# Implementing Contract-based Software Architectures in Java: The Structural, Behavioural, and Interaction Design Decisions

Mert Ozkaya
*Department of Computer Engineering, Altinbas University, Istanbul, Turkey*

Keywords: Software Architecture, XCD, Code Generation, Java, Design-by-Contract.

Abstract: Architectural languages (ALs) have been so actively researched since the nineties and this leads to many ALs with different capabilities that have been proposed each day. However, most of the ALs ignore the specifications of the structural, behavioural, and interaction design decisions and do not support their analysis and development. The ALs that do support suffer from the process algebra based notation sets that practitioners do not prefer to work with. This issue is tackled in the paper by extending the Design-by-Contract based architectural language called XCD that supports the contractual specifications of the structural, behavioural, and interaction design decisions about software systems and their formal analysis. XCD is extended with a development support in Java so as to transform the contractual specifications in XCD into a complete Java program that considers all the structural, behavioural, and interaction decisions specified. So, practitioners will be able to specify non-algebraic, contractual specifications in XCD, formally analyse them, and produce the Java source-code that reflects the specified architecture completely and consistently. The produced Java code can also be tested for incompleteness and wrong use of component services automatically.



## 1 INTRODUCTION

Software architecture have been proposed essentially for describing large software systems in terms of independent computations (i.e., components) and their relationships with each other (i.e., connectors) (Clements et al., 2003). Many high-level design decisions can also be specified at the level of software architecture design. This includes the design decisions about the system structure, behaviour, interaction protocols, non-functional properties, concurrency, development, and deployment.

With the evolution of the model-based software engineering, architectural design decisions can be specified at different levels of abstractions, analysed for the properties of interest, and then transformed into software implementation code. Indeed, many architectural languages (ALs) (Medvidovic and Taylor, 2000; Ozkaya and Kloukinas, 2013) have been proposed, which offer different ways of modelling software architectures and provide many interesting features such as non-functional property support, analysis support, extensibility, code generation, graphical notation set, etc.

One of the most trendy topics of ALs is the formal analysis, i.e., the exhaustive checking of architectural models for some properties (e.g., deadlock and live-lock) or any user requirements. This allows for checking the behavioural and interaction design decisions for every state of the components and proving their correctness. Many of the ALs nowadays offer formally defined semantics and tool-support for transforming architectural models in these formalisms. By doing so, the exhaustive model checker tools of the formalisms can be used and the design errors can be detected before the design decisions are used in the low-level design and implementation stages.

While the analysis of the design decisions is so crucial, it is also desirable to transform the analysed specifications in some programming languages so as to reduce the development effort. This not only prevents the coding-from-scratch but also any inconsistencies between the architectural design decisions and the actual system implementation due to manual transformations. Developing code from software architecture also guarantees the communication integrity (Aldrich et al., 2002). That is, the implemented software systems consist of the components that can communicate only with those components that have been specified to communicate in the software architecture.



Ozkaya, M.
Implementing Contract-based Software Architectures in Java: The Structural, Behavioural, and Interaction Design Decisions.
DOI: 10.5220/0006568203460355
In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018)*, pages 346-355
ISBN: 978-989-758-283-7

## 2 MOTIVATION AND GOAL

There are tens of ALs that are existing in the literature, and, Malavolta et al. have kept a list of 125 different ALs (Ivano Malavolta, 2017). When those ALs have been analysed, many ALs (48%) have been found out to support the structural and behavioural design decisions[1]. So, these ALs support the *(i)* structural decompositions of software systems into independent components and connectors, *(ii)* the behavioural descriptions of how the components will change their states upon the events occuring in their environment (e.g., receiving/making method-calls) and *(iii)* the interaction protocols for connectors (if supported as part of the behaviour specifications). However, when also the exhaustive model checking and software implementation are considered too, many of those ALs supporting the structural and behavioural design decisions remain weak. As discussed in Section 6, just 5 ALs support the specifications of the system structures and behaviours, their exhaustive analysis, and generation into software code; and, unfortunately, those ALs are not so convenient for practical use in industry due to their notation sets. Indeed, those ALs offer process algebra based notation sets; so, practitioners are forced to learn and use process algebras for the behaviour specifications. Note however that as indicated by Malavolta et al.'s survey (Malavolta et al., 2012), algebraic ALs are not used by the practitioners due to being found as requiring a steep learning curve.

In this paper, the goal is to propose a new software architecture modeling approach that allows practitioners to specify, analyse, and develop their software architectures without using any algebraic code. To this end, the XCD AL (Ozkaya and Kloukinas, 2014) is aimed to be extended in this study, which uses the well-known Design-by-Contract (DbC) (Meyer, 1992) approach for specifying the structural, behavioural, and interaction design decisions. XCD's semantics have been defined in terms of its precise translation in SPIN's ProMeLa formal verification language (Holzmann, 2004). Using XCD's translator, practitioners can obtain the ProMeLa model of their contractual software architectures that satisfy the semantics rules and verify them using the SPIN model checker for a set of pre-defined properties (e.g., deadlock, race-conditions, wrong use of services, and incompleteness) and user-defined properties in linear temporal logic (LTL). However, XCD currently does not provide any support for the implementation of the software architecture specifications. Practitioners are expected to transform their design decision specifications into software implementation manually, which is error-prone and time-consuming. Therefore, to bridge this gap, this paper introduces XCD's precise translations in the Java programming language. This will let practitioners who use XCD for the contractual specifications of component behaviours and connectors' interaction protocols have the complete and consistent Java code that reflects all the decisions specified. Furthermore, to increase the code maintenance and understandability, the adapter design pattern has been applied for the Java code produced.

