Architectural Specification and Analysis with \textit{XCD} \\
\textit{The Aegis Combat System Case Study}

Mert Ozkaya and Christos Kloukinas \\
\textit{Department of Computer Science, City University London, London, U.K.}

Keywords: Design-by-Contract, ProMeLa, Architectural Modelling, Formal Analysis.

Abstract: Despite promoting precise modelling and analysis, architecture description languages (ADLs) have not yet gained the expected momentum. Indeed, practitioners prefer using far less formal languages like UML, thus hindering formal verification of models. One of the main issues with ADLs derives from process algebras which practitioners view as having a steep learning curve. In this paper, we introduce a new ADL called \textit{XCD} which enables designers to model their software architectures through a Design-by-Contract approach, as for example in the Java Modelling Language (JML). We illustrate how \textit{XCD} can be used in architectural modelling and analysis using the Aegis combat software system.

\section{INTRODUCTION}

Architectural modelling and analysis of complex software systems has always been a crucial aspect of system development for two reasons. First modelling of architectures enables a highly abstracted view of systems making their complexity tractable. Second, models, if specified formally, can be analysed mechanically thus enabling the detection of errors long before the implementation phase.

Unified Modelling Language (UML) (Rumbaugh et al., 1999) gained wide popularity in modelling design of software systems. Despite partially serving the first reason of modelling, i.e, tractability of large and complex systems, it however does not do so for the second reason – analysis of models for early error detection. This is because UML lacks in formally precise semantics thus leading to informal and ambiguous models which cannot be mechanically analysed. Apart from UML, since nineties, several architecture description languages (ADLs) (Medvidovic and Taylor, 2000) have been proposed. Unlike UML, many of these ADLs are based on formally precise semantics, so as to enable analysis of software architectures too.

Despite enabling modelling and analysis, ADLs unfortunately have not gained the expected momentum among practitioners. As stated in (Malavolta et al., 2012), this could be due to the steep learning curve these languages require. According our ADL study (Ozkaya and Kloukinas, 2013a), indeed ADLs are based on process algebras (e.g., FSP (Magee and Kramer, 2006) and CSP (Hoare, 1978)) which most practitioners are unfamiliar with.

In this paper, we present our new ADL called \textit{XCD} that aims at architectural modelling and analysis in a more practitioner-friendly manner. To this end, \textit{XCD} is based on widely known Design-by-Contract (DbC) approach (Meyer, 1992). So, just like Java Modelling Language (JML) (Chalin et al., 2006), behaviour of architectural elements is specified with contracts, but, in a more systematic and comprehensive way. Indeed, we consider a number of extensions to DbC facilitating the specification of components and connectors. To enable formal verification of contractual software architectures, we provide formal mapping of \textit{XCD} constructs to SPIN’s formal ProMeLa language (Holzmann, 2004) which is not only supported by a powerful model checker but also developer-friendly language resembling C programming.

\section{XCD ADL via Aegis CASE STUDY}

Figure 1 depicts the meta-model of the \textit{XCD} ADL\footnote{Although \textit{XCD} supports emitter/consumer ports too for emitting/consuming asynchronous events, we have not mentioned them herein due to lack of space.}. \textit{XCD} offers two main architectural elements, \textit{components} and \textit{connectors}. As introduced in (Ozkaya and Kloukinas, 2013b), components are used to specify the abstractions of computational units and connectors the interaction protocols for the interact-
In the rest of this section, using the Aegis combat software system, we illustrate the contractual specifications of components and connectors. The Aegis system has been developed for navy ships to make them capable of controlling their weapons against enemies. The Aegis has firstly been tackled by Wright (Allen, 1997) in (Allen and Garlan, 1996), which can informally be specified as Figure 2 comprising a set of components interacting with each other. The Experiment Control, at the top of the diagram, essentially provides linked components the information obtained via sensors. The track information is, for instance, required by the Track Server which stores it and provides other components (Doctrine Validation and Geo Server) the location information about the enemies operating around the ship. The Doctrine Authoring requires doctrine rules from the Experiment Control and provides them to the other components (Geo Server, Doctrine Validation, and Doctrine Reasoning) that require rules to take actions. Using the doctrine rules and track information from its environment, the Geo Server provides to the Doctrine Reasoning the precisely calculated region information for enemies. Lastly, the Doctrine Reasoning makes the decision of which task(s) to take against the enemies.

