FORMAL VERIFICATION OF CONTRACTUAL SOFTWARE ARCHITECTURES USING SPIN

Mert Ozkaya

Department of Computer Engineering, Istanbul Kemerburgaz University,

Email: mert.ozkaya@kemerburgaz.edu.tr

Abstract

Software architectures have become one of the most crucial aspects of software engineering. Software architectures let designers specify systems in terms of components and their relations (i.e., connectors). These components along with their relations can then be verified to check whether their behaviours meet designers' expectations. XCD is a novel architecture description language, which promotes contractual specification of software architectures and their automated formal verifications in SPIN. XCD allows designers to formally verify their system specifications for a number of properties, i.e., (i) incomplete functional behaviour of components, (ii) wrong use of services operated by system components, (iii) deadlock, (iv) race-conditions, and (v) buffer overflows in the case of asynchronous (i.e., event-based) communications. In addition to these properties, designers can specify their own properties in linear temporal logic and check their correctness. In this paper, I discuss XCD and its support for formal verification of software architectures through a simple shared-data access case study.

Keywords: software architectures, formal verification, SPIN, design-by-contract

1.0 INTRODUCTION

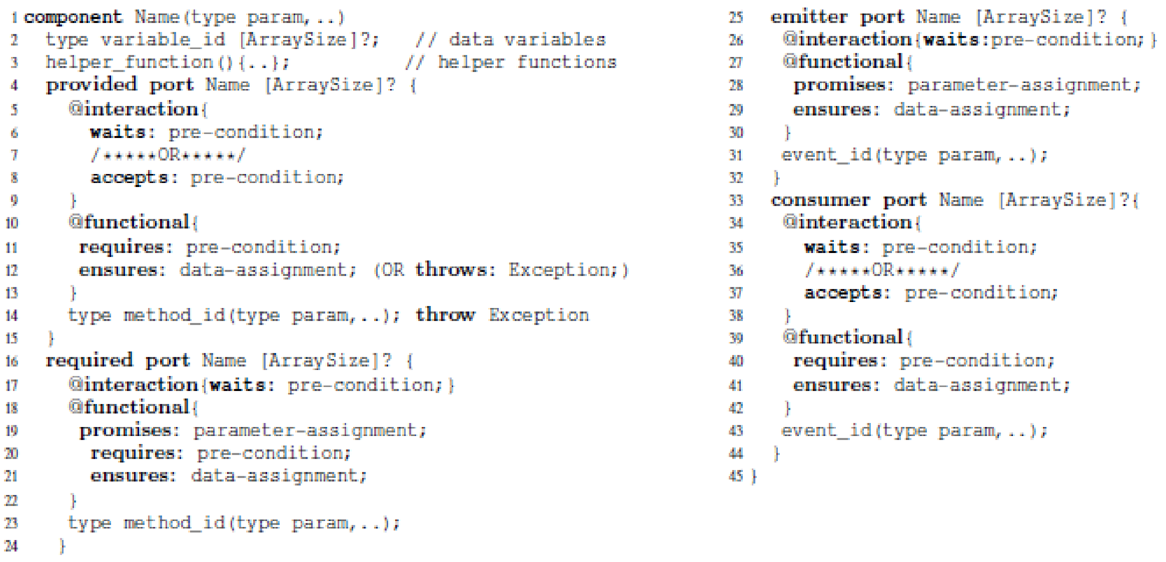
Software architecture [8, 11, 31] is a high-level design activity, concerned with the successful composition of components into an entire system that meets functional and non-functional requirements. It is at the level of architectural design where low-level details of components are suppressed, and, their high-level complex interactions via the component interfaces (i.e., the protocols of interactions) can be focused on and reasoned about. So, design problems, e.g., the use of interface services in the wrong order, can be identified early on at the stage of high- level design. Indeed, problems due to incompatible interfaces of inter-connected components are crucial, which prevent the components from being composed to a whole system and analysed for non-functional properties, e.g., reliability and security.

Unified Modelling Language (UML) [35] is the de facto language for visually specifying and designing software systems. UML supports both high-level and low-level designs, which is widely used in specifying high-level software architectures too. It offers a variety of diagrams, such as class and component diagrams. Using these diagrams, systems can be specified as a composition of components that are connected with each other via association links [18]. However, UML does not support first-class specification of interaction protocols for the linked compo- nents, which are crucial for reasoning about their composition. Moreover, UML has very weak formal semantics, which are open to different interpretations and not easily formally analysed.

Another alternative method for specifying software architectures is the architecture description languages (ADLs), which have emerged in the nineties and become one of most active areas of software engineering [7, 25]. There are numerous ADLs developed so far, e.g., Darwin [22], Wright [1], LEDA [5], CONNECT [17], PiLar [33], etc. Each ADL offers its own architectural notation, but, they share basic notions, e.g., components, interfaces, and connectors. Unlike UML, ADLs allow designers to specify the architectures of their systems precisely. Moreover, ADLs are offered with various features depending on their scope of interest. Some offer automatic code generation for facilitating the implementation of the specified systems. Some offer notations for specifying non-functional properties of systems (e.g., reliability and security), which can be communicated among stakeholders and analysed via analysis tools. Some offer notations based on formal methods (e.g., process algebras [3]) for specifying the behaviours of architectural elements and formally verifying them using formal analysis tools, e.g., model checkers.

As introduced above, there are many techniques for specifying software architectures, such as UML and ADLs. Each technique has its own advantages and disadvantages. UML for instance are found easy to learn and use by practitioners thanks to its visual notation set. However, UML does not have formally defined semantics, which may lead to imprecise specifications that are interpreted differently. ADLs differ from UML by their precise notations, which also let designers perform further operations on their software architectures such as automatic code generation, simulation, and formal analysis. However, ADLs are not as widely-used as UML by practitioners, since ADLs are based on formal methods to enable formal analysis that make their learning curve steep too.

I developed an architecture description language called XCD that addresses the problems of UML and ADLs. Unlike UML, XCD promotes the modular design by viewing software architectures as a composition of components and connectors. While components in a software architecture represent the main computation units of the software system, connectors represent the interaction protocols for these components. To reduce the learning curve, unlike ADLs XCD does not adopt algebraic notations for specifying components and connectors. XCD extends the well- known Design-by-Contract (DbC) approach [26] and allows for the contractual specification of components and connectors. DbC was first introduced with the Eiffel programming language. Later on, it has been applied to many programming languages, e.g., Java Modelling Language (JML) [6] for Java and Spec# [2] for C#. Moreover, DbC has been found by some academics as easy to teach and use. They use DbC to teach their undergraduate students how to use formal methods to specify software behaviours and check their correctness [19]. XCD is also supported by a compiler1 that translates XCD architecture specifications into ProMeLa formal verification language that is accepted by the SPIN model checker. ProMeLa models can then be verified via SPIN for (i) incomplete functional behaviour of components, (ii) wrong use of services operated by system components, (iii) deadlock, (iv) race- conditions, and (v) buffer overflows in the case of asynchronous (i.e., event-based) communications. Designers can also specify their own properties in linear temporal logic.



```
component Name(type param,..)
  type variable_id [ArraySize]?;    // data variables
  helper_function(){..};            // helper functions
  provided port Name [ArraySize]? {
    @interaction{
      waits: pre-condition;
      /*****OR*****/
      accepts: pre-condition;
    }
    @functional{
     requires: pre-condition;
     ensures: data-assignment; (OR throws: Exception;)
    }
    type method_id(type param,..); throw Exception
  }
  required port Name [ArraySize]? {
    @interaction{waits: pre-condition;}
    @functional{
     promises: parameter-assignment;
      requires: pre-condition;
      ensures: data-assignment;
    }
    type method_id(type param,..);
   }
  emitter port Name [ArraySize]? {
   @interaction{waits:pre-condition;}
   @functional{
    promises: parameter-assignment;
    ensures: data-assignment;
   }
   event_id(type param,..);
  }
  consumer port Name [ArraySize]?{
   @interaction{
     waits: pre-condition;
     /*****OR*****/
     accepts: pre-condition;
   }
   @functional{
    requires: pre-condition;
    ensures: data-assignment;
   }
   event_id(type param,..);
  }
}
```

Figure 1. Generic Component structure

Array notation is optional (indicated via ?) and used for specifying array elements.

1.1. Structure of XCD

Xcd consists structurally of component and connector elements. Components are used to specify the behaviours of computational units composing a system. Connectors are used to specify the interaction protocols for these components so that they can be composed to a whole system successfully.