In the rest of this paper, initially, XCD's DbC-based notation set is introduced. This is followed by the explanations of the algorithms for translating XCD in Java. Next, the Java translation of XCD has been illustrated via the gas station case-study. Lastly, the related work and conclusion are given.

## 3 XCD's NOTATION SET

XCD offers components and connectors as first-class elements, which can be specified contractually.

### 3.1 Components in XCD

Components in XCD are the units of computations, which are specified with state data and ports. Figure 1 illustrates the component specifications via the client (lines 1-12) and server (lines 13-24) components. Both component specifications include the variable initialisations firstly (line 2 and 14), which represent the component state that are changed via the ports. The asynchronous (i.e., one-way event) communication is performed via the emitter and consumer types of ports (lines 8-11 and lines 20-23 respectively). The synchronous (i.e., request-response methods) communication is performed via the required and provided types of ports (lines 3-7 and 15-19 respectively).

XCD extends DbC and offers *functional* and *interaction* contracts for the port method/event behaviours, whose syntax and semantics differ by the type of the port. XCD introduces *accept* and *wait* interaction contracts. If the accept contract is not satisfied, the method call (or the event) is rejected immediately (line 16). If the wait contract is not satisfied, the method call (or the event) is delayed until it is satisfied (line 21). Whenever the interaction contract for a method/event is satisfied, the functional contract for that method/event is evaluated that essentially updates the component state under some conditions. The functional contract for emitter events or

[1]The analysis results for the ALs can be found in the following link: https://sites.google.com/site/ozkayamert1/als

```
1 component Client(int clientName){
2   bool clientData:= -1;
3   required port clientPort1{
4     @functional{promises: name := clientName;
5         requires: \result >= 0;
6         ensures: clientData:=\result;}
7     int getData(int name);  }
8   emitter port clientPort2{
9     @functional{promises: data := clientData;
10     requires: true; ensures: \nothing;}
11    setData(int data);  }
12 }
```

```
13 component Server(){
14   bool isDataSet:=false; int serverData=1;
15   provided port serverPort1{
16     @interaction{accepts:isDataSet;}
17     @functional{ requires: name >= 0;
18      ensures: \result := serverData*3;}
19    int getData(int name);  }
20  consumer port serverPort2{
21    @interaction{waits:!isDataSet;}
22    @functional{ensures: isDataSet:=true;serverData=data;}
23    setData(int data);  }
24 }
```

Figure 1: Contractual component specifications in XCD - client and server components.

```
1 connector client_server_conn(client{getData, setData},
2  role client{
3   bool isSet := false;
4   required port_variable pv1{
5    @interaction{ waits: isSet;ensures: \nothing;}
6    int getData(int name); }
7   emitter port_variable pv2{
8    @interaction{waits:!isSet;ensures:isSet:=true;}
9    setData(int data); }
10 }
```

```
server{getData, setData}){
11 role server{
12  provided port_variable pv1{
13   int getData(int arg);  }
14  consumer port_variable pv2{
15    setData(int arg2); }
16 }
17 connector link1(client{pv1},server{pv1});
18 connector link2(client{pv2},server{pv2});
19 }
```

Figure 2: Contractual specification of a connector for client and server.

required methods consists of *promises* (lines 4 and 9) and *requires-ensures* pair (lines 5-6, and line 10), where the former assigns parameters for the event/method to be emitted/called and the latter changes the component state after the emission/call if the *requires* pre-condition is satisfied. The functional contracts for consumer events or provided methods are evaluated after the method-call (or event) is received. Each functional contract consists of the *requires* and *ensures* pair (lines 17-18 and line 22) for updating the component state with the *ensures* data assignments whenever the *requires* pre-condition is satisfied.

Lastly, a component may be a composite, which consists of the instance specifications of other components and connectors. Indeed, system configurations are specified as a composite component in XCD, which includes the component instances composing the system and the connector instance(s) that are responsible for the interactions of those components.

### 3.2 Connectors in XCD

Connectors in XCD are the units of component interactions and used to specify the interaction protocols for a set of interacting components. As illustrated in Figure 2, connectors are specified with *roles* that are played by the participating components and a set of *links* that establish the connections between the roles (essentially the components). In Figure 2, the client role in lines 2-10 are played by the client component and the server role in lines 11-16 are played by the server component. Each role, just like components, consists of state data variables (line 3) and port-variables that represent the ports of the component playing the role. Role port-variables (lines 4-6 & 7-9 for the client role and lines 12-13 & 14-15 for the server role) may include any methods/events of the corresponding component ports, and each method-/event is attached with an extra interaction contract that constrains the interaction behaviour of the methods (lines 5 and 8). Note that role interaction contracts also have *ensures* data assignments for changing the role data if the *waits* condition is satisfied.