2.1 XCD Specification of Aegis

We specify three types of primitive components to be able model the components depicted in Figure 2. These are client, server, and mixedComp types. Each component comprises a set of data variables representing their state and ports representing the points of interactions with their environment. To model the interactions between the components, we also specify a connector type, client2server which is specified as a set of roles played by the components representing their interaction protocols. It also has built-in connectors establishing the communication links between the component ports. Finally, we specify a composite component type aegis configuration which represents a configuration of client, server, and mixed-Comp components interacting via the client2server connectors.

2.1.1 Client Component

Listing 1 gives the client component type specification, from which client components are instantiated (i.e., Doctrine Validation and Doctrine Reasoning) that only require services of server components to be able perform their tasks. The state of the component is specified with two data variables (lines 2-3): the data holds any information maintained by the component and openedConns holds the number of client ports that open their connections with their servers (by making call for the method open).

Listing 1: Client Type Specification.

```java
1 component client (int numOfPorts)
2 int data = 0,
3 int openedConns = 0;
4 required port service[numOfPorts]{
5     @interaction{
6         waits: openedConns < numOfPorts;
7     }
8     @functional{
9         ensures: post(openedConns)++;
10     }
11     void open();
12 }
13 @interaction{
14     waits: openedConns > 0;
15 }
16 @functional{
17     ensures: post(openedConns) == false
18         && post(openedConns)--;}
19 void close();
20 }
21 }
```
The client includes an array of required ports service (lines 4–22) for making method calls to the connected server ports. The size of the service is specified as the component parameter numOfPorts. Each port of the service includes three methods that can be requested from the connected server ports: open, close, and request. Methods are augmented with @interaction and @functional contracts comprising a set of constraints to satisfy the ultimate goal: a client can make request for a service only after it opens the connections to all of its connected servers.

The methods of a required port firstly get their parameters promised via their functional constraint’s promise expression sequence (FCPromises in Figure 1). However, since none of the methods in the port service[@]² has parameters, they do not have FCPromises. So, if the port interaction constraint guard (ICWaits) on the method open (lines 5–10 in Listing 1) is satisfied, i.e., the data openedConns is less than the component parameter numOfPorts, the request is made for the open. When the respective response is received, if the pre-condition of the functional constraint (FCRequires) is met, its post-assignment (FCEnsures) may then be performed. Since FCPromises is not specified (i.e., therefore, true) for the open, its FCEnsures increments the openedConns.³ Note that a while FCPromises assigns promised values to method parameters, the FCEnsures assigns new values to data variables. The other port method close (lines 12–16) is requested if the data openedConns is greater than zero (ICWaits). Upon receiving the response, FCEnsures decrements the openedConns data directly, without any pre-conditions (FCRequires). For the method request (lines 18–21), it is called if the openedConns data is equal to the numOfPorts parameter (ICWaits) indicating that all the port of the client opened their connections. Upon calling the method and receiving the response, FCEnsures updates the data directly (with no FCRequires) assigning it the received result from the connected server port.

2.1.2 Server Component

Listing 2 gives the server type specification from which server components (i.e., Experiment_Control) instantiated that provide services to the client components. The component state is specified with two data variables (lines 2–3): the opened array variable holding for each port true if a method-call open is received (false otherwise) and the data holding the information maintained by the server.

```
Listing 2: Server Type Specification.

1 component server (int numOfPorts) {
2     bool opened [numOfPorts] = false;
3     int data = 1;
4     provided port service [numOfPorts] {
5         @interaction waits: opened[@] ==> false;
6         @functional
7             ensures: post (opened[@]) := true;
8         void open () {
9             @functional
10             ensures: post (opened[@]) := true;
11             void close () {
12                 @functional
13                 ensures: post (opened[@]) := false;
14             }
15             @interaction waits: opened[@] == true;
16             @functional ensures: \result := data;
17             int request () {
18             }
19         }
20     }
21 }
```