[1]XCD's compiler is available via the link: https://sites.google.com/site/ozkayamert1/home/xcd

```
connector usertype (roleName[ArraySize]?{pv_prov, pv_req, pv_cons, pv_em},...) {

  role roleName  {

    type variable_id [ArraySize]?;    // data variables
    helper_function() {...}           // helper functions

    provided port_variable pv_prov [ArraySize]? {
      @interaction{
       waits: pre-condition;
       ensures: data-assignments;}
      type method_id(type param,..);
    }
    required port_variable pv_req [ArraySize]? {
      @interaction{
       waits: pre-condition;
       ensures: data-assignments;}
      type method_id(type param,..);
    }
    consumer port_variable pv_cons [ArraySize]? {
      @interaction{
       waits: pre-condition;
       ensures: data-assignments;}
      event_id(type param,..);
    }
    emitter port_variable pv_em [ArraySize]? {
      @interaction{
       waits: pre-condition;
       ensures: data-assignments; }
      event_id(type param,..);
    }
  }
  //link connectors
  connector link1(roleName{pv_prov}, ...);
}
```

Figure 2. Generic connector structure

Array notation is optional (indicated via ?) and used for specifying array elements.

1.1.1. Component

Figure 1 depicts the structure of XCD components, which consists of data and ports. Just like instance variables in Java classes, component data represent the state of the component, which are manipulated through the component ports.

Port

Component ports are the interfaces of the component through which the component can interact with its environment. The communication of any two components can be either synchronous (i.e., two-way) or asynchronous (i.e., one-way). In synchronous communication, components interact through their required and provided ports, where the required port of the component makes method-call to the provided port of the other component and waits for the method-response. In asynchronous communication, components interact through their emitter and consumer ports, where the emitter port of the component emits events to the consumer port of the other component.

While required and provided ports are specified with the methods that they request and offer respectively, emitter and consumer ports with events that they emit and consume respectively. Each port event/method has a behaviour, which is specified with interaction and functional contracts in XCD. An interaction contract for a method/event describes when the port method/event can be operated. A functional contract describes how the component state changes when the method/event is operated. Interaction contracts have delaying pre-condition (waits); however, for provided and consumer ports, the contract may alternatively have an accepting pre-condition (accepts) that accepts a call or immediately rejects it without any delays. Functional contracts differ by the port type. Indeed, functional contracts for provided and consumer ports are specified with a pre-condition (requires) and data-assignments (ensures) where the satisfaction of the pre-condition ensures that the component state is updated using the data- assignments. Note that provided port methods may alternatively throw an exception (throws) for abnormal cases. For required and emitter ports, functional contracts are further enriched with parameter-assignments (promises) too.

Composite Component
Components in XCD can also be specified in terms of the configuration of other components. By doing so, a component can be used to describe complex computational units in a modular and thus more understandable way.

1.1.2. Connector

The structure of XCD connectors is given in Figure 2, which consists of roles and link connectors.

Role
Each connector role represents a participating component and describes the interaction protocol that the component needs to satisfy in its interaction through the connector. Just like components, roles are specified with data and port-variables. Role data holds the state of the role; and, role port-variables represent the ports of the participating component.

Just like component ports, port-variables can be of four different types: required, provided, emitter, and consumer. Port-variables consists of methods/events. The behaviours of these methods/events are specified with interaction contracts, which aim at constraining the actual port methods/events to meet an interaction protocol. Role port-variables cannot have functional contracts since roles cannot access to component's action parameters and results. Note however that unlike port interaction contracts, role port-variables' interaction contracts can have ensures data-assignments to update the role state data.

Link connector
Each connector has a number of link connectors that establish the communication between the participating components.

2.0 SEMANTICS OF XCD

XCD's semantics [29, 30] have been defined using SPIN's ProMeLa formal verification language [14]. That is, any XCD architecture can be precisely translated into a ProMeLa model that can then be formally verified via SPIN's model checker. XCD also encodes a number of properties in its ProMeLa semantics, enabling their automatic checking during the formal verification. These properties are for checking (i) incomplete functional behaviours of system components, (ii) wrong use of component services, (iii) race-conditions, and (iv) buffer overflow in asynchronous communication. XCD's ProMeLa translations enable designers to specify their own properties and check their correctness too. However, giving just the precise ProMeLa translation is not enough for designers to translate their XCD specifications into ProMeLa models. Indeed, manual transformations are not only impractical but also error-prone. As shown in the rest of this section, the translation algorithms are quite complex, which makes it harder to get the resulting ProMeLa models working (i.e., accepted by the SPIN model checker for verification) in the first instance. Besides, for large systems it takes time and huge effort as the ProMeLa model grows proportionally with the number of components. Therefore, I developed a compiler that automates this transformation process and renders the formal analysis of XCD specifications sufficiently practical for designers.

2.1. Why SPIN

SPIN's input language ProMeLa has the following distinguishing features, which helped in defining the precise translation of XCD and also facilitate the formal analysis of software architectures.

Firstly, XCD's high-level semantics [28] was defined using Dijkstra's guarded command language [9]. ProMeLa has also its roots in Dijkstra's language, which made it easier to define XCD's semantics using ProMeLa.

Another decisive factor is the advanced model checker offered by SPIN. Unlike many model checkers, the SPIN model checker does not attempt to construct the state-space of each process as it is defined but only does so on- the-fly, as needed. It is also free and open-source, which can easily be installed and even modified by designers according to their own interest. Another good thing about SPIN is that it deals with the state space explosion problem of model checking effectively and provides (i) various search techniques, e.g., depth-first search and breadth-first search, and (ii) state storage techniques, e.g., bit-state hashing [15]. Furthermore, SPIN offers various simulation techniques too, e.g., interactive, random, and guided.

Unlike other formalisms (e.g., FSP [23], CSP [13], and π-calculus [27]), ProMeLa offers a more user-friendly notation, which resembles in some cases the C programming language. This can therefore help designers in understanding the ProMeLa translations of their XCD specifications and dealing with the results of the formal verifications.

```
1  Component2Promela(PrimitiveCInstance primComponent)
2  LET
3  RoleVars = {role.VariableSet | role ∈ roleSet(primComponent)};
4  VarSet= RoleVars ⋃ ctype(primComponent.typename).VariableSet;
5  IN
6   proctype primComponent.instancename () {
7     FORALL var ∈ VarSet
8       var.DataType pre_state(var) = initialValue(var);
9       var.DataType post_state(var) = initialValue(var);
10      FORALL port ∈ ∈ ctype(primComponent.typename).PortSet
11        IF port.RequiredPort != null ∨ port.ProvidedPort != null
12         var.DataType pre_state_copy(port, var) = initialValue(var);
13    FORALL port∈primComponent.ConsumerPortSet∪primComponent.ProvidedPortSet
14       bufferType(port) bufferID(port)[numOfConnections(port)];
15    FORALL port ∈ primComponent.RequiredPortSet
16       chan responseChannelID_cond(port)[2];
17   Start:
18   do
19    FORALL port ∈ ctype(primComponent.typename).PortSet
20      IF port.EmitterPort != null
21       Port2Promela_Emitter(primComponent, port.EmitterPort);
22      IF port.ConsumerPort != null
23       Port2Promela_Consumer(primComponent,port.ConsumerPort);
24      IF port.RequiredPort != null
25       Port2Promela_Required(primComponent, portRequiredPort);
26      IF port.ProvidedPort != null
27       Port2Promela_Provided(primComponent,port.ProvidedPort);
28   od
29  }
```

Listing 1: Translating a component specification

2.2. Mapping XCD to SPIN's ProMeLa

In this section, I introduce the precise translation of XCD specifications into ProMeLa models. To aid in understanding the entire translation algorithm, the whole algorithm is divided into a number of sub-algorithms, each handled by a distinct routine. Each routine is essentially responsible for the translation of a certain part of an XCD specification. A routine receives as parameters the relevant part of an XCD specification, which are navigated through dot notation. Note that to abstract details in algorithms, some translation routines are defined using some functions that are underlined [2] (e.g., numOfConnections(port)). These functions each accept a parameter, which is the specification of an architectural element, and return either (i) a specific ProMeLa code for the parameter or (ii) a result of a calculation required in the translation process.

2.2.1. Translating Components

Each component is translated into a separate ProMeLa process, which is then instantiated in composite component processes. The PrimitiveComponent2Promela routine in Listing 1 shows how a component type is translated into a process.
Initially, the routine records the component data and the data of the roles that the component assumes (lines 3-4). Lines 6–30 gives the process declaration. It contains a pair of variables for each component data and role data variables (lines 7–9), which are initialised with the initial value of the data. The first one pre_state(d) corresponds to the current value of the data right before a call, i.e., where the pre-conditions are evaluated. The second one post_state(d) corresponds to the data value immediately after a call, i.e., where the post-conditions are evaluated for establishing the data-assignments. Both variables are needed because an assignment of some post_state(di) in constraint data-assignments may refer to some pre_state(dj) values.

In lines 13–14, user-defined buffers are produced, one for each consumer port and provided port of the component[3].

[2]Functions are underlined so as to distinguish them from translation routines.