Connectors also have role parameters and these parameters are passed with the relevant components when the connectors are instantiated in a composite component for specifying the system configuration.

## 4 TRANSLATING XCD TO JAVA

In this section, XCD's translation in Java is discussed in a modular and thus more understandable manner. Initially the component translation is given, which is further modularised into translations of different types of ports. Then, the connector translation is given. Lastly, the translation of the system configurations is given, which consists of the instances of the components and the connectors.

Basically, each component is translated as a separate Java class and each connector is translated as a set of classes each corresponding to a different com-

munication link of the connector. These classes are instantiated and used together in another main Java class translated from the system configuration.

## 4.1 Translating Components

Listing 1 gives the component translation in Java. Each component that is instantiated in the system configuration is translated into a Java class. The body of the Java class firstly includes the declarations of the instance variables corresponding to the component state data (lines 3–4) and the data of the connector role(s) that the component plays (lines 5–7). Then, the class body includes the port method/event translations, which is discussed in the next section.

```
1 FORALL c ∈ configuration.componentSet
2  public class c.ID {
3   FORALL data ∈ c.dataSet
4       data.type data.ID = data.initialValue;
5   FORALL role ∈ c.roleSet
6     FORALL data ∈  role.dataSet
7        data.type data.ID = data.initialValue;
8   ComponentPorts_to_Java(c.type)
9 }
```

Listing 1: Translating component type.

## 4.2 Translating Component Port to Java

For each port, its methods/events are translated into a Java method in the body of the Java class translated for the component that owns the port. The port method/event translation is considered in two parts as shown in Listing 2. While the provided methods and consumed events are translated in the same way (line 8), the required methods and emitted events are the other two that are also translated in the same way (line 9).

```
1 ComponentPorts_to_Java(component c)
2  LET
3   providedMethods={port.methodSet | port∈c.providedPortSet}
4   requiredMethods={port.methodSet | port∈c.requiredPortSet}
5   consumerEvents={port.methodSet | port∈c.consumerPortSet}
6   emitterEvents={port.methodSet | port∈c.emitterPortSet}
7  IN
8   offeredServices(c, providedMethods ∪ consumerEvents)
9   requestedServices(c, requiredMethods ∪ emitterEvents)
```

Listing 2: Translating component port.

### 4.2.1 Translating Provided Methods and Consumer Events to Java

Listing 3 gives the translation of provided methods and consumer events. For each provided method (or consumer event), a Java method with *public* visibility is created (lines 3-22) in the Java class of the relevant component type. Note that each method is synchronised, which locks the class instance when the method is called. The body of the Java method starts with a *while* loop (lines 4–6) that essentially checks whether the port's *wait* interaction contract and the component's role interaction contracts for the relevant service are together satisfied. Java Object class's *wait* method is called (line 6) in the loop until the interaction contracts are satisfied. The *wait* method serves to stop the thread that executes the calling method and another thread that executes a different method of the same class (i.e., another port service of the same component) may interleave. The newly executing method may indeed change the component state that will later on satisfy the interaction contracts of the waiting thread. A provided method (or a consumed event) may alternatively have an *accepting* interaction constraint (lines 8–9). In such a case, the interaction contract is translated into an *if* statement whose dissatisfaction leads to an exception to be thrown, indicating the wrong use of component services. When the interaction contracts are all satisfied, firstly the role state data are updated using the role interaction contract's *ensures* data-assignments (lines 10–11). Then, the functional contracts for the provided method (or consumed event) can be processed (lines 13–19). Each functional contract for a method (or an event) is translated as an *if* statement, whose condition is the *requires* pre-condition and body includes the *ensures* assignments. So, whenever the functional constraint pre-condition is satisfied, the functional contract's *ensures* assignments are performed. Note that the *ensures* clause changes the component state data and assign the method result (if the result type is not void). If none of the functional contract's pre-condition is satisfied, an exception is thrown (lines 18-19), indicating the incompleteness of the functional constraint specifications. In line 20, the Java Object class's *notify* method is called, which notifies any waiting thread to resume its execution. Lastly in line 21, the result is returned for the method-call.

```
1 offeredServices(component c,providedMethods,consumedEvents)
2 FORALL s ∈ providedMethods ∪ consumedEvents
3    public synchronized s.returnType s.ID (s.paramList){
4       while !(s.IContract.waits  ⋀
5                 ⋀role∈c.roleSet role.s.IContract.waits)
6       {Object.wait();}
7       IF s.IContract.accepts ≠ NULL
8        if !(s.IContract.accepts)
9         throw new Exception("Wrong use of component services");
10      FORALL role ∈ c.roleSet
11       role.s.IContract.ensures;
12      IF s.FContractSet ≠ NULL
13       FORALL fc ∈ s.FunctionalContractSet
14         if(fc.requires) {
15           fc.ensures_dataAssignment;
16           result = fc.ensures_result;
17         }
18        else
19         throw new Exception("Incomplete func. behavior");
20      Object.notify();
21      return result;
22   }
```

Listing 3: Translating provided and consumer ports.

