Towards Design-by-Contract Based Software Architecture Design

Mert Ozkaya
Department of Computer Science
City University London
London, EC1V 0HB, UK
Email: mert.ozkaya.1@city.ac.uk

Christos Kloukinas
Department of Computer Science
City University London
London, EC1V 0HB, UK
Email: C.Kloukinas@city.ac.uk

***Abstract*—Design-by-Contract (DbC) gained wide familiarity among software developers for specifying software. It aids in documenting the behaviour of class methods as contracts between clients of the methods and their suppliers. This not only provides a user-friendly way of specifying software behaviour but also facilitates the verification of software correctness.**

In this paper, we provide a comprehensive extension to DbC so that it can be applied to the level of software architecture design. We illustrate this through our architecture description language XCD. Components in XCD have four different types of interfaces: provided and required interfaces of synchronous methods or emitter and consumer interfaces of asynchronous events where methods/events are contractually specified. Contract specification is separated into functional and interaction contracts thus modularising the functional and interaction component behaviours. Furthermore, treating interaction protocols as connectors, XCD allows to specify connectors with interaction contracts that participating components adhere to.

The formal semantics of XCD are defined using Finite State Process (FSP) thus enabling formal analysis of contractually specified software architectures for quality properties, e.g., deadlock.

I. Introduction

Since early nineties, several architecture description languages (ADLs) have been developed, e.g., Darwin [17], UniCon [29], Wright [2], LEDA [6], Koala [31], SOFA [26], and CONNECT [13]. They allow designers to specify architectures of large and complex systems. Some (Koala and UniCon) place their focus on automatic code generation, and some (Darwin, Wright, LEDA, SOFA, and CONNECT) on formal analysis of software architectures. Those addressing formal analysis mostly adopt process algebras (e.g., FSP [18] by Darwin and CONNECT, CSP [12] by Wright or π-calculus [22] by LEDA) in specifying the behaviour of software architectures. Indeed, process algebras provide formally defined, mathematical syntax and semantics and also lead to formal specifications which can be rigorously analysed through model checker tools. However, the syntax of process algebras looks unfamiliar to the practising designers who might find it hard to specify their systems as parallel composition of processes [1].

Design-by-Contract is another approach [20] that can be considered as alternative to process algebras in specifying the behaviour of software architectures. Based on Hoare's logic [11] and VDM's rely-guarantee [4] specification approach, DbC allows for formal specification of contracts for software components. A contract herein applies in general to class methods and is specified as a pair of pre- and post-condition where the former states what the caller of the method is obliged to do and the latter what benefits are guaranteed by the method supplier. Practitioners prefer DbC essentially in test-driven developments to specify test conditions which are used to verify the software quality. [14], [19]. Originally intended for Eiffel [21], DbC has so far been adopted by many programming languages, e.g., Java through JML [5], [7], [8]. Hence, given the level of familiarity DbC gained, we strongly believe that if DbC were adopted in specifying the behaviour of software architectures, practitioners would feel more comfortable in specifying their software architectures.

DbC has so far been primarily considered for OO programming languages; its application to the level software architecture design is relatively immature. Indeed, while software components at the programming level provide only interfaces to their environments, at the software architecture level they explicitly require interfaces from outside too. Furthermore, objects of classes perform synchronous method-based communication only, whereas components at the level of software architecture can perform asynchronous event-based communication too.

Being aware of this gap, we focus on extending DbC to the level of software architecture design; so, designers can specify software architectures in a both formal and user-friendly way. To this end, in this paper we present our XCD ADL adopting our extensions to the DbC and thus enabling DbC-based software architecture specifications. The rest of the paper firstly describes syntactically and semantically how component and connector in XCD are specified in the form of contracts. Next, the formal semantics of component and connector specifications are given in Finite State Process (FSP) enabling the formal analysis of XCD software architectures. Last part is the related work where similar works are discussed.

II. A Design-by-Contract Based Architecture Description Language

XCD, initially introduced in [15], extends DbC to better support features commonly found in component models such as CCM [23] and OSGi [24], [30]. XCD supports the contractual specification of both provided and required (synchronous) method interfaces which components need to function properly. Additionally, XCD adds support for the contractual specification of (asynchronous) event interfaces. Furthermore, as it treats connectors as first-class entities, XCD also supports the contractual specification of protocols that the components use to interact.