[3]I used ProMeLa's typedef construct to create buffers for consumer/provided ports. The typedef construct is the same as C++'s struct construct and thus used for specifying a data-structure to store some data (e.g., received message via communication channels). See the link

```
1  Port2Promela_Emitter(PrimitiveCInstance comp, EmitterPort port)
2   FORALL event ∈ port.emitterEventSet
3    LET
4    parameters = {event.EventSignature.paramSeq};
5    compICAwait = event.IC_waits.Waits;
6    roleAwait = $\bigwedge_{re \in \underline{roleEventSet}(event)}$ re.IC_waits_ensures.Waits;
7    rolePostEnsures={re.IC_waits_ensures.Ensures | re ∈ roleEventSet(event)};
8    InteractionWaits = roleAwait ∧ compICAwait;
9    IN
10    FORALL fc ∈ event.FC_emitter.EmitterFConsSet
11     ::atomic{
12       true->
13        ContractAssignment2Promela(fc.Promises);
14        FORALL var ∈ omittedParameterVars(e, fc.Promises)
15         select(param: min(var.DataType) .. max(var.DataType));
16        if
17        :: InteractionWaits ->
18          ContractAssignment2Promela(rolePostEnsures);
19          ContractAssignment2Promela(fc.Ensures);
20          FORALL var ∈ updatedVarSet(fc.Ensures ∪ rolePostEnsures)
21           pre_state(var) = post_state(var);
22          channelID(port) ! eventMessage(event);
23        :: else-> goto Start
24        fi
25     }
```

Listing 2: Translating emitter port specifications

These buffers store the received events and method requests. The size of a port buffer is equal to the number of connections that the port has. For instance, if a provided port can receive requests from 3 required ports, then, its buffer size is 3, where one message can be stored from each required port at a time.

Finally, in lines 18–28, the component port behaviours are specified inside an infinite do::od loop of guarded atomic actions. For each port, the corresponding routine is called to construct the guarded actions for its event/method operations. Note also that the beginning of the loop is labelled with the label Start (line 17), which is used to break back to the beginning of the loop in certain cases.

2.2.2. Translating Emitter Ports

The routine in Listing 2 translates an emitter port of a component into ProMeLa code. For each event of the emitter port, a single atomic block is produced from each of the event's functional constraints (lines 11–25). One of the atomic blocks is processed nondeterministically in the loop[4], enabling the non-deterministic choice of one of the event's functional constraints. The block initially calls the ContractAssignment2Promela routine (line 13), which assigns to the event parameters the promised values of the chosen functional constraint (promises clause). Note here that the parameters not included in the promises are assigned nondeterministically to some value within their range (line 14–15). After the event parameters are assigned, then, it is checked whether the component port interaction constraint and its roles' interaction constraints on the event are satisfied or not (line 17). If unsuccessful (line 23), no data update takes place (nor the event emission), and, the control moves back to the Start label (i.e., the beginning of the component loop in Listing 1). If successful, firstly, the data of the roles are assigned new values of the interaction constraint data-assignment (ensures clause) via the ContractAssignment2Promela routine (line 18). Then, the component data are assigned to their new values of the functional constraint data-assignment, calling again the same routine (line 19). Next, the pre-state of each variable is updated with its post-state value for the next method/event operation of the component (lines 20–21). Finally, the event message is emitted to the channel (line 22).

for further information about typedefs: http://spinroot.com/spin/Man/typedef.html

[4]The guard of emitter event blocks is always true for their non-deterministic execution (see line 11 of Listing 2).

```
1  Port2Promela_Consumer(PrimitiveCInstance comp, ConsumerPort port)
2   FORALL event ∈ port.consumerEventSet
3    LET
4    compICAwait =  event.IC_waits_accepts.Waits;
5    compICAccept = event.IC_waits_accepts.Accepts;
6    roleAwait = ⋀re ∈ roleEventSet(event) re.IC_waits_ensures.Waits;
7    rolePostEnsures={re.IC_waits_ensures.Ensures | re ∈ roleEventSet(event)};
8    InteractionWaitsAccepts = roleAwait ⋀ compICAwait ⋀ compICAccept;
9    InteractionReject = roleAwait ⋀ ¬compICAccept;
10   IN
11     ::channelID(port) ? eventMessage(event)->
12         assert(!isEventBufferFull(port));    // check event buffer overflows
13         push(port, eventMessage(event));
14     ::atomic{
15        pop(event, InteractionWaitsAccepts) →
16         ContractAssignment2Promela(rolePostEnsures);
17         if
18         FORALL fc ∈ event.FC_consumer.ConsumerFConsSet
19           ::fc.Requires->
20              ContractAssignment2Promela(fc.Ensures);
21          ::else->printf("incomplete functional constraints"); assert(false);
22         fi
23         FORALL var ∈ updatedVarSet(fc.Ensures ∪ rolePostEnsures)
24          pre_state(var) = post_state(var);
25       }
26     ::pop(event, InteractionReject) →
27        printf("unsafe interaction constraints – chaos;"); assert(false);
```

Listing 3: Translating consumer port specifications

2.2.3. Translating Consumer Ports

The routine in Listing 3 translates a consumer port specification into ProMeLa code. For each event, three blocks are produced. The top first block (lines 11–13) receives event messages from the channel. Then, firstly, the user-defined buffer for the consumer is checked (line 12). If it is full and cannot store the received message, the verification fails due to buffer overflow. Otherwise, the event message is pushed into the consumer buffer (line 13).

The middle block (lines 14–25) is the one that processes the received event messages atomically. The block guard (line 15) enables the block's execution only if an event message can be popped from the user-defined buffer non-deterministically that satisfies its component and role interaction constraints. Upon their satisfaction, firstly, the role data are updated using the role interaction constraints ensures (line 16). Then, the component data are updated using one of the functional constraints (ensures) chosen non-deterministically whose requires pre-condition is satisfied (line 17–22). If however none of the functional constraint pre-conditions are satisfied (i.e., they are incomplete), the verification fails (line 21), indicating that they have been specified erroneously. Finally, having assigned the data variables, the pre-state of each variable is updated with its post-state value for the next method/event operation of the component (lines 23–24).

The last block is produced as shown in lines 26–27. Its guard is satisfied if an event message can be popped from the consumer buffer nondeterministically that violates the event's accepting interaction constraint (if there is any) while the role interaction constraints on the event being satisfied. So, this means that the event cannot be accepted by the consumer; and, the verification fails due to unsafe interaction constraints, which put the component in a chaotic, illegal state.

2.2.4. Translating Required Ports

Required ports are translated as shown in Listing 4. For each required method, two co-dependent atomic blocks are produced from each functional constraint on the method (lines 21–51). One these atomic block pairs is chosen to be processed non-deterministically, which therefore enables the non-deterministic choice of one of the functional constraints. The top block makes a method request to a provided port; and, the bottom treats the response received from the provided port.

The request atomic block (lines 21–34) is enabled if the port has no active method (i.e., those waiting for response).

```
1  Port2Promela_Required(PrimitiveCInstance comp, RequiredPort port)
2   FORALL method ∈ port.requiredMethodSet
3    LET
4    parameters = {method.MethodSignature.ParSeq};
5    compICAwait = method.IC_waits.Waits;
6    roleAwait=⋀rm ∈ roleMethodSet(method) rm.IC_waits_ensures.Waits;
7    rolePostEnsures={rm.IC_waits_ensures.Ensures | rm ∈ roleMethodSet(metho
8    InteractionWaits = roleAwait ⋀ compICAwait;
9
10   UpdatedRoleVarsRace={updatedVarSet(rm.IC_waits_ensures)
11                        | rm ∈ roleMethodSet(method)};
12   UpdatedCVarsRace(fc)={updatedVarSet(subfc.Ensures)
13                        | subfc ∈ fc.requiresEnsuresSet};
14   UpdatedVarSetRace = UpdatedRoleVarsRace ⋃ UpdatedCVarsRace;
15   UsedRoleVarsRace = {usedVarSet(rm.IC_waits_ensures)
16                        | rm ∈ roleMethodSet(method)};
17   UsedCVarsRace(fc)={usedVarSet(subfc) | subfc ∈ fc.requiresEnsuresSet};
18   UsedVarSetRace = UsedRoleVarsRace ⋃ UsedCVarsRace;
19   IN
20   FORALL fc∈method.FC_required.RequiredFConsSet
21     ::atomic{  // sending request
22       activeMethod(port) = null ->
23         ContractAssignment2Promela(fc.Promises);
24         FORALL param ∈ omittedParameterVars(m, fc.Promises)
25           param = select(min(param.DataType), max(param.DataType));
26         if
27         :: InteractionWaits ->
28            activeMethod(port) = method;
29            FORALL var ∈ UsedVarSetRace ∪ UpdatedVarSetRace
30             pre_state_copy(port, var) = pre_state(var);
31            requestChannelID(port) ! methodRequestMessage(method);
32         :: else-> goto Start
33         fi
34       }
35     ::atomic{  // receiving response
36       responseChannelID_cond(port)[activeMethod(port)=method ->0:1] ?
37                                          methodResponseMessage(method)→
38         raceConditionChecking2Promela(port,UpdatedVarSetRace(fc),
39                                          UsedVarSetRace(fc));
40         ContractAssignment2Promela(rolePostEnsures);
41         if
42         FORALL subfc ∈ fc.requiresEnsuresSet
43           :: subfc.Requires ->
44              ContractAssignment2Promela(subfc.Ensures);
45         :: else-> printf("incomplete functional constraints");  assert(false);
46         fi
47         activeMethod(port) = null;
48         FORALL var ∈ updatedVarSet(subfc.Ensures ∪ rolePostEnsures)
49            pre_state(var) = post_state(var);
50            pre_state_copy(port, var) = post_state(var);
51     }
```

Listing 4: Translating required port specifications

So then, the ContractAssignment2Promela routine is used in line 23 for establishing the promised method parameters (i.e., promises clause of the chosen functional constraint). Those parameters that are not assigned in the promises are assigned to some value within their ranges non-deterministically (lines 24-25). After assigning the parameters, it is checked whether the required port's interaction constraint and the role interaction constraints on the event are satisfied or not. If unsuccessful, control moves back to the beginning of the component loop (line 32). If successful, then, the method is recorded as active (line 28), and, the copies of the variables that might suffer from a race-condition are kept, so as to identify these later (lines 29–30). Finally, the request message is emitted via the request channel (line 31).