#### 4.2.2 Translating Required Methods and Emitter Events to Java

Required methods and emitter events are translated as depicted in Listing 4. For each required method (or emitter event), a Java method is created again within the Java class corresponding to the component (lines 5–28). Required ports (or emitter ports) of components make method requests (or event emission in the case of emitter ports) to the connected provided port (or consumer). Therefore, the translated Java methods are actually supposed to make method-calls to the translated Java methods of the connected provided methods (or consumed events). This issue has been tackled with using the adapter design pattern, where the adapter is the connector link class and the adaptee is the component class that operates the provided port (consumer port). As shown in lines 3-4, each Java class includes an instance variable for each required port (or emitter event). This instance variable is expected to store a reference to the adapter class instance (i.e., the connector link). Whenever the required method (or emitter event's method) is called via the adapter instance, the adapter directs the call to the class instance corresponding to the providing component.

The first statement in the translated Java methods for required methods (or emitter events) is the while loop for checking the port and role interaction contracts (lines 5–8). As in the translations of provided methods (and consumed events), the Java *Object* class's *wait* method is called in the loop (line 8) until the interaction contracts are satisfied, which causes another thread that executes another port service of the same component to interleave. When the interaction contracts are satisfied, the role data are updated initially using the role interaction constraints' *ensures* data-assignments (lines 9–10). Then, the Java method of the connected provided port are called via the connector variable (line 11). Note that the functional contract's *promises* assignments for the required port method (emitter event) are passed as parameters to the called method. After calling the method of the provided port (or consumer port), the functional contracts for the required method (or emitter event) are processed (lines 12-17). If none of the functional contract's pre-condition is satisfied, the program execution is again halted with an exception thrown to indicate the incomplete behaviour specification (lines 16-17). Lastly in line 18, the Java *Object* class's *notify* method is called to notify the waiting thread (i.e., the one whose *wait* method has been called) to resume their execution.

If a component has a required port (or emitter), besides the Java methods created for each required method (emitter event), another *compute* method is created (lines 20-27). The compute method serves for making service requests on behalf of the component's required (or emitter) ports. So, it is translated as the concurrent execution of multiple Java threads each calling a Java method corresponding to a unique required port method (or emitter event).

```
1 requestedServices(Component c,requiredMethods,emitterEvents)
2 FORALL port ∈ c.requiredPortSet ∪ c.emitterPortSet
3   port.ID link_port;
4 FORALL s ∈ requiredMethods ∪ emitterEvents
5   public synchronized s.returnType s.ID (s.paramList) {
6     while !(s.IContract.waits  ∧
7           ∧_{role∈c.roleSet} role.s.IContract.waits)
8     {wait();}
9     FORALL role ∈ c.roleSet
10          role.s.IContract.ensures;
11    result = link_port.service(s.FContractSet.promises);
12    IF service.FContractSet ≠ NULL
13     FORALL fc ∈ s.FunctionalContractSet
14       if(fc.requires)
15         fc.ensures;
16     else
17      throw new Exception("Incomplete fun. behavior");
18     notify();
19  }
20  public void compute(){
21   FORALL service ∈ requiredMethodSet ∪ emitterEventSet
22     (new Thread() {
23      public void run() {
24       try {service(service.argumentList);}
25         catch (Exception e){e.printStackTrace();}
26       }
27     }).start();
28  }
```

Listing 4: Translating required and emitter ports.

## 4.3 Translating Connectors

Besides components, connectors are translated in Java too, depicted in Listing 5. As aforementioned, connectors are specified with *(i)* component roles for imposing interaction protocols on components and *(ii)* links for connecting the component ports. For each link that a connector has, a Java class is created (lines 3–19). This Java class implements a Java interface (line 3) that has the methods corresponding to the methods/events communicated between the connected component ports. The connector class also has a constructor (lines 6–7) for receiving the class instance for the providing component (i.e., the one that provides the method/event services to the connected requiring component). In lines 9-18, the interface methods are implemented in a way that each method directs the call to the corresponding method of the providing component class instance. So, as discussed in Section 4.2.2, the required component class may use the connector link class instance to call the methods of the provided component class indirectly.

```
1 FORALL connector ∈ configuration.connectorSet
2  FORALL link ∈ connector.linkSet
3   public class link.ID implements interfaceID(requiredPort(link)){
4    type(providesComponent(link)) offeringComponent;
5
6    public link.ID(type(providesComponent(link)) c) {
7      offeringComponent = c;
```

```
8    }
9    IF type(link) = provided_required
10    FORALL m ∈ providedPort(link).methodSet
11     m.retType m.ID (m.paramSignatures) throws Exception {
12       return offeringComponent.m(m.paramList);
13     }
14   ELSE
15    FORALL e ∈ emitterPort(link).eventSet
16     void e.ID (e.paramSignatures) throws Exception{
17      offeringComponent.e(e.paramList);
18     }
19  }
```

Listing 5: Translating connectors.

*The underlined functions each returns a specific part of a VXCD specification (e.g., port element) and serves for simplifying the algorithm, making it more modular and easier to understand.*

## 4.4 Translating Configurations

The system configuration is translated as a Java class, depicted in Listing 6. The Java class herein includes *(i)* for each component involved in the configuration an instance variable referencing to the class instance of that component and *(ii)* for each connector link that connects the components an instance variable referencing to the class instance of that link. In lines 13-17, the link variables of the requiring component instances (i.e., the components with required/emitter ports) are each instantiated with a reference to the instance of the relevant connector's link.