A. Component Specification

XCD allows designers to describe freely as components whatever they deem appropriate performing a functional behaviour in their system. Components are distinguished from component types. Each component type, as shown in Listing 1, is essentially specified in terms of (i) *ports* (or interfaces) representing the points of interaction with their environment, (ii) *data* representing the component state.

Listing 1: Generic component structure

```
component Name {
  data;*
  provided port Name {
    method;+
  };*
  required port Name {
    method;+
  };*
  emitter port Name {
    event;+
  };*
  consumer port Name {
    event;+
  };*
}
```

Figure 1 depicts the types of ports a component type can possess: $required$ and $provided$ for making method-call to outside and providing methods to outside respectively; $emitter$ and $consumer$ for emitting events to outside and receiving events from outside respectively. Required and provided ports are described with method specifications, while emitter and consumer ports with event specifications.

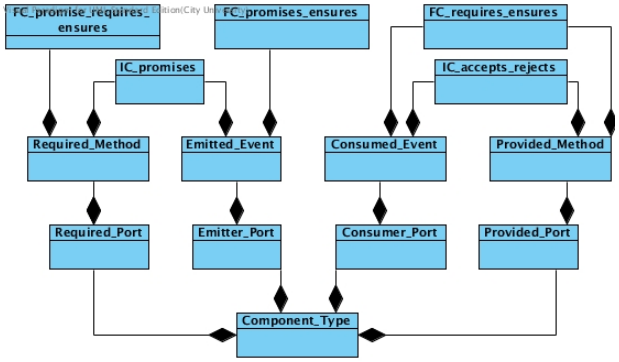

Fig. 1: Meta-model of component ports

XCD extends DbC firstly by separating in specification of components the interaction behaviour from functional behaviour. As illustrated in Figure 1, each component port consists of a set of actions (i.e., either method or event actions) which are described in terms of cleanly separated functional (FC_*) and interaction constraints (IC_*). The former allows for specifying the contract on the parameter arguments of method/event actions, and the latter for specifying the contract describing (i) the particular manner in which components want to behave (i.e., the order of actions) or (ii) the cases in which they do not know how to behave thus leading to *chaotic behaviour*. Note here that XCD allows for the contractual specification of required port behaviours. Furthermore, the behaviour of ports emitting/receiving events can also be contractually specified.

1) Required Port: Listing 2 exemplifies a required port specification through which a client components can make a $request$ call to a server. Constrained with interaction constraints in lines 2-4, the call for $request$ is delayed until the *promised* (pre) condition is met, the component data $opened$ evaluating to $true$. When a connection is opened, then the functional constraints in lines 5-12 can be evaluated. There in line 6, the parameter of the $request$ are *promised* to be equal to $self$ (i.e., the id of the component). In this case, upon receiving the response from the provided port of a server, if the *requirement* that an exception is not thrown is satisfied, the data $serverReply$ is *ensured* to be equal to the received result; *otherwise* (lines 10-11), the component state is not changed.

Listing 2: Required port specification

```
1 required port client_port{
2   @Interaction{
3     @promises: \when(opened);
4   }
5   @Functional{
6     @promises: caller == self;
7       @requires: !\exception;
8       @ensures: serverReply=\result ;
9      @otherwise
10      @requires: \exception;
11      @ensures: true;
12  }
13  int request(ID caller);
14 }
```

2) Provided Port: Listing 3 exemplifies a provided port specification. The port $server\_port$ receives calls for the method $request$ from clients. Upon receiving a call for the $request$, first the interaction constraints in lines 2-6 are evaluated. The call is $accepted$ when the $initialised$ data is $true$. However, the call is $rejected$ (line 5) if the $initialised$ evaluates to $false$, indicating chaotic behaviour. When the $accepts$ condition is met,then the functional constraints in lines 7-13 are evaluated. If the *requirement* that the $caller$ parameter of the received method-call is non-null is met, then the component data $numOfrequests$ is incremented and the result to be returned is assigned to 3. If however the caller is unassigned (lines 11-12), then a $NullID\_Exception$ is *ensured* to be thrown to the client.

