MAPPING SCDL/BPEL TO ADA FOR FORMAL VERIFICATION OF THE BEHAVIORAL PROPERTIES OF SERVICE-COMPONENT ARCHITECTURE

Sakka Rouis Taoufik[1], Bhiri Mohamed Tahar[2] and Kmimech Mourad[3]

[1]LIPAH Laboratory, FST, University of Tunis El Manar, Tunisia
[2]Miracl Laboratory, ISIMS, Technological Pole of Sfax, Tunisia
[3]UR-OASIS Laboratory, ENIT, University of Tunis El Manar, Tunisia

ABSTRACT

The Service Component Definition Language (SCDL) and the Web Service Business Process Execution Language (WS-BPEL) are the standards de-facto used in the modeling and implementing of Service-Component Architecture (SCA). However, these powerful languages lack a formal foundation for the specification and verification of the SCA properties. In this study, the use of Wright formal ADL and Ada programming language was proposed to check the behavioral properties of SCDL/WS-BPEL Service-Component architectures. To achieve this, the mapping of SCDL/WS-BPEL to the Wright formal ADL was suggested in order to verify the standard behavioral consistency of the source description. As a second step, the target specification could be transformed into Ada to check the specific and dynamic behavioral properties of the SCDL/WS-BPEL source architecture.

KEYWORDS

Service-Component Architecture, Behavioral Verification, SCDL, WS-BPEL, Model-Checker, FDR2, Ada Concurrent Program

1. INTRODUCTION

The Service-Component Architecture (SCA) technology [19] combines the advantages of component-based approach and service-oriented approach. Indeed, the main idea behind this technology is to be able to build distributed applications which are independent of implementation technology and protocol. SCA software architecture is generally described by a composite of services-components written by the Service Component Definition Language (SCDL) [19]. The latter is an XML based formatted language which allows expressing all the relations in this SCA composite. The communication mechanisms of this SCA composite can be implemented by WSDL interfaces, Java class or WS-BPEL [3] process. Where WS-BPEL (Web-Service Business Process Execution Language) represents a convergence of two languages to compose web services: WSFL [17] of IBM and XLANG [21] of Microsoft. This WS-BPEL (abbr. BPEL) language offers a standard- based approach to build flexible business processes by the choreographing and orchestrating of multiple Web services. In addition, it aims to model the behavior of component processes by specifying both executable and abstract business processes. It also defines an interoperable integration model that should facilitate the expansion of automated process integration both within and between businesses.

For several years now, the SCDL [19] and WS-BPEL [3] technologies appear as powerful complementary models for the development of service-component architectures. However, they lack a formal foundation for the specification and verification of their structural, behavioural and qualitative properties [5]. As solutions for this problem, several works have been proposed to translate these source models into another which supports analysers. For example, in our previous work [11] we proposed to map SCA to Acme/Armani for the verification of the structural and qualitative properties of an SCA software architecture. For the verification of the WS-BPEL behavioral specifications, numerous works are presented to translate the WS-BPEL activities into a formal technique. For example, the works presented in [23], [28] and [26] prose, respectively, to translate the WS-BPEL activities into LOTOS, FSP and PetriNet.

In this work, we target a formal verification of the behavioral properties of SCDL/WS-BPEL service-component architectures. To achieve this, the model transformation approach is used to translate an SCDL/WS-BPEL service-component architecture to an Ada concurrent program [25]. In this study, the Wright formal ADL [2] is used as an intermediate modeling language. The choice of these two languages in our verification approach is justified mainly by the following three factors:

- The Wright ADL defines eleven standard properties related to the consistency of software architecture among which four -assimilated to behavioral contracts- are automated by our Wr2fdr tool [13, 20]. The latter contracts can be checked with the FDR2 model-checker [24].
- The semantic rapprochement between the Wright process and the Ada task favors the formalization of the Wright configuration by an Ada concurrent program.
- The presence of different analysis tools related to the detection of the dynamic and specific behavioral problems of an Ada program. For example, SPIN, SMV, FLAVERS and INCA [7, 8] are four formal analysis tools of Ada concurrent programs and are complementary.