The Java class for a configuration also has a *main* method (lines 19–26). This main method creates an instance of the configuration class and activates its components that need to request services from other components which offer services. To do so, the *compute* methods of those components are called.

```
1 public class configuration.ID {
2  FORALL component ∈ configuration.componentSet
3   component.type component.ID;
4  FORALL connector ∈ configuration.connectorSet
5   FORALL link ∈ connector.linkSet
6    link.ID link_instance;
7  public configuration.ID(){
8   FORALL component ∈ configuration.componentSet
9    component.ID = new component.type();
10  FORALL connector ∈ configuration.connectorSet
11   FORALL link ∈ connector.linkSet
12    link_instance = new link.ID();
13  FORALL component ∈ configuration.componentSet
14   FORALL reqPort ∈ component.type.requiredPortSet
15    component.link_reqPort = ConnectorLinkIns(reqPort);
16   FORALL emPort ∈ component.type.emitterPortSet
17    component.link_emPort = ConnectorLinkIns(emPort);
18 }
19 public static void main (String args[]){
20  configuration.ID system_config = new configuration.ID();
21
22  FORALL component ∈ configuration.componentSet
23   IF component.requiredPortSet != NULL OR
24      component.emitterPortSet != NULL
25    (component.ID).compute();
26  }
27 }
```

Listing 6: Translating system configurations.

# 5 EVALUATION

The code generation facility has been illustrated using the well-known gas station case-study (Naumovich et al., 1997). The gas station system consists of a number of customer components, a cashier component, and a pump component. Each customer basically makes a payment to the cashier, and, the cashier informs the pump for releasing the gas to the customer. Then, the customer gets the gas from the pump. The gas station case-study has been specified in XCD and translated into a formal ProMeLa model using XCD's ProMeLa translator. The resulting ProMeLa models have been formally analysed using the SPIN model checker. During the analysis here, the deadlock, race-condition, incompleteness, and wrong use of services properties have been checked, which are supported by XCD. Some user-defined properties have also been specified in LTL, which can be analysed via the SPIN model checker. More detailed info can be found in (Ozkaya and Kloukinas, 2014). After analysing the specifications and correcting the design errors, the gas station specification has been transformed into a Java source-code, which is partially given in Figure 3. Note that due to the space restriction, the translation of gas station's configuration is not shown here[2]. To automate the translation process, a code generator has been developed in the Metaedit environment[3], which can be accessible via the XCD's visual extension called VXCD[2].

Figure 4 depicts the structure of the classes generated from gas station. Initially, a Java class is generated for each component involved - customer (lines 1–24 of Figure 3), cashier (lines 25-50), and pump (lines 52-81). Note that these classes have method definitions for each required/provided method and emitter/consumer event of the corresponding components. For each connector link that connects any two component ports, a class is created. This link class implements a Java interface that consists of the Java method definitions corresponding to the services which are communicated between the linked component ports. The Java interfaces for gas station are given in lines 83-98 of Figure 3, and the connector link classes that implement the relevant interfaces are given in lines 100-120. Each connector link class is associated with the component classes. Whenever the component classes are instantiated as part of the configuration class, their link variables are initialised with

[2] The XCD specification for the gas station system, the translated full Java source-code, and the Java code-generator can be accessible via the link: https://sites.google.com/site/ozkayamert1/ vxcd