Listing 3: Provided port specification

```
1 provided port server_port{
2   @Interaction{
3     @accepts: \when(initialised);
4     @otherwise
5     @rejects: \when(!initialised);
6   }
7   @Functional{
8     @requires:  !(caller == null);
9     @ensures: numOfrequests++ && \result = 3;
10    @also
11    @requires: caller == null;
12    @ensures: \throws(NullID_Exception);
13  }
14  int request(ID caller);
15 }
```

3) Emitter Port: Listing 4 exemplifies an emitter port specification. There, the port $client\_port2$ emits an event $initialise$ to a server. Note that unlike methods, event are specified without return types – only names and parameters allowed in its signature. Constrained with an interaction constraint,

the emission of the event $initialise$ is delayed until what is $promised$ is met, i.e., the component data $opened$ is $true$. When the client opens its connection, then the functional constraint in lines 5-8 is evaluated. It states that the actual parameter of the $initialise$ to be emitted is *promised* to be the id of the client which then ensures that the data $isInitialised$ is $true$. Note that unlike synchronous method ports, event ports are asynchronous; so, emitter ports do not wait for a response, nor do reicever ports send response.

Listing 4: Emitter port specification

```
1 emitter port client_port2{
2   @Interaction{
3     @promises: \when(opened);
4   }
5   @Functional{
6     @promises: client == self;
7     @ensures: isInitialised = true;
8   }
9   initialise(ID client);
10 }
```

4) Consumer Port: Listing 5 exemplifies a consumer port specification. The $server\_port2$ receives event $initialise$ from the emitter port of its clients specified in Listing 4. Constrained with interaction constraints, the event $initialise$ is *accepted* when the component data $initialised$ is $false$. Otherwise, when $initialised$ is $true$, the $rejects$ condition holds leading to chaotic behaviour. When the server is not yet initialised, the event $initialise$ is received successfully leading to the functional constraint in lines 7-11 being evaluated. There, the *client* parameter of the received $initialise$ event is *required* to be non-null which then ensures that the $client$ argument is stored in the data $initialiser$.

Listing 5: Emitter port specification

```
1 consumer port server_port2{
2   @Interaction{
3     @accepts: \when(!initialised);
4     @otherwise
5     @rejects: \when(initialised);
6   }
7   @Functional{
8     @requires: !(client == null);
9     @ensures: initialised = true &&
10              initialiser = client;
11  }
12  initialise(ID client);
13 }
```

B. Connector Specification

Besides component types specified in the form of $@interaction$ and $@functional$ contracts, connector types are also specified with contracts in XCD.

Connector types are, as shown in Listing 7, specified in terms of *roles* and *channels*. Each role acts as a component wrapper that represents the interaction behaviour of that component interacting via the connector. A role is described with *data*, and *port-variables*. The port-variables of a role essentially represent the respective ports of the components adopting the role. Channels of an XCD connector represent the communication links between interacting role port-variables and can have different types, e.g., synchronous, buffered, etc.

Representing the component ports, role port-variables can also be either of four different types depicted in Figure 1: required, provided, emitter, and consumer. However, port-variables impose only interaction constraints (through $@interaction$ contracts) on their actions. Indeed, they serve to mediate the interaction behaviour of component ports, avoiding chaotic behaviour. They essentially represent *high-level* interaction protocols that are imposed on the component(s) acting as the roles of the port-variable. The interaction protocols are intended for enforcing components to behave in a particular manner (i.e., through execution of certain action order). In doing so, components can be avoided from getting involved in unexpected 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.

Listing 6: Generic connector structure

```
connector Name {
  role Name {
    data;*
    provided port_variable Name {
      method;+
    };*
    required port_variable Name {
      method;+
    };*
    emitter port_variable Name {
      event;+
    };*
    consumer port_variable Name {
      event;+
    };*
  }
  channel;+
}
```

Listing 7 exemplifies a connector type specification for mediating the interaction between a server and a client. Client role in lines 3-11 are played by client components; server role in lines 12-20 by server components. The port-variable $client\_pv$ (lines 5-10) in the client role constrains the interaction behaviour of the $client\_port$ in Listing 2; the $@interaction$ contract herein delays the calls for method $request$ until the role data $isInitialised$ is $true$. The $@interaction$ specified in the $server\_pv$ of the server role constrains the $server\_port$ in Listing 3 so that call for method $request$ cannot be accepted until the role data $initialised$ becomes $true$. Therefore, client and server components are prevented from interacting before they ensure that server is initialised thus avoiding chaos. Note that due to space restriction, we have not included the port-variables associating with $client\_port2$ and $server\_port2$ which are to update $isInitialised$ and $initialised$ in client and server roles respectively.

