ends
61   role memoryRole{
62      bool initialised_m = true;
63      provided port_variable pv_mem_p{
64         @interaction{
65          accepts: when(intialised_m);}
66         int get();
67      };  // port-variable ends
68      consumer port_variable pv_mem_c{
69         @interaction{
70          accepts: true;
71          ensures: initialised_m = true; }
72         set(int data_arg);
73      }; // port-variable ends
74   };  // role ends
75   channel async user2memory_m(pv_user_r,
76                               pv_mem_p);
77   channel async user2memory_e(pv_user_e,
78                               pv_mem_c);
79 }; // connector ends
```

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 avoids 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 $x \in [0,25] \wedge y \in [0,25-x] \wedge z \in [0,25-x-y]$, 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 [3].

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.

[3] *Less is more.*

Acknowledgements

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

REFERENCES

Allen, R. and Garlan, D. (1997). A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249.

Alur, R., Etessami, K., and Yannakakis, M. (2003). Inference of message sequence charts. *IEEE Trans. Software Eng.*, 29(7):623–633.

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

Bergstra, J. A., Ponse, A., and Smolka, S. A., editors (2001). *Handbook of Process Algebra*. Elsevier.

Bjørner, D. and Jones, C. B., editors (1978). *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer.

Canal, C., Pimentel, E., and Troya, J. M. (1999). Specification and refinement of dynamic software architectures. In Donohoe, P., editor, *WICSA*, volume 140 of *IFIP Conference Proceedings*, pages 107–126. Kluwer.

Chalin, P., Kiniry, J. R., Leavens, G. T., and Poll, E. (2006). Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *FMCO'05 – Formal Methods for Comp. and Obj.*, volume 4111 of *LNCS*, pages 342–363. Springer.

Dashofy, E. M., van der Hoek, A., and Taylor, R. N. (2002). An infrastructure for the rapid development of xml-based architecture description languages. In Tracz, W., Young, M., and Magee, J., editors, *ICSE*, pages 266–276. ACM.

Delanote, D., Baelen, S. V., Joosen, W., and Berbers, Y. (2008). Using aadl to model a protocol stack. In *ICECCS*, pages 277–281. IEEE Computer Society.

Feiler, P. H., Gluch, D. P., and Hudak, J. J. (2006). The Architecture Analysis & Design Language (AADL): An Introduction. Technical report, Software Engineering Institute.

Garlan, D., Allen, R., and Ockerbloom, J. (1995). Architectural mismatch or why it's hard to build systems out of existing parts. In *ICSE*, pages 179–185.

Garlan, D. and Shaw, M. (1993). An introduction to software architecture. In Ambriola, V. and Tortora, G., editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore. World Scientific Publishing Company. Also appears as SCS and SEI technical reports: CMU-CS-94-166, CMU/SEI-94-TR-21, ESC-TR-94-021.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

Issarny, V., Bennaceur, A., and Bromberg, Y.-D. (2011). Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability. In Bernardo, M. and Issarny, V., editors, *SFM*, volume 6659 of *Lecture Notes in Computer Science*, pages 217–255. Springer.

Luckham, D. C. (1996). Rapide: A language and toolset for simulation of distributed systems by partial orderings of events. Technical report, Stanford University, Stanford, CA, USA.

Magee, J. and Kramer, J. (1996). Dynamic structure in software architectures. In *SIGSOFT FSE*, pages 3–14.

Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., and Tang, A. (2013). What industry needs from architectural languages: A survey. *IEEE Transactions on Software Engineering*, 99(PrePrints):1–25. DOI: 10.1109/TSE.2012.74 .

Medvidovic, N. and Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Trans. Software Eng.*, 26(1):70–93.

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

OMG (2006). Corba component model 4.0 specification. Specification Version 4.0, Object Management Group.

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

Plasil, F. and Visnovsky, S. (2002). Behavior protocols for software components. *IEEE Trans. Software Eng.*, 28(11):1056–1076.

Schmidt, H., Poernomo, I., and Reussner, R. (2001). Trust-by-contract: Modelling, analysing and predicting behaviour of software architectures. *J. Integr. Des. Process Sci.*, 5(3):25–51.

Schreiner, D. and Göschka, K. M. (2007). Explicit connectors in component based software engineering for distributed embedded systems. In *Proceedings of the 33rd conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM '07, pages 923–934, Berlin, Heidelberg. Springer-Verlag.

Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., and Zelesnik, G. (1995). Abstractions for software architecture and tools to support them. *IEEE Trans. Software Eng.*, 21(4):314–335.

Tripakis, S. (2001). Undecidable problems of decentralized observation and control. In *Proc. of the 40th IEEE Conf. on Decision and Control*, volume 5, pages 4104–4109, Orlando, FL, USA. IEEE.

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