[3] http://metacase.com/

```
1 public class Customer {
2    //Data-variables
3     int chosenAmount=0;
4     boolean requestMade=false;
5    //Required-port link
6     gas_Link  gas_LinkInstance;
7 public synchronized void  pump() {
8  while(!requestMade){try{wait();}catch(InterruptedExp e){}}
9  int result = gas_LinkInstance.pump();
10  if(result == chosenAmount){requestMade = false;}
11  else if (result != chosenAmount){//..}
12  else throw new Exp("Incomplete func. behaviour");
13  notify();
14 }
15 //Emitter port link
16 pay_Link  pay_LinkInstance;
17 public  synchronized void pay(int amount)  {
18  while(requestMade){try{ wait();}catch(InterruptedExp e) {}}
19  pay_LinkInstance.pay(0);
20  requestMade = true;
21  notify();
22 }
23 //compute methods for the requested methods (see Sec. 4.2.2)
24 }
25 public class Cashier{
26  int payment_amount[]= new int[3];
27  int var_N, var_M;
28  public Cashier(int N ,int M)  {
29   var_N = N;
30   var_M = M;
31   for(int i=0;i<payment_amount.length; i++)
32    payment_amount[i]=0;
33  }
34  public synchronized void pay(int amount_arg, int portID)  {
35   if(!(payment_amount[portID] == 0))
36     throw new Exp("Wrong use of services");
37   if(true){payment_amount[portID]= amount_arg;}
38   else throw new Exp("Incomplete func. behaviour");
39   notify();
40  }
41  // emitter port link
42  toPump_Link  toPump_LinkInstance;
43  public synchronized void releasePump(int pumpID,int amount)
44   while((payment_amount [pumpID]== 0))
45   {try {wait();}catch(InterruptedExp e) {}}
46   toPump_LinkInstance.releasePump(pumpID,amount);
47   notify();
48  }
49  //compute methods for the requested methods (see Sec. 4.2.2)
50 }
51
52 public class Pump {
53  boolean PumpReleased[]= new boolean[3];
54  int payment_amount[]= new int[3];
55  int var_N;
56  public Pump(int N){
57   var_N = N;
58   for(int i=0;i<PumpReleased.length; i++)
59    PumpReleased[i]=false;
60   for(int i=0;i<payment_amount.length; i++)
61    payment_amount[i]=0;
62  }
63  public synchronized int pump(int portID)  {
64   if(!(PumpReleased[portID] == true))
65    {throw new Exp("Wrong use of services");}
66   if(payment_amount[portID] != 0){
67      PumpReleased[portID] = false;
68   }
69   else{throw new Exp("Incomplete func. behaviour");}
70   notify();
71   return payment_amount[portID];
72  }
73  public synchronized void releasePump(int pumpID,int amount) {
74   if(true) {
75    PumpReleased[pumpID]=true;
76    payment_amount[pumpID]=amount;
77   }
78   else throw new Exp("Incomplete func. behaviour");
79   notify();
80  }
81 }
82
83 public interface pay_Link{public void pay(int amount)throws Exp;}
84
85 public interface toPump_Link{
86  public void releasePump(int pumpID,int amount)throws Exp;}
87
88 public interface gas_Link{public int pump()throws Exp;}
89
90 public class  Customer_to_Pump implements  gas_Link {
91  Pump Pump_ins;
92  int portID ;
93  public  Customer_to_Pump(Pump Pump_arg, int portID){
94   Pump_ins=Pump_arg;
95   portID=portID;
96  }
97  public int pump(){ return Pump_ins.pump(portID);}
98 }
99
100 public class  Customer_to_Cashier implements pay_Link {
101  Cashier Cashier_ins;
102  int portID_var ;
103  public  Customer_to_Cashier(Cashier Cashier_arg, int portID){
104   Cashier_ins=Cashier_arg;
105   portID_var=portID;
106  }
107  public void pay(int amount){
108   try {Cashier_ins.pay(amount,portID_var);}
109   catch (Exp e) {e.printStackTrace();}
110  }
111 }
112
113 public class Cashier_to_Pump implements toPump_Link {
114  Pump Pump_ins;
115  public  Cashier_to_Pump(Pump Pump_arg ) {Pump_ins=Pump_arg;}
116  public void releasePump(int pumpID,int amount){
117    try {Pump_ins.releasePump(pumpID,amount);}
118    catch (Exp e) {e.printStackTrace();}
119  }
120 }
```

Figure 3: The translated Java code from the Gas Station specification in XCD.

Note: "Exp" represents Exception.

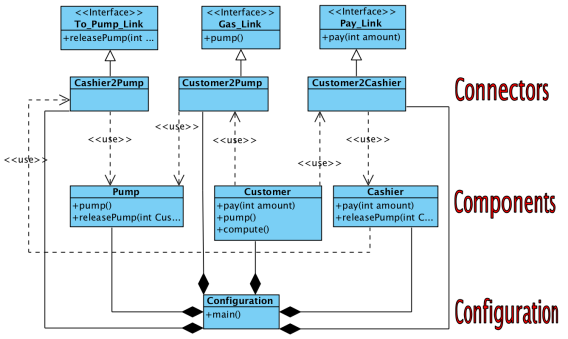


Figure 4: Gas station class diagram.

the relevant connector link class instances. The customer link variable in line 6 is initialised with the Customer2Pump class instance (lines 90-96). The customer link variable in line 16 is initialised with the Customer2Cashier class instance (lines 100-111). The cashier link variable in line 42 is initialised with the Cashier2Pump class instance (lines 113-120). So, whenever the component class instances make a method-call using the associated link variables, the link class instances receive the calls and direct them to the linked component class instances. For instance, the customer class instance makes a pump method-call using the Customer2Pump link class instance (see line 9). Then, the Customer2Pump class instance receives the call (see line 97) and directs it to the pump component (i.e., calling the pump class instance's pump method).

As illustrated in Figure 3, the Java source-code translations for XCD consider not only the structural design decisions but also the behavioural and interaction design decisions that may be specified. For in-

stance, the pump component's pump method is translated in lines 63-72, which essentially consists of three parts. In the first part (lines 64–65), the interaction contracts imposed on the pump method are translated. Apparently, the method is not constrained with any delaying interaction contracts (*waits*) - neither by the component port nor the connector roles. However, the method is constrained with an accepting interaction contract, translated as a selection construct in lines 64-65. So, if the pump has not been released yet, a Java exception will be thrown, which will halt the system execution. If the interaction constraints are satisfied (i.e., no exception thrown), the functional contract translated in lines 66-69 is executed. If the payment amount recorded for the customer is not zero (i.e., the first *requires* pre-condition), the corresponding *ensures* assigns the pump-released data to *false* to be able to release the gas again upon request. Otherwise, a Java exception is thrown and the program is halted (line 69) to indicate the incomplete behaviour specification (i.e., the case when the payment information is not received is not considered by the pump specification). Lastly, inline 70, the Object class's notify method is called and the result (i.e., the payment amount) is returned to the caller.