The channel specification in lines 21-22 essentially describes the component port pair that are to communicate with each other. Indeed, the client port playing the $client\_pv$ communicates with the server port playing the $server\_port$ in a synchronous manner.

Components with their ports are passed as actual parameters to connectors. As shown in Listing 7, the $clients\_server$ has a parameter list for each of its role each augmented with its port-variables. At configuration time, when the $clients\_server$ is instantiated, the components playing the roles are passed as actual parameters along with their ports.

Listing 7: A connector specification

```
connector clients_server(client{client_pv},
                         server{server_pv} ) {
  role client{
    bool isInitialised = false;
    required port_variable client_pv{
      @Interaction{
        @promises: \when(isInitialised);
      }
      int request(ID caller);
    }
  }
  role server{
    bool initialised = false;
    provided port_variable server_pv{
      @Interaction{
        @accepts: \when(initialised);
      }
      int request(ID caller);
    }
  }
  channel sync clients2server(client.client_pv,
                              server.server_pv);
}
```

III. Component Semantics

Above, we implicitly explained the semantics of different port types without providing precise information. Now, we give the formal semantics of a component specification by showing how its data and different port types can be translated into formal Finite State Process (FSP) processes.

Definition 1 *The semantics of a component with data D and ports $p_1, .., p_n$ is the composite process:*

$$P_{D_c} \;||\; P_{p1} \;..\; ||\; P_{pn} \qquad (1)$$

where P_{D_c} is the data process and P_{p1},..., P_{pn} each is a composite port process whose definition is:

$$P_{IC} \;||\; P_{FC_{a1}} .. \;||\; P_{FC_{am}} \quad (2)$$

where P_{IC} is the interaction constraints process and $P_{FC_{a1}}$..,$P_{FC_{am}}$ each is a process for a functional constraints imposed on a single method/event action taken via the port.

Acting as the component memory, the data process P_D stores the component data as index variables of its sub process D. The process D executes $read$ and $write$ actions in a random order where the read has index variables holding the current data values, and the write has variables (V_n) holding the new data values to overwrite the current values of the D.

```
P_D = D([InitialValue(V)])*,
D([Name(V):Type(V)])* =
    read([Name(V)])* → D([Name(V)])*
  | write([Name(V)_n:Type(V)])* → D([Name(V)_n])*
).
```

As aforementioned, the interaction constraints for a port are mapped to P_{IC}. P_{IC} includes a sub process $Port$ which firstly $locks$ component data and performs $read$ action. Upon reading the data, for each event/method action of the port, a code snippet is produced in the body part.

```
P_IC(ID = 1)= Port,
Port=(lock→ read([Name(V):Type(V)])*
      → P([Name(V)])*),
P([Name(V):Type(V)])* = (
  ∀action∈port.actionList
    ..body part..
).
```

If the port is of emitter or required type, the body part is produced with the following pattern. There, for each functional constraint (fc) on the current action a $when$ statement is produced whose guard is the $logical\ OR$ of the action's interaction constraints. When the guard is $true$ (i.e., at least one of the interaction constraints are met), the event/method action is emitted/send, as in line 3, which has the promised values of the parameters as index variables. If the port is required, the response action is waited for, as in line 4, which has result/exception as its index variables. Then, the control is passed to the process P_{FC} through the internal action in line 5. Note that it is the P_{FC} that executes the functional constraints thus updating component data. P_{FC} then responds with another internal action as in line 7 where new data values are stored in the index variables (V_n). The component memory is updated with the new data values by executing $write$ action, and then the memory is released with $unlock$.

```
1 ∀fc∈action.FunctionalConstraints
2  when(⋁ InteractionConstraint)
3   action_e/m([promises(fc,arg)])*
4   (→ action_e/m([promises(fc,arg)])*[r:RES][e:EX])?
5   → internal_action([Name(arg)])*([Name(V)])*
6                                    ([r][e])?
7   → internal_action([Name(arg)])*([Name(V)])*
8                     ([Name(V_n):Type(V)])*
9   → write([Name(V_n)])*
10  → unlock
11  → Port
```