The response atomic block (lines 35–51) is guarded by the response message that can be received if the current method has already been activated (lines 36–37). Note that to receive the response messages conditionally, the required port's channel array that is introduced in Section 2.2.1. is used. The channel array's particular slot is chosen using ProMeLa's conditional expression operator (C hannelArray[C ond $\rightarrow$ 0 : 1])5 . If the method has been activated (i.e., the condition holds), the channel array's index 0 is chosen that stores the reference to the response channel. If the condition does not hold, the index is chosen 1 and the system execution starts reading from the blocking channel, stored in the index 1, where no message exists actually. This prevents the guard of the atomic block being satisfied. Upon receiving the response for a request, the race-conditions for the component and role data variables are checked via the raceConditionChecking2Promela routine (lines 38–39). If there is no race condition, firstly, the role data are updated via the ContractAssignment2Promela routine (line 40). Following that, the requires–ensures pairs of the functional constraint are evaluated (lines 41–46). One of the pairs is picked nondeterministically among those whose requires pre-condition is met. Using the respective ensures data-assignment, the component data are updated via the ContractAssignment2Promela routine (line 44). If none of the requires pre-conditions is met, the verification fails (line 45) due to incomplete functional constraint pre-conditions. Lastly, in line 47, the method is dis-activated so that another method can be requested. Furthermore, the pre-state and pre-state-copy of each updated data variable are updated with its post-state value for the next method/event operation of the component (lines 48–50).

```
1 Port2Promela_Provided(PrimitiveCInstance comp,ProvidedPort port)
2 FORALL method ∈ port.providedMethodSet
3  LET
4  compICAwait = method.IC_waits_accepts.Waits;
5  compICAccept = method.IC_waits_accepts.Accepts;
6  roleAwait = ⋀rm ∈ roleMethodSet(method) rm.IC_waits_ensures.Waits;
7  rolePostEnsures={rm.IC_waits_ensures.Ensures | rm ∈ roleMethodSet(method)};
8  InteractionWaitsAccepts = roleAwait ⋀ roleAwait_req ⋀ compICAwait
9                            ⋀ compICAccept;
10 InteractionReject = roleAwait ⋀ roleAwait_req ⋀ ¬compICAccept;
11 IN
12 ::requestChannelID(port) ? methodRequestMessage(method)->
13     push(port, methodRequestMessage(method));
14 ::atomic{
15   pop(method, InteractionWaitsAccepts ∧ requestedMethod(port) = null) →
16   ContractAssignment2Promela(rolePostEnsures);   //role method
17      if
18      FORALL fc ∈ method.FC_provided.ProvidedFConsSet
19        ::fc.Requires->
20          ContractAssignment2Promela(fc.Ensures);
21      ::else->printf("incomplete functional constraints"); assert(false);
22      fi;
23      FORALL var ∈ updatedVarSet(fc.Ensures);
24       pre_state_copy(var) = post_state(var);
25       pre_state(var) = post_state(var);
26      responseChannelID(port) ! methodResponseMessage(method);
27   }
28 ::pop(method, InteractionReject) →
29   printf("unsafe interaction constraints – chaos;"); assert(false);
```

Listing 5: Translating provided port specifications

2.2.5. Translating Provided Ports

Like consumer events, provided port methods are each translated into three blocks, shown in lines 11–29 of Listing 5. The top block (lines 12–13) acts similarly to that of consumers. Unlike the consumer translation, it is not checked herein whether the port buffer overflows or not when receiving a method request. Indeed, the buffer of a provided port cannot overflow as required ports cannot make consecutive requests – they have to wait for the response of each request.

[5] See http://spinroot.com/spin/Man/cond_expr.html

The middle block (lines 14–27) processes a method request atomically. The block's guard enables its execution if (i) the request message can be popped from the buffer non-deterministically that satisfies the component and the roles' interaction constraints, and, (ii) the port has no active methods (e.g., processed complex method requests). Upon its satisfaction, the data of the roles that the component plays are updated initially (line 16).

Upon completing the role data updates successfully, one of the functional constraints is chosen non-deterministically whose requires pre-condition is met (lines 17–22). If none of the pre-conditions is satisfied, then, the verification fails again (line 21) due to incomplete functional constraint pre-conditions. If successful, the component data are updated using the functional constraint's ensures paired with the satisfied requires (line 20). Further- more, the ensures are also expected to assign the method \result (unless the method is void type). Otherwise, the \result is assigned to a random value within the range of the method return type. Note that designers could specify an exception via the throws clause instead of the ensures data-assignments. In that case, the specified exception is simply added to the response message for the method. Having processed the method's functional constraint, the pre-state and pre-state-copy values of the data variables are updated with the post-state values for the next method/event operations of the component (lines 23–25). Finally, the response including the result/exception is sent back to the caller via the response channel (line 26).

The bottom block is shown in lines 28–29. Its guard is satisfied if the method request message can be popped from the buffer nondeterministically that violates the method's accepting interaction constraint (if there is any) while the role interaction constraints on the method request being satisfied. This triggers an assertion violation that leads to the failure of the model analysis due to the wrong use of services.

3.0 FORMAL VERIFICATION OF XCD ARCHITECTURES

In this section, XCD's support for formal verification is discussed. Firstly, a case-study on shared-data access is specified in XCD. Then, the properties that XCD supports for verification purposes are introduced and it is shown via the shared-data access case study how XCD architectures can be verified for these properties using the SPIN model checker. Lastly, the verification error types that can be caught through the SPIN model checker are discussed.

3.1. Shared-Data Case Study

In the shared-data system, user components retrieve and update some shared data stored in a memory component. The memory component accepts requests for data retrieval only if the data has been initialised – otherwise, it rejects the request and commences a chaotic behaviour.

The XCD specification of the shared-data access is given in Figure 3. Its elements are explained in the following text.

3.1.1. User Component Type

Component user has a required port puser_r (lines 3–6) through which it makes method calls to its environment (i.e., the memory) to retrieve the value of some data. Port puser_r has a single method get, whose functional contract's **ensures** data-assignments clause (line 4) assigns the method's result to the component data – it has no pre-condition (i.e., a **requires** clause). Component user also has an emitter port puser_e (lines 7–10) to emit events. Port puser_e declares a single event set, whose functional contract **promises** clause assigns its parameter to 7 – the event has no data-assignments (i.e., no **ensures** clause).

3.1.2. Memory Component Type

Component memory has an array of provided ports pmem_p (lines 15–19). It uses each of these ports to provide the method get to a different user component instance. Unlike the contracts of component user, the contract of these ports have an additional **@interaction** part (line 16). This states that the pmem_p port will accept a get method-call only if the component data initialized_m is true. Otherwise, the call is rejected and the component starts behaving in a chaotic manner. If the call is accepted, then the functional contract (line 17) is considered, which sets the result of the method call to be the value of the component sh_data variable. The array of consumer ports pmem_c (lines 20–26) serves to receive set events. Reception of such an event modifies the component state.

```
1 component user(){
2  int data:=0;
3  required port puser_r {
4   @functional{ensures: data:=\result;}
5   int get();
6  }
7  emitter port puser_e {
8   @functional{promises: data_arg:=7;}
9   set(int data_arg);
10  }
11 }
12 component memory(int numOfUsers) {
13  bool initialised_m := false;
14  int sh_data := 0;
15  provided port pmem_p[numOfUsers] {
16   @interaction{accepts: initialised_m;}
17   @functional{ensures: \result:=sh_data;}
18   int get();
19  }
20  consumer port pmem_c[numOfUsers] {
21   @functional{
22    ensures: initialised_m := true;
23             sh_data := data_arg; }
24   set(int data_arg);
25  }
26 }
27 component sharedData() {
28  component user userIns1();
29  component user userIns2();
30  component memory memoryIns(2);
31  connector memory2user x1(userIns1{puser_r,puser_e},memoryIns{pmem_p[0],pmem_c[0]});
32  connector memory2user x2(userIns2{puser_r,puser_e},memoryIns{pmem_p[1],pmem_c[1]});
33 }
34 component sharedData configuration();

36 connector memory2user(
37      userRole{pvuser_r,pvuser_e},
38      memoryRole{pvmem_p,pvmem_c}) {
39  role userRole {
40   required port pvuser_r {
41    int get();
42   }
43   emitter port pvuser_e {
44    set(int data_arg);
45   }
46  }
47  role memoryRole {
48   bool initialised := false;
49   provided port pvmem_p {
50    @interaction{ waits: initialised;}
51    int get();
52   }
53   consumer port pvmem_c {
54    @interaction{
55     ensures: initialised := true;}
56    set(int data_arg);
57   }
58  }
59  connector user2memory_m(
60   userRole{pvuser_r},memoryRole{pvmem_p});
61  connector user2memory_e(
62   userRole{pvuser_e},memoryRole{pvmem_c});
63 };
```

Figure 3. Specification of shared-data access in XCD

3.1.3. Memory2User Connector Type

Connector type memory2user (lines 36–63 of Figure 3) specifies the protocol used in the system between the memory and the users. It guarantees that the memory will not behave chaotically. The connector has two roles, userRole (lines 39–46) and memoryRole (lines 47–58). The role userRole has a required port-variable pvuser_r (lines 40–42), reflecting the port puser_r of the component user, and an emitter port-variable pvuser_e (lines 43–46), reflecting the port puser_e. These port-variables do not impose any interaction constraints on the role.