The server includes an array of provided ports service (lines 4–18) each of which is to receive method-calls from a required port of the client. Note that the server ports are connected with the client ports via the connectors which we will discuss shortly. The interaction (ICWaits) and functional constraints (FCRequires pre-condition and FCEnsures post-assignment) attached to the service port methods serve to meet the goal: requests for services can be received after the respective connection is opened. The method open of the port service[@] (lines 5–8) is delayed by ICWaits until the data opened[@] is false. Upon receipt of the request, if the FCRequires is satisfied, FCEnsures post-assignment is performed. So, since FCRequires is not specified for the method open (i.e., therefore, true), FCEnsures assigns true to the data opened[@] directly. Then, the response is sent back with no result due to the method open holding void type. For the method close (lines 10–13), its requests are delayed until the opened[@] is true. When received, FCEnsures assigns false to the same data directly again, and, the response is sent. The calls for the method request (lines 15–17) are also delayed until the data opened[@] is true. Then, FCEnsures assign the value of data to result that is sent back to the client port. Note that provided ports process method operations atomically; that is, upon receiving a request successfully, the response is to be sent back immediately.
2.1.3 MixedComponent Component

Listing 3 gives the mixedComp component type specification which represents those acting both as server and clients (i.e., Doctrine_Authoring, Track_Server, and Geo_Server) That is, they not only require services from outside, but also offer too.

The component state is represented via three data variables (lines 2–3): the server_opened array variable holds for each server port true if a method-call is received for open. The data openedConns holds the number method-calls made for open via the client ports. Finally, the data holds any information maintained by the component.

Listing 3: Mixed-Component Type Specification.

```c
component mixedComp | int CSize, int SSize| {
    bool server_opened[CSize] = false;
    int openConns = 0, data = 3;
    required port client[CSize]{
        @interaction waits: openedConns < CSize;
        @functional ensures: post(openedConns)++;}
    void open();
    @interaction waits: openedConns >= SSize;
    @functional ensures: post(data) := result;
    int request();
}

provides port server[SSize]{
    @interaction req{
        waits: openedConns == CSize && server_opened[8] == false;
        @functional req{
            ensures: post(server_opened[8]) := true;
            void open();
        }
        @interaction req{
            waits: openedConns == CSize && server_opened[8] == true;
            @functional req{
                ensures: post(server_opened[8]) := false;
                void close();
            }
            @interaction req{
                waits: server_opened[8] == true && openedConns == CSize;
                @functional res[ensures: \{result := data;]
                int request();
            }
        }
    }
}
```

The mixedComp has an array of required ports client (lines 4–16) with the size equal to the component parameter CSize. Herein, the ports client[@] behave in the same way as those of the client component type aiming to meet the same goal.

There is also an array of provided ports server specified (lines 17–37) with the size equal to the SSize parameter. The ports server[@] comprises complex methods, i.e., upon successful receipt of the request, the response does not have to be sent immediately, as the component may need to require some services via its client ports, to calculate the response result. In complex method specifications, interaction and functional contracts are split into two atomic parts: the request part (+_req) evaluated upon the receipt of the method request and the response (+_res) part evaluated when the port is ready to send the method response. The methods of the server ports are attached with such contracts to meet the goal: the ports server[@] may not operate until all the client ports open their connections. To this end, the request ICWaits for the method open (lines 18–23) delays the method request until the openedConns is equal to the CSize and the respective data server_opened[@] is false. Upon receiving the method request, the request FCEnsures assigns true to the server_opened[@] directly as the request FC Requires is not specified (i.e., therefore, true). There is no constraints specified for the method response, indicating that it may be sent back randomly at any time after receiving the request. The server port operates the method close (lines 25–30) in the opposite way of the open, receiving the request when the server_opened[@] is true, and the request FC Ensures of the close assigns false to the same data. For the method request, its request is accepted when the port’s already received a call for open (i.e., server_opened[@] is true) and all of the clients’ve called open (i.e., the openedConns being equal to CSize is true). Unlike the open and close, the method request (lines 32–36) is attached with a @functional res that includes FC Ensures to assign the value of data to the result directly. Indeed, its return type is int requiring a result to be sent back in the response.

2.1.4 Client2Server Connector

The connectors of the Client2Server type essentially represent the complex interactions between client and server components. The Client2Server is specified with two roles and one built-in connector. The role client is played by the participating client component, while the role server by the participating server component. Note also that the mixedComp components can play either of the roles in their interaction. Each role comprises data-variables representing their local state and a set of port-variables representing the ports of the components. The port-variables attach the port methods with @interaction contracts that comprise interaction constraints further constraining the method behaviours. Unlike port interaction constraints, the port-variable
interaction constraints may update the role state data too via their post-assignments and, thereby, comprising a pair of RICWaits and RICEnsures in Figure 1.