If the port is of consumer or provided type, the body part, following the below pattern, includes a single $when$ statement whose guard condition is the $logical OR$ of the action's normal interaction constraints. When the guard evaluates to $true$, then the event/method action is accepted as in line 2. Next, just like emitter/required ports, the control is passed to the process P_{FC} through the internal action in line 3. P_{FC} then responds with another internal action as in lines 4-6 where new data values are stored in the index variables (V_n) and in the case of provided ports so are the result/exception ($[r : RES][e : EX]$). The component memory is updated with the new data values by executing $write$ action, and then the memory is released with $unlock$. In the case of provided ports, a response action is executed as in line 9 which includes as index variables the action arguments and result/exception.

Besides the $when$ statement for normal interaction constraints, another alternative $when$ statement is for the exceptional cases as in lines 11-12. When any exceptional interaction constraints are met, this leads to $ERROR$ state.

```
1 when(⋁ Normal_InteractionConstraint)
2   action_e/m([Name(arg):Type(arg)])*
3   → internal_action([Name(arg)])*([Name(V)])*
4   → internal_action([Name(arg)])*([Name(V)])*
5                     ([Name(V_n):Type(V)])*
6                     ([r:RES][e:EX])?
7   → write([Name(V_n)])*
8   → unlock
9   (→ action_e/m([arg])* [r][e])?
10  → Port
11 | when(⋁ Exceptional_InteractionConstraint)
12    action_e/m([Name(arg):Type(arg)])*→ ERROR
```

As aforementioned, the process P_{FC} is produced for each event/method action of a port to compute the respective functional behaviour. The control is passed from P_{IC} to the P_{FC} through the internal action in lines 3-4. Note that the internal

action has index variables $[r : RES][e : EX]$ in the case of required ports which are the result/exception received from the response action ($action\_e/m$). Then, for each functional constraint (fc) on the action, a $when$ statement is produced whose guard is the $requires$ condition of the fc. When the guard is $true$, the internal action is responded to the P_{IC} again along with updated data values (V') as index variables Note that if the port is provided the result/exception calculated are also passed as index variables ($[r']['e]$ in line 9).

```
1 ∀action∈port.actionList
2   PFC(ID = 1) = (
3     internal_action([Name(arg):Type(arg)])*
4       ([Name(V):Type(V)])*
5       ([r:RES][e:EX])?
6     →(∀fc∈FunctionalConstraints
7         when(requires(fc))
8         internal_action([Name(arg)])* ([V']*
9                                       ([r']['e'])?
10        → PFC)
11  ).
```

IV. Connector Semantics

Just like components, we have not yet provided a precise definition of connector specification. Below, we give a formal definition of connector in FSP.

Definition 2 *The semantics of a connector with roles* r_1,..., r_n *channels* ch_1,..., ch_n *is the composite process:*

$$P_{r1} \;..\; ||\; P_{rn} \qquad (3)$$

where P_{r1}..., P_{rn} each is a role process whose definition is:

$$P_{D_r} \;||\; P_{pv_1} \;..\; ||\; P_{pv_n} \qquad (4)$$

where P_{D_r} is the data process and P_{pv_1},..., P_{pv_n} each is a port-variable process that represents the interaction constraints imposed on method/event actions taken by the port-variable.

While role data is mapped to a process in the same way as the component data, port-variables in a role are mapped in a different way from component ports. This is due to port-variable imposing solely interaction constraints on actions.