The role memoryRole has a provided port-variable pvmem_p (lines 49–52) reflecting the port pmem_p of the component memory. Unlike the port-variables of the userRole, this port-variable introduces extra interaction constraints on the behaviour of its methods. It requires that calls to the method get are considered only when the role's initialized data is true, thus delaying them while this condition is not satisfied.

The role's consumer port-variable pvmem_c (lines 53–57) reflects the port pmem_c of the component memory. It uses its interaction contract to note that the memory has been set, through its **ensures** clause. The combination of the contracts of the two ports means that the memory cannot start behaving chaotically, as requests at non-accepting states are delayed until they are safe.

3.1.4. SharedData Composite Component Type

The sharedData component type (lines 27–33 of Figure 3) includes two instances of the user component and a single instance of the memory component. The component instances are passed as arguments to the two connector instances, in lines 31–32, to bind them together and constrain their interactions.

3.2. Checking Model Correctness via SPIN

The ProMeLa language is supported by the SPIN model checker [16], which exhaustively checks formal ProMeLa models to prove their correctness. I use the SPIN model checker to verify the correctness of system configurations, each of which describes a group of components interacting via some connectors to compose a system. Through the verification of a system configuration, I aim at detecting whether the components can be composed successfully in the way specified in the configuration and check for: (i) wrong use of services, (ii) incomplete functional behaviours of components, (iii) race conditions, and (iv) deadlocks. These properties are discussed in the rest of this section.

```
spin -a configuration.pml
gcc -O2 -DMEMLIM=7024 -DSAFETY -o pan pan.c
./pan -m50000
```

Listing 6: Commands for SPIN verification

Model Configuration	State-vector (in Bytes)	States		Memory (in MB)	Time (in sec)
		Stored	Matched		
1 user	156	477	284	128	0.00
2 users	248	169380	248188	163	0.3
3 users	344	16156062	39898631	4701	41
4 users	436	19630407	65378729	7024†	57.7
BITSTATE 4 users	436	62680212	1.9209748e+08	16	226

Spin (version 6.2.4) and gcc (version 4.7.2) used.
For bit-state verification, the -DBITSTATE option needs to be passed to gcc.
Using a 64bit Intel Xeon CPU (W3503 @ 2.40GHz × 2), 11.7GB of RAM, and Linux version 3.5.0-39-generic.
Column "States Stored" shows the number of unique global system states stored in the state-space, while column "States Matched" the number of states that were revisited during the search - see: spinroot.com/spin/Man/Pan.html#L10
† Cases marked with † in the Memory column run out of memory.

Table 1. Verification results for 4 different configurations of shared-data

3.2.1. Checking Wrong Use of Services and Behaviour Incompleteness

As discussed in [29], consumer and provided ports may receive event and method requests respectively at unacceptable states, violating their accepting interaction contracts. This causes chaos, indicating the use of actions in the wrong order. Moreover, even if requests are received at acceptable states, the functional contracts may not be complete. This occurs when none of the functional pre-conditions is satisfied. While the former indicates the wrong use of services, the latter indicates the wrongly specified contracts. In both scenarios, the verification of a system configuration fails. This is encoded in XCD's semantics [29] as an assertion violation. Therefore, the translated ProMeLa models via XCD's compiler abort their execution. To use services correctly, user components must always request method/event actions when the requests are expected by the components offering the actions and satisfy their accepting interaction contracts (e.g., a server expecting its service requests when its connection is opened). To specify the functional constraints of a method/event correctly (i.e., complete), designers must consider all possible cases that the component can be in once the method/event request is accepted (e.g., the functional behaviour of sqrt(x) method considering not only the case when $x \geq 0$ but also $x < 0$).

Designers can use the ispin GUI of the SPIN model checker to perform formal verification via a graphical tool[6]. Alternatively, the SPIN model checker can be used over a command line. Then, the set of commands that can be used to verify for assertion violations is shown in Listing 6. Note that I specified the memory limit as 7024MB for formal verifications (i.e., -DMEMLIM=7024 in line 2 of Listing 6), and the maximum search depth as 50.000 (-m50000) – these can be changed to other values.

I run the commands in Listing 6 for the verification of the shared-data specification, given in Figure 3, and successfully verified it for the absence of chaotic and incomplete behaviours – no assertion violation reported. The verification results are displayed in Table 1 for four different configurations of the shared-data, each varying by the number of users involved. So, this means that users always request methods and events of the memory in the correct order (i.e., the interaction contracts are satisfied). Moreover, since the method and event functional contracts do not have the **requires** pre-conditions (i.e., these are true), they are complete by definition.

Note that when memory proves insufficient during the formal verification (marked with a † in Table 1), designers can use instead SPIN's bit-state hashing mode [15], which uses Bloom filters [4] to reduce memory drastically.

[6]See the following link for installation information of ispin: http://spinroot.com/spin/Man/README.html.

```
[bufferLength=100]
consumer port samplePort{......}
```

Listing 7: Attribute for specifying consumer buffer size

3.2.2. Checking Race Conditions

Race condition is the commonly observed problem of concurrent software systems. As discussed in [29], XCD's ProMeLa semantics consider the detection of race conditions. Indeed, race conditions may occur in system behaviours specified with XCD because XCD components execute their ports concurrently. So, when multiple ports of the same component perform their method/event actions concurrently, accessing and updating the component state in an arbitrary order, the component may then have race conditions, which leave the component at an inconsistent state.

Race conditions may occur in the case of required methods, whose execution consists of non-atomic request and response parts. That is, whenever a component makes a request via its required port, the same component may operate its other port(s) until the response is received.

Race conditions are indicated with an assertion violation in the ProMeLa models (just like chaos and incomplete behaviours). So, designers can use the verification commands given in Listing 6 for detecting race conditions too.

Having verified the shared-data specification using the commands in Listing 6, I essentially guaranteed the absence of race conditions, apart from the absence of chaotic and incomplete behaviours. Race condition is in fact not possible in the shared-data system. This is because the user's required port puser_r requests the method get and, upon receiving the response, the component state is updated using the method's functional contract **ensures**. The **ensures** assigns the received result to the data, which is however not updated by the functional constraint of the user's emitter event.

3.2.3. Checking Buffer Overflow for Consumer Ports

In XCD, consumer and provided ports of components store their received requests in a buffer, where the requests can be obtained and processed [29, 30]. However, consumer buffers may overflow, as the emitter events can be emitted asynchronously without waiting for a response. Note that provided buffers never overflow because required ports wait for the method response of each request before making another request.

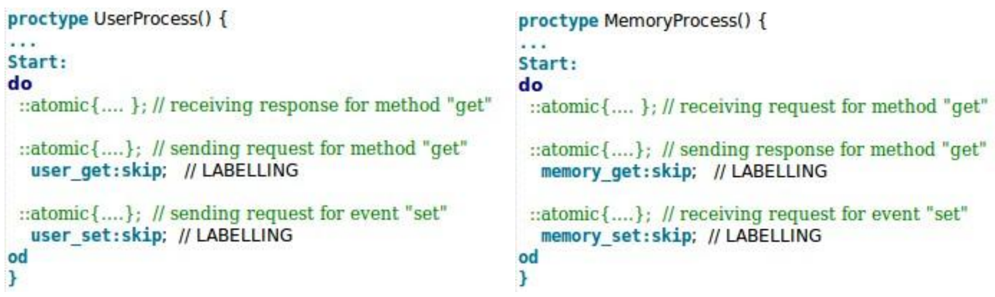
Checking buffer overflow is encoded in the semantics of consumer ports, which are mapped as an assertion violation in ProMeLa. So, translating their XCD specifications into ProMeLa model via XCD's compiler, designers can verify their system configurations for the absence of the event buffer overflows.