To enhance the understandability of the generated Java code, the translation algorithms have been designed using the adapter design pattern. For any connector link that connects the compatible ports of any two components, the translated Java class from the connector link acts as an adapter, the providing component class (i.e., the one with the provided/consumer port) acts as an adaptee, and the requiring component class (i.e., with the required/emitter port) acts as a client. The requiring component class uses the connector class to call the methods of the providing class indirectly. So, the classes of the two components whose required and provided (or emitter and consumer) ports are interconnected and request the methods (or events) of each other do not know each other thanks to the connector.

Practitioners may also test the executable Java programs produced from the XCD specifications. Indeed, as discussed in Section 4.2.1 and 4.2.2, the translation algorithms consider the incomplete functional behaviours and wrong use of component port services. Whenever such issues occur during the system execution, a Java exception is thrown with the necessary explanation message. When the Java source-code in Figure 3 produced from the gas station specification was run, a Java exception was thrown due to the wrong use of services and the program halted. According to the error message, the error has essentially been resulted from the pump method of the pump component (see lines 63-72 in Figure 3). During the system execution, the accepting interaction contract in lines 64-65 has been violated as the pump method has been called before the pump has been released. Note though that unlike formal analysis, software testing does not check every possible system paths for the aforementioned properties. Therefore, the non-existence of any errors during the software testing does not guarantee that every single path can also be executed successfully.

# 6 RELATED WORK

According to Malavolta etal.'s recent survey on the ALs, there are 125 different ALs that have been developed so far (Ivano Malavolta, 2017). The existing ALs have been analysed in this study for *(i)* the support for structural, behavioural, and interaction design decisions, *(ii)* the exhaustive model checking of those design decisions, and *(iii)* generating software code from the design decision specifications. According to the analysis results, only 6 ALs support all those requirements, i.e., Prisma (Pérez et al., 2003), Ambient-Prisma (Ali et al., 2010), AspectLeda (Navasa et al., 2007), Korrigan (Choppy et al., 2001), pi-ADL (Oquendo, 2004), and XCD (Ozkaya and Kloukinas, 2014). Table 1 shows the notations that the ALs use for specifying the structural, behavioural, and interaction decisions, the supported properties for analysis, and the programming language used in generating software code.

Prisma offers a UML-like graphical notation for specifying system structures in terms of components and connectors, and their behaviours/protocols are specified as an aspect that is specified in the pi-calculus process algebra (Milner et al., 1992). Prisma's modelling toolset allows for specifying user-defined requirements in temporal logics and checking the consistencies of the specifications and analysing them for the requirement specifications. Prisma's toolset also generates C# code, which can be executed on the .NET platform. Ambient-Prisma is the extension of Prisma with the notion of ambients for specifying mobile software systems. Unlike Prisma, Ambient-Prisma is based on Ambient-calculus (Cardelli and Gordon, 1998), in which the aspects are specified formally. Like Ambient-Prisma, pi-ADL also focuses on the software architecture specifications of mobile systems. Pi-ADL is based on the pi-calculus process algebra. Pi-ADL supports the specifications of user-defined requirements in pi-calculus (or temporal logic) and supports the exhaustive model checking via the CADP verification tool

Table 1: The ALs supporting the structural, behavioural, and interaction design decisions, and their analysis & code generation.

| ALs | Structure Specification | Behaviour Specification | Interaction Specification | Analysis Support | Code Gen. Support |
|---|---|---|---|---|---|
| Prisma | Graphical Notation | Pi-calculus | Pi-calculus | Consistency, User-defined reqs. | C# |
| Ambient-Prisma | Graphical Notation | Ambient-calculus | Ambient-calculus | Consistency, User-defined reqs. | C# |
| Aspect-Leda | Graphical Notation | Pi-calculus | Pi-calculus | Consistency, Model Checker | Java |
| Korrigan | Graphical Notation | Symbolic Transition System | Temporal Logic | Model Checker | Java |
| pi-ADL | Predicate Logic | Pi-calculus | Pi-calculus | Model Checker, User-defined reqs | .NET |
| XCD | Textual Notation | DbC | DbC | Model Checker, Completeness, User-defined reqs | Java |

(Garavel et al., 2001). Pi-ADL is supported with a compiler that translates the pi-ADL specifications into software code (e.g., C# and Visual Basic) in .NET platform. Aspect-Leda is another aspectual AL that extends the LEDA AL (Canal et al., 1999). So, like LEDA, Aspect-LEDA is based on pi-calculus for specifying component behaviours and connector interactions. Aspect-LEDA is supported by a couple of translators that translate the specifications in LEDA for using LEDA's model checking facilities. The Aspect-Leda specifications can be translated in Java for testing whether the software system behaves in the expected way or not. Korrigan uses again a UML-like graphical notation for specifying system structure and the behaviours of components are specified in symbolic transition system (i.e., a derivation of state transition systems). Korrigan uses temporal logic for specifying the component interactions (i.e., connectors). Korrigan specifications can be translated in the LOTOS formal verification language (Bolognesi and Brinksma, 1987) for model checking. Korrigan specifications can also be translated in Java. Lastly, XCD that has been extended with the Java development support in this paper is the only AL that offers non-algebraic, contractual (i.e., pre- and post-conditions) notation set for the specifications of structural, behavioural, and interaction design decisions. Note also that thanks to XCD's ProMeLa translator, the contractual specifications can be exhaustively analysed for a number of properties including completeness, deadlock, race-condition, wrong use of services, and user-defined requirements.