The role client (lines 3–18) imposes on the client an interaction protocol that the client may not request a service of the server before opening the connection of the respective server. To this end, the role client has a single data opened. Its port-variable service imposes interaction constraints on the methods. So the method open of the associated port may not be called until the opened is false (RICWaits). Upon the satisfaction of the interaction constraints, the respective post-assignment (RICEnsures) assigns true to the same data. This then allows the methods request and close to be called, whose role interaction guards delay them until the opened is true. Note also that the interaction constraint on the close has RICEnsures that assigns false to the opened allowing the method open to be called again.

The role server (lines 19–25) does not impose any interaction constraints on its port-variable methods allowing the associated component ports to receive method requests in any order.

There is a built-in connector specified (lines 26–27) which represents the communication link between the role port-variables. Therefore, the component port represented by the service port-variable of the server role may communicate with the component port represented by the service port-variable of the client role.

The matching between component ports and role port-variables are performed when the connector is instantiated in composite components and components are passed as parameters (see Section 2.1.5).

Listing 4: Client2Server Type Specification.

```java
connector client2server(client (service), server (service)){
  role client{
    bool opened := false;
    required port_variable service{
      @interaction
      waits: opened == false;
      ensures: post(opened) := true;
    }
    void open();
    @interaction
    waits: opened == true;
    ensures: post(opened) := false;
    void close();
    @interaction
    waits: opened == false;
    int request();
  }
  role server{
    provided port_variable service{
      void open();
      void close();
      int request();
    }
  }
}
```

2.1.5 Aegis Configuration Component

Unlike the client and server component types, the aegis_configuration is a composite type that consists of component and connector instances representing an architectural configuration for the Aegis.

The aegis_configuration includes a component instance for each component depicted in the Figure 2. There is also a set of connector instances specified that represent the interactions between those component instances. Note that the connector instances receive as parameters the participating components and their ports (component{portList}). Indeed, this is how connector roles and their port-variables are associated with the components and their ports.

Listing 5: Composite Component Type Specification.

```java
component aegis_configuration(){
  component server e_Ctrl(3);
  component mixedComp d_Valid(1,3);
  component client d_Auth(3);
  component mixedComp t_Srvr(1,3);
  component mixedComp g_Srvr(2,1);
  component client2server cs_1{
    d_Auth{client[0]}, e_Ctrl{service[1]};
  }
  component client2server cs_2{
    d_Auth{client[0]}, e_Ctrl{service[0]};
  }
  component client2server cs_3{
    d_Auth{server[0]}, e_Ctrl{service[1]};
  }
  component client2server cs_4{
    d_Auth{client[0]}, e_Ctrl{service[2]};
  }
  component client2server cs_5{
    d_Auth{service[1]}, d_Auth{server[0]};
  }
  component client2server cs_6{
    d_Auth{server[0]}, d_Auth{server[1]};
  }
  component client2server cs_7{
    d_Auth{client[0]}, d_Auth{server[2]};
  }
  component client2server cs_8{
    d_Auth{client[0]}, t_Srvr{server[1]};
  }
  component client2server cs_9{
    d_Auth{client[1]}, t_Srvr{server[2]};
  }
  component client2server cs_10{
    d_Auth{service[2]}, g_Srvr{server[0]};
  }
}
```

3 FORMAL MODELLING/REPRESENTATION

We implemented the semantics of XCD in SPIN’s formal ProMeLa language (Holzmann, 2004) which enables formal reasoning about XCD specifications. We
chose ProMeLa due to two main reasons. XCD semantics match those of ProMeLa well, facilitating the transformation from XCD constructs to ProMeLa. Finally, ProMeLa is supported by a powerful model checker whose implementation is open. The rest of this section summarises at an abstract level how components and connectors can be mapped to ProMeLa notation.

3.1 Transforming XCD to ProMeLa

3.1.1 Composite Component Transformation

A composite component $c$ is mapped as shown in Listing 6. A request and response asynchronous channel arrays are produced from each sub-component provided port (lines 3–6). The arrays include a distinct channel for each required port connected to the provided port via built-in connectors. As shown in the port semantics, these channels are used by the provided ports and the connected required ports of other sub-components, to transfer request and response messages. Then, a process is declared for the composite component $c$ (lines 7–10) that executes via ProMeLa’s run operator the processes corresponding to its sub-components and thereby enables their concurrent interaction.