Event buffer overflows can be dealt with in two ways. The repetitive emission of events, causing the consumer buffer overflow, can be prevented by modifying the protocol contracts. Alternatively, designers may choose to increase the size of the consumer buffer. This is done in XCD as illustrated in Listing 7, where the bufferLength precedes the consumer port specification and denotes the desired new size of that consumer port. By doing so, the default buffer size (i.e., 1) can be replaced by the compiler with the desired bufferLength.

I checked the shared-data, specified in Figure 3, for consumer buffer overflow and got a verification error by the SPIN model checker. The error is due to the consumer port pmem_c of the memory, whose buffer overflows with the user events. Indeed, it is easy to understand from the shared-data specification that the event set can be emitted repetitively by the user without any delaying interaction contracts.

3.2.4. Checking Deadlocks

Deadlock is one of the most common properties that concurrent systems are verified against. The SPIN model checker warns designers automatically when a deadlock occurs globally that stops the system components from operating. It halts the formal verification with an invalid end state error. The invalid end state indicates that a system execution terminates at a state where the component processes are not able to reach their end state and complete their operations. For deadlock verification, designers can use the command set given in Listing 6.



```
proctype UserProcess() {
...
Start:
do
 ::atomic{.... }; // receiving response for method "get"

 ::atomic{....};  // sending request for method "get"
   user_get:skip;   // LABELLING

 ::atomic{....};  // sending request for event "set"
   user_set:skip;  // LABELLING
od
}
```

(a) Labelling user process

```
proctype MemoryProcess() {
...
Start:
do
 ::atomic{.... }; // receiving request for method "get"

 ::atomic{....};  // sending response for method "get"
   memory_get:skip;   // LABELLING

 ::atomic{....};  // receiving request for event "set"
   memory_set:skip;  // LABELLING
od
}
```

(b) Labelling memory process

Figure 4. Process labels for tracing the executions of shared-data users and memory

```
ltl sharedData_liveness{
□ ( (instance_name(userIns1)@user_get || instance_name(userIns2)@user_get)
                                 -> ◇ instance_name(memoryIns)@memory_get)
}
```

Listing 8: *LTL* specification of a liveness property for shared-data

```
ltl sharedData_safety{
 !instance_name(memoryIns)@memory_get U instance_name(memoryIns)@memory_set
}
```

Listing 9: *LTL* specification of a safety property for shared-data

I successfully verified the shared-data configuration for the absence of deadlock. So, the users and memory components interact with each other without getting blocked indefinitely.

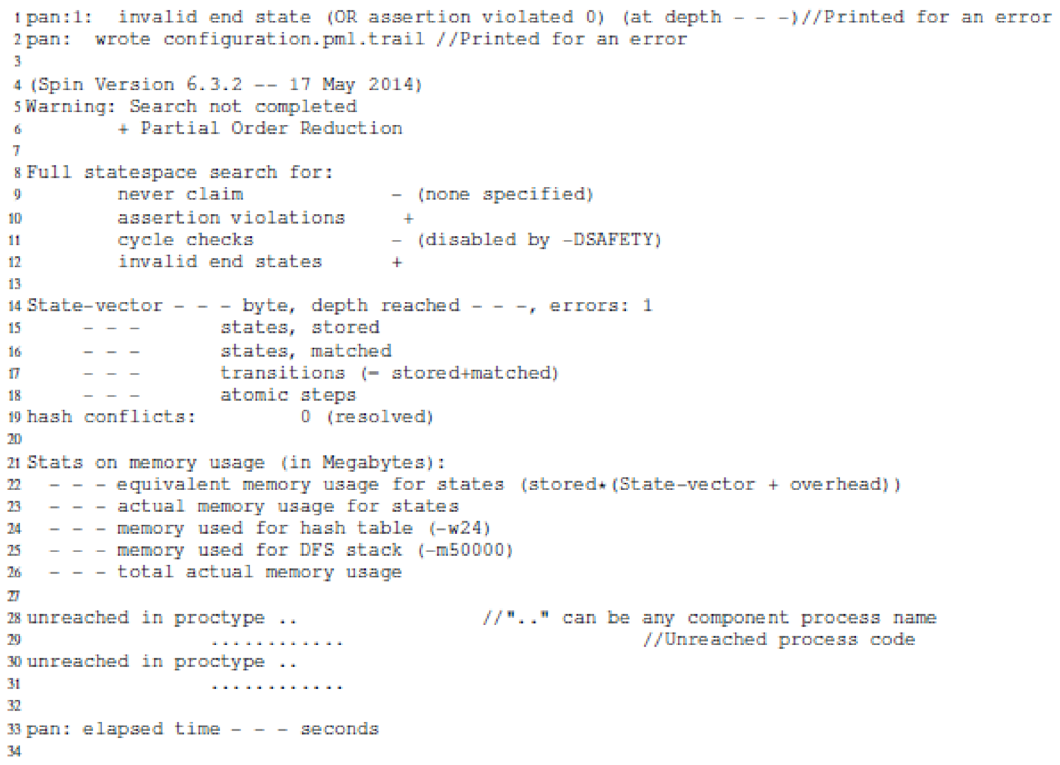
3.2.5. Checking System Properties

Designers may want to verify their system behaviours for high-level system requirements, e.g., the shared-data must always be initialised first before any user access. While XCD does not yet provide a (sub) language to specify system properties for such system requirements, it is still possible by using the ProMeLa language's notation for the translated ProMeLa models of XCD architectures.

Linear Temporal Logic (LTL) ProMeLa offers Linear Temporal Logic (LTL) [32] construct, through which designers can specify safety and liveness properties of their systems via temporal operators (e.g., , U, and). To facilitate the use of LTL for the transformed ProMeLa models of XCD architectures, XCD's prototype tool further adds labels to each atomic block transformed from the component port actions. These action labels aid in identifying whether the actions of component ports are executed or not (i.e., the labelled state is reached). So, using the labels, designers can specify LTL properties on the execution of port actions.

Figure 4 shows the action labels for the user and memory component processes of the shared-data. Using these labels, one can identify at any time whether these labelled states are reached, and, thus, the set and get actions are executed. For instance, Listing 8 gives a liveness property that I specified for the shared-data inside its configuration mapping (i.e., configuration.pml) using ProMeLa's ltl. It basically checks that whenever one of the two user instances in the configuration (specified in lines 27–33 of Figure 3) requests the method get, the memory instance eventually processes the request and sends back the response. Another ltl is given in Listing 9, where a safety property is specified for checking that the memory processes the event set before receiving and processing the method get.

I used the SPIN commands given in Listing 6 and verified the shared-data configuration successfully for the LTL properties.



```
1  pan:1:  invalid end state (OR assertion violated 0) (at depth - - -)//Printed for an error
2  pan:  wrote configuration.pml.trail //Printed for an error
3
4  (Spin Version 6.3.2 -- 17 May 2014)
5  Warning: Search not completed
6          + Partial Order Reduction
7
8  Full statespace search for:
9          never claim             - (none specified)
10         assertion violations      +
11         cycle checks            - (disabled by -DSAFETY)
12         invalid end states      +
13
14 State-vector - - - byte, depth reached - - -, errors: 1
15      - - -        states, stored
16      - - -        states, matched
17      - - -        transitions (= stored+matched)
18      - - -        atomic steps
19 hash conflicts:          0 (resolved)
20
21 Stats on memory usage (in Megabytes):
22   - - - equivalent memory usage for states (stored*(State-vector + overhead))
23   - - - actual memory usage for states
24   - - - memory used for hash table (-w24)
25   - - - memory used for DFS stack (-m50000)
26   - - - total actual memory usage
27
28 unreached in proctype ..                //".." can be any component process name
29                  ............                    //Unreached process code
30 unreached in proctype ..
31                  ............
32
33 pan: elapsed time - - - seconds
34
```

Figure 5. SPIN's verification report template

The places denoted with "- - -" are to be filled with some experimental values obtained during formal verifications performed via the SPIN model checker.

3.3. Dealing with Verification Errors in SPIN

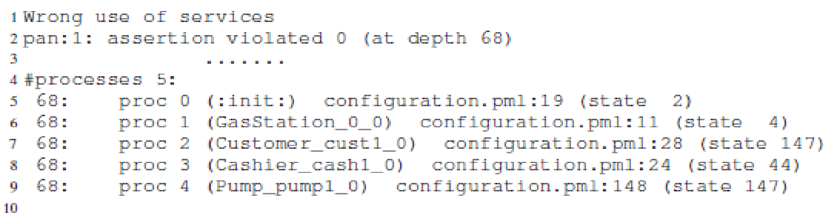
So far, I introduced the property types that are supported by XCD's semantics and checked via the SPIN model checker automatically. However, I have not yet shown how to deal with the SPIN verification errors occurring due to the violation of these properties. Now, I show how designers can understand which property is violated when they encounter a verification error in their SPIN verification and how designers can inspect property violations to find out its cause.