Below is the pattern followed in mapping a port-variable of any of the four types into an FSP process (P_{pv}). Firstly, role memory is $locked$ and data are $read$ as in line 2. Then, in lines 5-10, for each action of the port-variable, a set of $when$ statements is produced each corresponding to a unique interaction constraint (ic) imposed on that action. The $when$ guard herein is either a $promises$ condition in the case of emitter and required port-variables or an $accepts$ condition in the case of consumer and provided port-variables. When an interaction constraint of an action is met leading to the respective $when$ guard being $true$, then the event/method action is executed as in line 8. This is followed by the $write$ action which updates the role memory with the new data values (V_n) imposed by the $ensures$ of the ic.

```
1 Ppv(ID = 1)= Port_var,
2 Port_var=(lock→ read([Name(V):Type(V)])*
3           → Pv([Name(V)])*),
4 Pv([Name(V):Type(V)])* =
5 (∀actionεportvar.actionList
6   ∀icεaction.InteractionConstraints
7    when(promises(ic)) //OR accepts(ic)
8     pv_action_e/m([Name(arg):Type(arg)])*
9     →write([Name(V_n)])* → unlock
10    → Port_var
11 ).
```

Channels of a connector are mapped to *relabelling functions* employed in the composite process corresponding to the connector. The relabelling function, for each channel, re-names the actions taken by the required/emitter port-variable in one end of the channel to the names of the respective actions taken by the provided/consumer port-variable in the other end. This enables the port-variable processes to synchronise on these actions.

As aforementioned, connector receives as actual parameters the components and their ports that play its roles. Each matching here between a component port and a role port-variable is also mapped to a relabelling function. That is, the role port-variable actions are renamed to the respective port actions. This enables the port and port-variable processes to synchronise on these actions.

V. Related Work

To the best of our knowledge, XCD ADL is the first approach to specifying software architectures which applies DbC comprehensively to both component and connector specifications. Indeed, contractual specification of components is two folds: functional and interaction contracts cleanly separating functional from interaction behaviour of components. The contracts can be attached to methods provided or required by ports and events too emitted or received by ports. Moreover, components interact with each other under the control of interaction protocols specified by the connectors. So, XCD allows to specify interaction protocols as interaction contracts which the participating components adhere to in their behaviour.

Research community has so far placed their main focus on applying DbC to component specifications only. Beugnard et al.'s approach [3] is considered the inspiring work in applying DbC to components. They proposed four types of component contracts: basic, behavioural, synchronisation, and quality-of-service contracts. However, components, just like Java classes, are considered with provided interfaces ignoring explicit specification of required interfaces and also interfaces of asynchronous events. There are also DbC based approaches that consider components with explicit required interfaces e.g., [9] and [16]; however, just like Beugnard et al.'s work, they do not support interfaces for events either.

Interest shown towards DbC-based architecture specification remains rather weak. This seems to be due to current approaches following CBSE where connectors are not granted top-level status. Thus, contracts for connectors, through which components interact, seem to be found immaterial. The work of Schreiner et al. [28] along with Schmidt et al.'s TrustME ADL [27] are some of the very few examples applying DbC at the level of software architecture and introducing contracts for connectors too. Schreiner et al.'s work however does not support component interfaces emitting or receiving events. Moreover, unlike XCD, 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 thus neglecting interfaces of asynchronous events. Furthermore, unlike current approaches, Fiadeiro et al. [10] proposed a connector-centric DbC based approach that focuses on defining contracts for connectors only which act as coordinators for the participating components.

VI. Conclusion

In this paper, we presented a series of extensions to DbC for adapting it to software architecture design. We illustrated this through our architecture description language, XCD which enables to specify software architectures in the form of contracts. XCD is unique in the sense that components can be contractually specified either with required and provided interfaces of synchronous methods or with emitter and consumer interfaces of asynchronous events. Contracts are applied at the level of methods/events to specify their behaviour. To enhance modularity, they are split into two types: functional and interaction contracts where the former represents the functional behaviour and the latter the interaction behaviour of the component interfaces.

Treating interaction protocols explicitly as connectors in architectural designs, XCD also enables the contractual specification of connectors. They are specified with interaction contracts for participating components. Thus, components interacting through the connectors are ensured to adhere to interaction protocols in their behaviours.

Furthermore, we provide formal semantics of component and connector specifications in Finite State Process (FSP) which enables the formal analysis of contractual specifications for safety and liveness properties (e.g., deadlock).

VII. Acknowledgements

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

References

[1] Alessandro Aldini, Marco Bernardo, and Flavio Corradini. *A Process Algebraic Approach to Software Architecture Design*. Springer, 2010.

[2] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997.