# 7 CONCLUSION

While the software architecture community offers tens of different ALs with a variety of capabilities, only a few of those ALs have been determined to support the specifications of the system structures, behaviours, and interactions that can be exhaustively analysed and then transformed into software implementation. Those ALs however have algebraic notation sets that are not preferred by practitioners unfortunately due to requiring a steep learning curve. To bridge this gap, this paper focuses on the XCD AL, which is based on the well-known DbC approach for the contractual (i.e., non-algebraic) specifications of the system structure, behaviour and interaction design decisions. XCD also supports the formal (i.e., exhaustive) analysis of the specifications using the SPIN model checker. So, XCD has been extended in the paper with a development viewpoint support and offered with the precise translation of the contractual XCD specifications in Java. XCD's Java translations considers all the structural, behavioural, and interaction design decisions supported by XCD and the translations have been designed using the adapter design pattern so as to enhance the quality of the code to be produced. The translation algorithms also consider checking for incomplete functional behaviours and wrong use of component services. So, practitioners can obtain the executable Java-based software systems that are complete and consistent with regard to their architecture specifications. To illustrate the Java extension for XCD, the gas station system has been specified, formally analysed, and then translated in Java. A prototype tool has also been developed for automating the Java source-code translation.

In the future, the XCD language is planned to be extended with the deployment, physical, and operational viewpoints to offer practitioners with the language that supports the full development life-cycle.

## ACKNOWLEDGEMENTS

This work has been supported by the project of the Scientific and Technological Research Council of Turkey (TUBITAK), Grant No: 215E159.

## REFERENCES

Aldrich, J., Chambers, C., and Notkin, D. (2002). Archjava: Connecting software architecture to implementation. In Tracz, W., Young, M., and Magee, J., editors, *ICSE*, pages 187–197. ACM.

Ali, N., Ramos, I., and Solís, C. (2010). Ambient-prisma: Ambients in mobile aspect-oriented software architecture. *Journal of Systems and Software*, 83(6):937–958.

Bolognesi, T. and Brinksma, E. (1987). Introduction to the iso specification language lotos. *Computer Networks and ISDN Systems*, 14(1):25 – 59.

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.

Cardelli, L. and Gordon, A. D. (1998). *Mobile ambients*, pages 140–155. Springer Berlin Heidelberg, Berlin, Heidelberg.

Choppy, C., Poizat, P., and Royer, J. (2001). The korrigan environment. *J. UCS*, 7(1):19–36.

Clements, P. C., Garlan, D., Little, R., Nord, R. L., and Stafford, J. A. (2003). Documenting software architectures: Views and beyond. In Clarke, L. A., Dillon, L., and Tichy, W. F., editors, *ICSE*, pages 740–741. IEEE Computer Society.

Garavel, H., Lang, F., and Mateescu, R. (2001). An overview of CADP 2001. Research Report RT-0254, INRIA.

Holzmann, G. J. (2004). *The SPIN Model Checker - primer and reference manual*. Addison-Wesley.

Ivano Malavolta, Patricia Lago, H. M. P. P. A. T. (2017). Architectural languages today. http://www.di.univaq.it/malavolta/al/.

Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., and Tang, A. (2012). What industry needs from architectural languages: A survey. *IEEE Transactions on Software Engineering*, 99.

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.

Milner, R., Parrow, J., and Walker, D. (1992). A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40.

Naumovich, G., Avrunin, G. S., Clarke, L. A., and Osterweil, L. J. (1997). Applying static analysis to software architectures. In Jazayeri, M. and Schauer, H., editors, *ESEC / SIGSOFT FSE*, volume 1301 of *Lecture Notes in Computer Science*, pages 77–93. Springer.

Navasa, A., Pérez, M. A., and Murillo, J. M. (2007). *AspectLEDA: Extending an ADL with Aspectual Concepts*, pages 330–334. Springer Berlin Heidelberg, Berlin, Heidelberg.

Oquendo, F. (2004). pi-adl: an architecture description language based on the higher-order typed pi-calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes*, 29(3):1–14.

Ozkaya, M. and Kloukinas, C. (2013). Are we there yet? analyzing architecture description languages for formal analysis, usability, and realizability. In Demirörs, O. and Türetken, O., editors, *EUROMICRO-SEAA*, pages 177–184. IEEE.

Ozkaya, M. and Kloukinas, C. (2014). Design-by-contract for reusable components and realizable architectures. In Seinturier, L., de Almeida, E. S., and Carlson, J., editors, *CBSE'14, Proceedings of the 17th International ACM SIGSOFT Symposium on Component-Based Software Engineering (part of CompArch 2014), Marcq-en-Baroeul, Lille, France, June 30 - July 4, 2014*, pages 129–138. ACM.

Pérez, J., Ramos, I., Martínez, J. J., Letelier, P., and Navarro, E. (2003). Prisma: Towards quality, aspect oriented and dynamic software architectures. In *QSIC*, pages 59–66. IEEE Computer Society.