3.3.1. SPIN Verification Result

After each verification, the SPIN model checker produces the verification report depicted in Figure 5. The verifi- cation report includes information about the state space of the verified system that has been explored exhaustively during the verification. The report gives in lines 14–18 of Figure 5 (i) the vector size of each state, (ii) the reached depth of the explored state space, (iii) the number of stored and matched states, (iv) the number of stored state transitions, and (v) the number of taken atomic steps. The details about the memory that is used to store the state space are also given in lines 21–26. Furthermore, the verification report may also include unreachable code for the component processes, which are given at the end of the verification report in lines 28–31. Unreachable code repre- sent the ProMeLa code that can never be executed. Note that Table 1 given in Section 3.2.1. (page 12) essentially represents the verification reports resulted from the verification of the shared-data system configurations.

3.3.2. Verification Error Types in SPIN

Besides providing information about the explored state space, the verification report lets designers know whether the verification was successful or not. When an error is caught during the formal verification, SPIN's model checker halts the verification at the point where the error is caught and reports the verification error in the first line of the verification report – see line 1 of Figure 5. As aforementioned, the error can be either an invalid end state or an assertion violation. The former indicates a deadlocking system behaviour. The latter indicates the violation of some pre-defined properties, i.e., wrong use of services, incomplete functional behaviours, event buffer overflow, and race conditions. In the occurrence of errors, designers can run the SPIN command "$./pan -r" to obtain the error trail, which can be gone through to identify what causes the error.



```
1 Wrong use of services
2 pan:1: assertion violated 0 (at depth 68)
3              .......
4 #processes 5:
5  68:     proc 0 (:init:)  configuration.pml:19 (state  2)
6  68:     proc 1 (GasStation_0_0)  configuration.pml:11 (state  4)
7  68:     proc 2 (Customer_cust1_0)  configuration.pml:28 (state 147)
8  68:     proc 3 (Cashier_cash1_0)  configuration.pml:24 (state 44)
9  68:     proc 4 (Pump_pump1_0)  configuration.pml:148 (state 147)
10
```

Figure 6. An example error trail - assertion violation error

Inspecting SPIN's Error Trace for Assertion Violation Error In the case of an assertion violation error, the error trace gives the sequence of ProMeLa code that lets identify the code each component process executes when the error occurs. To illustrate this, let us consider Figure 6 that gives the error trail of a system with customer, cashier, and pump components. Line 1 always indicates the reason for the assertion violation. Apparently, the assertion violation in this instance results from the wrong use of component services. Lines 5–9 indicates the exe- cuted ProMeLa code of each unique component process when the assertion violation occurs. It shows respectively (i) the id of the process (e.g., proc 0), (ii) the full name of the component consisting of the type name, instance name[7] and the instance index[8] (e.g., Customer_cust1_0), and lastly (iii) the line number of the component pro- cess code that has been executed at that point (e.g., configuration.pml:11). Designers can use these information to locate the cause of the assertion violation. For instance, following line 9 of the error trail, one can inspect the pump component's process, whose code in line 148 indicates that the error is due to the pump's particular method requested at an unacceptable state. Note also that the location information may sometimes be supplemented with the exact ProMeLa code in that location, especially if it is a channel I/O operation. This liberates designers from having to search the code in the process files of the components.

Inspecting SPIN's Error Trace for Invalid End State Error In the case of an invalid end state error, the error trail is supposed to give the sequence of ProMeLa channel I/O operations that cannot be executed by the component processes and thus causes the components to get blocked indefinitely. To illustrate this, let us consider a very simple software architecture specified in Figure 7. Therein, the Client1 and Client2 emit events to each other under no constraints. However, their interactions are deadlocking, indicated via the invalid end state error that has been reported during the formal verification. The error trail shown in Figure 8 includes the ProMeLa code for the Client1 and Client2 processes that cannot be executed. Lines 7–8 give the ProMeLa code for the Client1 process, while lines 9–10 give the code for the Client2 process. Apparently, the deadlock occurs due to that the former is stuck trying to emit event1 and the latter is stuck emitting event3.

4.0 RELATED WORK

To the best of our knowledge, XCD is the only ADL that promotes (i) the modular and contractual specification of software architectures and (ii) the automated formal verification of software architectures for a rich set of properties. In this section, the related work is discussed in three parts, which I believe aid a lot in showing XCD's novelty among many ADLs.

Modularity and Reusability Current ADLs can be grouped based on their support for (complex) connectors: those that are inspired from Darwin [22] and ignore the first-class specification of connectors and those that are inspired from Wright [1] and separate connectors from components. Thus, Darwin-inspired languages, such as UniCon [36], Rapide [21], LEDA [5], AADL [10], and RADL [34], do not promote modular and re-usable software architectures. Using Darwin, designers cannot separate interaction protocols from components and thus cause protocol-dependent components that may not be re-used in different contexts. Just like XCD, Wright-inspired languages, such as CONNECT [17] and PiLar [33], support complex connectors and separate interaction protocols from components. This allows designers to specify modular and thus re-usable software architectures that can more easily be designed and analysed.

[7] In the case of the configuration composite component, the error trail does not show the instance name, e.g., GasStation_0_0
[8] Single components are assigned index 0 while the components of component arrays are assigned their own index in the array.

```
component Client1(){
 emitter port ReqInterface1{ service1();}
 consumer port OfferInterface1{service2();}
}
component Client2(){
 emitter port ReqInterface2{service2();}
 consumer port OfferInterface2{service1();}
}
connector C1xC2(Client1{ReqInterface1,OfferInterface1},Client2{ReqInterface2,OfferInterface2}){
 role Client1{
  emitter port_variable ReqInterface1{service1();}
  consumer port_variable OfferInterface1{service2();}
 }
 role Client2{
  emitter port_variable ReqInterface2{service2();}
  consumer port_variable OfferInterface2{service1();}
 }
 connector async link1(Client1{ReqInterface1},Client2{OfferInterface2} );
 connector async link2(Client2{ReqInterface2}, Client1{OfferInterface1});
}
component configuration(){
 component Client1 C1inst();
 component Client2 C2inst();
 connector C1xC2  connIns(C1inst{ReqInterface1,OfferInterface1},
                          C2inst{ReqInterface2,OfferInterface2});
}

```

Figure 7. An example software architecture with deadlocking behaviour

```
                 .......
#processes 4:
309:    proc 0 (:init:)  configuration.pml:19 (state  2)
                 -end-
309:    proc 1 (configComp_0_0)  configuration.pml:8 (state  3)
                 -end-
309:    proc 2 (C1_c1inst_0)  configuration.pml:123 (state 117) (invalid end state)
                 CHANNEL_configComp_0_COMPONENT_C2_c2inst_0_PORT_eventPort_cons2[0]!event1
309:    proc 3 (C2_c2inst_0)  configuration.pml:123 (state 117) (invalid end state)
                 CHANNEL_configComp_0_COMPONENT_C1_c1inst_0_PORT_eventPort_cons1[0]!event3

```

Figure 8. The error trail for the verification of the XCD architecture in Figure 7

Contractual, Non-algebraic, Behaviour Specifications To enable the formal verification of software architectures, ADLs base their notation to process algebras that are supported by analysis tools such as model checkers. For instance, Wright is based on CSP [13], Darwin is based on FSP [23], and LEDA is based on π-calculus [27]. However, process algebras are found by practitioners as requiring a steep learning curve [24]. To minimise the learning curve, XCD is based on the well-known Design-by-Contract approach [26], which is already popular among practitioners thanks to languages such as Java Modeling Language [20].

Formal Verification As aforementioned, there are many algebraic ADLs that support formal verification through model checkers. These ADLs commonly aim at checking for a system specification whether its components can be composed successfully to construct the entire system or not. That is, it is checked whether the system components behave in a way that is compatible with the connected components. One of the most popular languages in formal verification, the Wright ADL [1] lets designers verify software architectures for deadlocking protocols, incompatibilities between connector protocols and component interaction behaviours, and consistencies between the computational behaviours of components and their interaction behaviours. CONNECT [17], inspired from Wright, forces designers to specify component and connector behaviours in FSP. Designers can use the LTSA analysis tool for deadlock (and liveness) verification. However, unlike Wright, CONNECT does not offer any different properties. Another popular language, Darwin [12] has been extended through the Tracta approach so that the behaviours of components can be specified in FSP process algebra and verified for some properties graphically specified using finite state automata. LEDA [5] is inspired from Darwin in its view of software architectures as a composition of components. LEDA supports the specification of component behaviours in π−calculus. It is also possible with LEDA to verify that the behaviours of components composing a system are compatible with each other. Unlike Darwin, LEDA does not offer any notations for specifying user-defined properties.

XCD offers various types of properties for which designers can verify their architecture specifications automatically. Unlike other languages, in XCD component behaviours are specified modularly in terms of functional and interaction behaviours. Therefore, during the verification, designers can determine incomplete functional behaviours, i.e., the components which perform unexpected computations. Designers can also determine the components that interact with their environments unexpectedly, i.e., using the services of each other in the wrong order. Moreover, thanks to connectors in XCD describing the interaction protocols for components, designers can easily analyse the impact of different protocols on component behaviours. By doing so, designers can evaluate different design decisions on the component protocols and find out the optimal one. Designers can also check whether components interacting under the protocols of connectors suffer from deadlock, race conditions, and event buffer overflow (in the case of asynchronous event communications). Besides the pre-defined properties, in XCD designers can specify their own properties in the form of ltl formulas and verify their correctness.