Listing 6: Composite Component semantics in ProMeLa.

```promla
forall subcomp ∈ c.components
forall pp ∈ subcomp.ProvidedPorts
chan requestChannel[pp.numOfConns]
= [1] of (Request);
chan responseChannel[pp.numOfConns]
= [1] of (Response);
proctype c.ID()
forall subcomp ∈ c.components
run subcomp.ID();
```

3.1.2 Primitive Component Transformation

A primitive component $c$ is mapped as shown in Listing 7. The process declaration comprises a set of variable declarations corresponding to component data and the data of the role(s) the component assumes (lines 3–5). Each component data is mapped to two variables, one for storing the pre-state and the other for its post-state value. Besides data variables, a repetition construct (i.e., do..od) is included (lines 7–9) that repeatedly executes a set of guarded action sequences for the component port behaviours.

Listing 7: Primitive Component semantics in ProMeLa.

```promla
proctype c.ID()
forall data ∈ c.Data ∪ ∪ role ∈ c.roleSet.role.Data
data.type data.pre_state = data.initialValue;
data.type data.post_state;
Start:
do
.......
od
```

Listing 8 shows that each method of a required port is transformed into two guarded atomic actions. The request action (lines 3–8) is guarded by the assign_params method that selects method parameters via the functional constraint FCRequires. The chosen parameters satisfy the port interaction constraint ICWaits and the role interaction constraint ICWaits guards, the request message is written to the requestChannel (line 7), and, the port holds the lock. Otherwise, control moves back to the beginning of the component repetition construct executing the port behaviour (line 6 in Listing 7). The response action (lines 12–20) is guarded by the responseChannel which is satisfied if the channel includes a response message and the port holds the lock. Upon reading the response, if the functional constraint pre-condition is met (FCRequires), the assign_data method is used that updates component role data via the constraint post-assignments (FCEnsures and RICEnsures respectively).

Listing 8: Required Port semantics in ProMeLa.

```promla
forall rp ∈ c.RequiredPorts
forall m ∈ rp.Methods::atomic|
assign_params (FCPromises) →
if ICWaits ∧ ∨ role ∈ c.roleSet.RICWaits →
requestChannel ! m, m.parameters;
lock = true;
::else → goto Start
fi
::atomic|
responseChannel ? m, m.result ∧ Alock→
if ::FCRequires →
assign_data (FCEnsures);
forall role ∈ c.roleSet
assign_data (RICEnsures);
fi
```

The methods assign_params and assign_data represent an iterative execution of ProMeLa’s select statement to implement each assignment of the inputted sequence.
Provided port methods are each mapped to a single atomic action as shown in Listing 9. The action is guarded by the requestChannel which is satisfied if there exist a request message that meets the port interaction constraint (ICWaits) and the role interaction constraint guards (RICWaits). Upon satisfaction of the guards, the component and role data are updated through the constraint post-assignments (lines 6-8), and, subsequently, the result is written to the responseChannel (line 9).

Listing 9: Provided Port semantics in ProMeLa.

```prople
    for all pp ∈ c. ProvidedPorts
    for all m ∈ pp.Methods
        :: atomic |
            requestChannel ? m, m. parameters
            ICWaits ∧ ∃ roleSet RICWaits →
            assign_data (FCEnsures);
        for all role ∈ c. roleSet
            assign_data (RICEnsures);
        responseChannel ! m, m. result
        
```

Complex methods of provided ports are each mapped to two separate atomic actions as shown in Listing 10, one for receiving the request and another for sending the response. Just like the simple method, the top request action (lines 3–10) is guarded by the requestChannel which is satisfied if there exists a request that meets request interaction constraint guards. Then, the component and role data are updated via the constraint post-assignments on the method request; and, the request flag is set to true. The bottom response action (lines 11–19) is executed if the interaction guards on the method response part are met and the method request has been received. Then, again, the component and role data are updated via the post-assignments on the method response. Finally, the response is written to the responseChannel.