[3] Antoine Beugnard, Jean-Marc Jézéquel, and Noël Plouzeau. Making components contract aware. *IEEE Computer*, 32(7):38–45, 1999.

[4] Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer, 1978.

[5] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, 2005.

[6] Carlos Canal, Ernesto Pimentel, and José M. Troya. Specification and refinement of dynamic software architectures. In Patrick Donohoe, editor, *WICSA*, volume 140 of *IFIP Conference Proceedings*, pages 107–126. Kluwer, 1999.

[7] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 342–363. Springer, 2005.

[8] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In Boris Magnusson, editor, *ECOOP*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255. Springer, 2002.

[9] Daniel Enselme, Gerard Florin, and Fabrice Legond-Aubry. Design by contract: Analysis of hidden dependencies in component based application. *Journal of Object Technology*, 3(4):23–45, 2004.

[10] José Luiz Fiadeiro and Luis Filipe Andrade. Interconnecting objects via contracts. In *TOOLS (38)*, pages 182–183. IEEE Computer Society, 2001.

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

[12] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[13] Valérie Issarny, Amel Bennaceur, and Yérom-David Bromberg. Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability. In Marco Bernardo and Valérie Issarny, editors, *SFM*, volume 6659 of *Lecture Notes in Computer Science*, pages 217–255. Springer, 2011.

[14] David Janzen and Hossein Saiedian. Test-driven development: Concepts, taxonomy, and future direction. *IEEE Computer*, 38(9):43–50, 2005.

[15] Christos Kloukinas and Mert Ozkaya. Xcd - Modular, realizable software architectures. In Pasareanu and Salaün [25], pages 152–169.

[16] Zhiming Liu, Jifeng He, and Xiaoshan Li. Contract oriented development of component software. In Jean-Jacques Lévy, Ernst W. Mayr, and John C. Mitchell, editors, *IFIP TCS*, pages 349–366. Kluwer, 2004.

[17] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *SIGSOFT FSE*, pages 3–14, 1996.

[18] Jeff Magee and Jeff Kramer. *Concurrency - state models and Java programs (2. ed.)*. Wiley, 2006.

[19] E.M. Maximilien and L. Williams. Assessing test-driven development at IBM. In *25th Intl. Conf. on Software Engineering*, pages 564–569, May 2003.

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

[21] Bertrand Meyer, Jean-Marc Nerson, and Masanobu Matsuo. Eiffel: Object-oriented design for software engineering. In Howard K. Nichols and Dan Simpson, editors, *ESEC*, volume 289 of *Lecture Notes in Computer Science*, pages 221–229. Springer, 1987.

[22] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, 1992.

[23] OMG. Common object request broker architecture (CORBA) specification, version 3.3 – Part 3: CORBA component model. Specification formal/2012-11-16, OMG, November 2012. //omg.org/spec/CORBA/3.3/.

[24] OSGi Alliance. OSGi core release 5. Specification, March 2012. // osgi.org/.

[25] Corina S. Pasareanu and Gwen Salaün, editors. *Formal Aspects of Component Software, 9th International Symposium, FACS 2012, Mountain View, CA, USA, September 12-14, 2012. Revised Selected Papers*, volume 7684 of *Lecture Notes in Computer Science*. Springer, 2013.

[26] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Trans. Software Eng.*, 28(11):1056–1076, 2002.

[27] Heinz Schmidt, Iman Poernomo, and Ralf Reussner. Trust-by-contract: Modelling, analysing and predicting behaviour of software architectures. *J. Integr. Des. Process Sci.*, 5(3):25–51, August 2001.

[28] Dietmar Schreiner and Karl M. Göschka. Explicit connectors in component based software engineering for distributed embedded systems. In Jan van Leeuwen, Giuseppe F. Italiano, Wiebe van der Hoek, Christoph Meinel, Harald Sack, and Frantisek Plasil, editors, *SOFSEM (1)*, volume 4362 of *Lecture Notes in Computer Science*, pages 923–934. Springer, 2007.

[29] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Trans. Software Eng.*, 21(4):314–335, 1995.

[30] Andre Luiz Camargos Tavares and Marco Tulio de Oliveira Valente. A gentle introduction to OSGi. *ACM SIGSOFT Software Engineering Notes*, 33(5), 2008.

[31] Rob C. van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, 2000.