5. CONCLUSION

Software architectures are very crucial in describing software systems at a high-level in terms of components that interact with each other through some connectors. There are many architecture description languages for specifying software architectures and performing operations on them such as formal verification. XCD is one of these languages, which is distinguished with its support for contractual and modular architecture specifications. In XCD, components are specified in terms of their functional and interaction behaviours, while connectors representing the interaction protocols for the components. XCD's semantics are defined by showing how XCD components and connectors can be translated in the ProMeLa formal verification language accepted by the SPIN model checker. Thanks to XCD's compiler tool, designers can translate their XCD architectures into ProMeLa model and verify their architectures for a rich set of properties using the SPIN model checker. These properties are: (i) wrong use of services, (ii) incomplete functional behaviours, (iii) race-conditions (iv) deadlock, and (v) event buffer overflow in the case of asynchronous communications. In this paper, I introduce XCD and its support for formal verification of software architectures via the shared-data case study. I showed what each of the above properties represents and how XCD architectures can be verified for these properties using the SPIN's model checker. I also discussed how designers can deal with verification errors caught by SPIN's model checker.

Currently, I am working on a visual notation set for XCD so as to better meet the needs of practitioners who seem to be more interested in visual ways of specifying software architectures. The visual notation set will also be supported by a toolset that consists of a drawing editor and a compiler. Using this toolset, designers can specify their XCD architectures visually and translate them into textual XCD for formal verification purposes.

REFERENCES

[1] ALLEN, R., AND GARLAN, D. A formal basis for architectural connection. ACM Trans. Softw. Eng. Methodol. 6, 3 (1997), 213–249.

[2] BARNETT, M., LEINO, K. R. M., AND SCHULTE, W. The spec# programming system: an overview. In Proceedings of the 2004 international conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (Berlin, Heidelberg, 2005), CASSIS'04, Springer-Verlag, pp. 49–69.

[3] BERGSTRA, J. A. Handbook of Process Algebra. Elsevier Science Inc., New York, NY, USA, 2001.

[4] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. Commun. ACM 13, 7 (1970), 422–426.

[5] CANAL, C., PIMENTEL, E., AND TROYA, J. M. Specification and refinement of dynamic software architectures. In WICSA (1999), P. Donohoe, Ed., vol. 140 of IFIP Conference Proceedings, Kluwer, pp. 107–126.

[6] CHALIN, P., KINIRY, J. R., LEAVENS, G. T., AND POLL, E. Beyond assertions: advanced specification and verification with jml and esc/java2. In Proceedings of the 4th international conference on Formal Methods for Components and Objects (Berlin, Heidelberg, 2006), FMCO'05, Springer-Verlag, pp. 342–363.

[7] CLEMENTS, P. C. A survey of architecture description languages. In Proceedings of the 8th International Workshop on Software Specification and Design (Washington, DC, USA, 1996), IWSSD '96, IEEE Computer Society, pp. 16–.

[8] CLEMENTS, P. C., GARLAN, D., LITTLE, R., NORD, R. L., AND STAFFORD, J. A. Documenting software architectures: Views and beyond. In ICSE (2003), L. A. Clarke, L. Dillon, and W. F. Tichy, Eds., IEEE Computer Society, pp. 740–741.

[9] DIJKSTRA, E. W. Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM 18, 8 (1975), 453–457.

[10] FEILER, P. H., GLUCH, D. P., AND HUDAK, J. J. The Architecture Analysis & Design Language (AADL): An Introduction. Tech. rep., Software Engineering Institute, 2006.

[11] GARLAN, D., AND SHAW, M. An introduction to software architecture. Tech. rep., Pittsburgh, PA, USA, 1994.

[12] GIANNAKOPOULOU, D., KRAMER, J., AND CHEUNG, S.-C. Behaviour analysis of distributed systems using the tracta approach. Autom. Softw. Eng. 6, 1 (1999), 7–35.

[13] HOARE, C. A. R. Communicating sequential processes. Commun. ACM 21, 8 (1978), 666–677.

[14] HOLZMANN, G. J. The spin model checker. IEEE Trans. on Software Engineering 23, 5 (May 1997), 279–295.

[15] HOLZMANN, G. J. An analysis of bitstate hashing. Formal Methods in System Design 13, 3 (1998), 289–307.

[16] HOLZMANN, G. J. The SPIN Model Checker - primer and reference manual. Addison-Wesley, 2004.

[17] ISSARNY, V., BENNACEUR, A., AND BROMBERG, Y.-D. Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability. In SFM (2011), M. Bernardo and V. Issarny, Eds., vol. 6659 of Lecture Notes in Computer Science, Springer, pp. 217–255.

[18] IVERS, J., CLEMENTS, P., GARLAN, D., NORD, R., SCHMERL, B., AND SILVA, J. R. O. Documenting component and connector views with UML 2.0. Tech. Rep. CMU/SEI-2004-TR-008, Software Engineering Institute (Carnegie Mellon University), 2004.

[19] KINIRY, J. R., AND ZIMMERMAN, D. M. Secret ninja formal methods. In FM (2008), J. Cuéllar, T. S. E. Maibaum, and K. Sere, Eds., vol. 5014 of Lecture Notes in Computer Science, Springer, pp. 214–228.

[20] LEAVENS, G. T., BAKER, A. L., AND RUBY, C. JML: A notation for detailed design. In Behavioral Specifications of Businesses and Systems, H. Kilov, B. Rumpe, and I. Simmonds, Eds., vol. 523 of The Kluwer International Series in Engineering and Computer Science. Springer, 1999, pp. 175–188.

[21] LUCKHAM, D. C. Rapide: A language and toolset for simulation of distributed systems by partial orderings of events. Tech. rep., Stanford University, Stanford, CA, USA, 1996.

[22] MAGEE, J., AND KRAMER, J. Dynamic structure in software architectures. In SIGSOFT FSE (1996), pp. 3–14.

[23] MAGEE, J., AND KRAMER, J. Concurrency - state models and Java programs (2. ed.). Wiley, 2006.

[24] HUANG, L.B., BALAKRISHNAN V. AND RAJ R. G. "Improving the relevancy of document search using the multi-term adjacency keyword-order model." Malaysian Journal of Computer Science 25,1 (2012) 1-10.

[25] MEDVIDOVIC, N., AND TAYLOR, R. N. A classification and comparison framework for software architecture description languages. IEEE Trans. Software Eng. 26, 1 (2000), 70–93.

[26] MEYER, B. Applying "Design by Contract". IEEE Computer 25, 10 (1992), 40–51.

[27] MILNER, R., PARROW, J., AND WALKER, D. A calculus of mobile processes, i. Inf. Comput. 100, 1 (1992), 1–40.

[28] OZKAYA, M., AND KLOUKINAS, C. Realizable, connector-driven software architectures for practising engineers. In Software Technologies - 8th International Joint Conference, ICSOFT 2013, Reykjavik, Iceland, July 29-31, 2013, Revised Selected Papers (2013), J. Cordeiro and M. van Sinderen, Eds., vol. 457 of Com- munications in Computer and Information Science, Springer, pp. 273–289.

[29] OZKAYA, M., AND KLOUKINAS, C. Design-by-contract for reusable components and realizable architectures. In Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering (New York, NY, USA, 2014), CBSE '14, ACM, pp. 129–138.

[30] OZKAYA, M., AND KLOUKINAS, C. Realizable, connector-driven software architectures for practising engineers. In Software Technologies, J. Cordeiro and M. van Sinderen, Eds., vol. 457 of Communications in Computer and Information Science. Springer Berlin Heidelberg, 2014, pp. 273–289.

[31] PERRY, D. E., AND WOLF, A. L. Foundations for the study of software architecture. SIGSOFT Softw. Eng. Notes 17, 4 (Oct. 1992), 40–52.

[32] PNUELI, A. The temporal logic of programs. In FOCS (1977), IEEE Computer Society, pp. 46–57.

[33] QUINTERO, C. E. C., DE LA FUENTE, P., BARRIO-SOLÓRZANO, M., AND GUTIÉRREZ, M. E. B. Coordination in a reflective architecture description language. In COORDINATION (2002), F. Arbab and C. L. Talcott, Eds., vol. 2315 of Lecture Notes in Computer Science, Springer, pp. 141–148.

[34] REUSSNER, R., POERNOM O, I., AND SCHM IDT, H. Reasoning about Software Architectures with Contractually Specified Components. In Component-Based Software Quality, A. Cechich, M. Piattini, and A. Valle- cillo, Eds., vol. 2693 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, p. 287–325.

[35] RUMBAUGH, J. E., JACOBSON, I., AND BOOCH, G. The unified modeling language reference manual. Addison-Wesley-Longman, 1999.

[36] SHAW, M., DELINE, R., KLEIN, D. V., ROSS, T. L., YOUNG, D. M., AND ZELESNIK, G. Abstractions for software architecture and tools to support them. IEEE Trans. Software Eng. 21, 4 (1995), 314–335.