Listing 10: Provided Port semantics in ProMeLa – Complex Methods.

```prople
    for all pp ∈ c. ProvidedPorts
    for all m ∈ pp. ComplexMethods
        :: atomic |
            requestChannel ? m, m. parameters
            ICWaitsreq ∧ ∃ roleSet RICWaitsreq →
            assign_data (FCEnsuresreq);
        for all role ∈ c. roleSet
            assign_data (RICEnsuresreq);
        requestedreq := true;
        
        :: atomic |
            ICWaitsreq ∧ ∃ roleSet RICWaitsreq,
            ∧ requestedreq := false;
            responseChannel ! m, m. result
        )
```

4 AUTOMATED FORMAL VERIFICATION

A tool is available (Ozkaya, 2013) to automatically translate XCD specifications into formal ProMeLa models. These produced models can be directly verified by the SPIN model checker.

Having transformed the Aegis specification in Section 2.1 into a ProMeLa model via the tool, we were able to formally verify it via the model checker. Table 1 shows the verification results – no deadlock was identified.

Table 1: Verification results for Aegis.

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>524</td>
<td>16734505</td>
<td>90863348</td>
<td>7024</td>
<td>62.7</td>
</tr>
</tbody>
</table>
```

Formal verification of software architectures is highly crucial for many reasons. Firstly, it aids in detecting design errors, e.g., incompatible component behaviours, causing deadlocks. If such issues were left to the implementation stage, the cost of correcting the errors would highly increase. Furthermore, different design choices can easily be explored that enables to determine the optimal one. Indeed, the current Aegis model in Section 2.1 includes in components a single port that has all three methods (open, close, and request). However, this choice of design minimises the level of concurrency. In XCD, each port operates its method sequentially while the ports are operated concurrently by the components. So, designers may wish the components to operate port methods concurrently. In such a case, a distinct port is created per method. That is, client and server have three ports each including a unique method. When we analyse our modified model with this design choice, the state-vector size nearly doubles as shown in Table 2; indeed, fewer number of states could be stored in the same amount of memory. This is because each newly added provided port introduces extra communication channels which causes the state-vector size to grow. Thus, while maximised concurrency may be a desired choice for designers, it requires greater state space and memory for formal verification.

Table 2: Verification results – Maximised Concurrency.

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>8350876</td>
<td>47805585</td>
<td>7024</td>
<td>48.1</td>
</tr>
</tbody>
</table>
```

*Our verification is limited with 7024MB of memory; for a full verification, more memory seems to be required.*
5 EVALUATION – Xcd Vs Wright

As aforementioned, Aegis has also been specified and analysed with Wright (Allen, 1997). We base our comparison with Wright’s Aegis specification on three key features that, we believe, affect designer’s choice in choosing an ADL to use.

Realisable connectors As mentioned in our ADL study (Ozkaya and Kloukinas, 2013a), Wright and those inspired from it include a glue in their connector structure which constrains the behaviour of the components globally. However, its global nature causes potentially unrealisable specifications for distributed systems, as explained in (Ozkaya and Kloukinas, 2013b). Indeed, the Aegis connector in (Allen and Garlan, 1996) includes such a glue for coordinating the client and server component behaviours. Therefore, Xcd connectors may only impose local constraints on the components via the roles; glues are not allowed. As shown in Section 2.1, the client-server connector has roles with local constraints only.

DbC-based behaviour specification To enable formal reasoning, Wright adopts an extended form of the CSP process algebra for behaviour specification. So Aegis is specified using CSP which is not found practical by practitioners (Malavolta et al., 2012). In Xcd, the behaviour of components and connectors are specified in an extended form of Design-by-Contract (DbC) approach which is more familiar to developers and easier to learn for them. For example, JML has been taught to undergraduate students for a number of years (Kiniry and Zimmerman, 2008).

SPIN’s Promela as the formal basis The semantics of Xcd are defined using ProMeLa which allows the use of a free and open tool for analysing architectures.

6 CONCLUSIONS

Xcd is a new ADL that extends Design-by-Contract approach and enables contractual architecture specification. While the functional and (minimal) interaction behaviours of components are specified via functional and interaction contracts respectively, the interaction protocols of connectors are via interaction contracts. Connectors in Xcd are decentralised and do not impose global constraints on the components. In this way, the common problem of connector-supporting ADLs – potentially unrealisable software architectures – is avoided. Xcd comes with a tool that translates architectures into ProMeLa models, which can be analysed by the SPIN model checker. As a further work, we are considering to improve our tool-set so that visual architecture specification can be possible. Designers might feel more comfortable if they could specify the structure of their components and connectors diagrammatically and attach contracts to them via a graphical user interface.

REFERENCES