The paper is structured as follows: Section 2 proposes an overview of our SCDL/WS-BPEL behavioral verification; Section 3 deals with our systematic rules allowing the translation of SCDL/WS-BPEL source software architecture to the Wright target software architecture; Section 4 exhibits the translation rules of the Wright abstract architecture into the Ada concurrent program. An overview of the main related works is presented in Section 5; Finally, Section 6 provides a conclusion and possible future work.

2. PROPOSED APPROACH

Our verification approach presented in Figure 1 can be considered as four consecutive steps. Firstly, we propose to use the SCDL language to describe the structural aspect of service-component architecture, and the WS-BPEL language to describe the behavioral aspect of this architecture. Secondly, a set of rules allowing the translation of an SCDL/WS-BPEL source architecture to Wright description is proposed. Then, the Wright description obtained can be automatically translated into a CSP specification through our Wr2fdr tool. Using the FDR2 model checker, this CSP specification can be checked by the standard properties proposed by the Wright ADL. Finally, a latest transformation of the Wright specification into an Ada concurrent program can be proposed. This is favored by the verification of the dynamic and specific behavioral properties of the SCDL/WS-BPEL source software architecture. Indeed, many are the tools that allow the analysis of the dynamic and specific properties of an Ada concurrent program.

For example, SPIN, SMV, FLAVERS and INCA [8] are four formal analysis tools of Ada concurrent programs and are complementary.

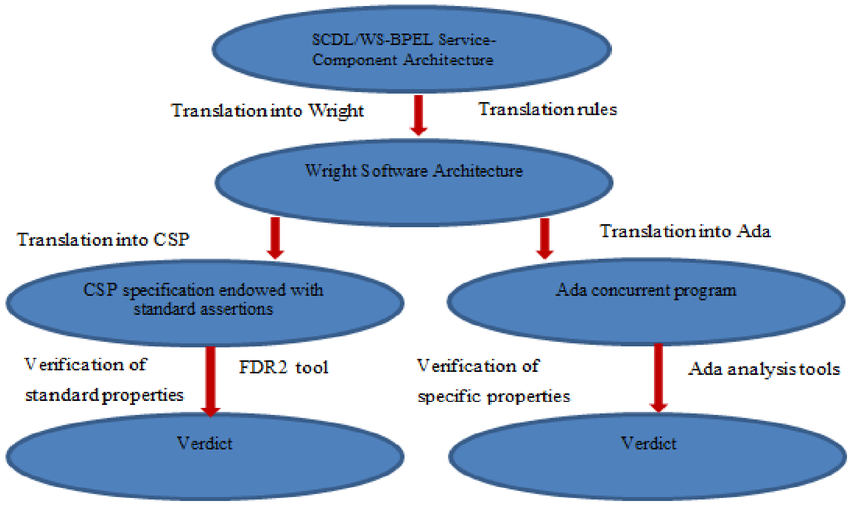

SCDL/WS-BPEL Service-Component Architecture
Translation into Wright
Translation rules
Wright Software Architecture
Translation into CSP
Translation into Ada
CSP specification endowed with standard assertions
Ada concurrent program
Verification of standard properties
FDR2 tool
Verification of specific properties
Ada analysis tools
Verdict
Verdict

Figure1.Verification approach

3. TRANSLATION OF SCDL/WS-BPEL TO WRIGHT

This section is subdivided on two subsections: In Sub-section 3.1, we propose a set of rules allowing the translation of an SCDL/WS-BPEL architecture to a Wright specification. This allows the verification of standard behavioral properties supported by the Wr2fdr tool accompanying the Wright ADL. Then, a validation of this proposed approach will be proposed in subsection 3.2.

3.1. Translation Rules

Regarding the static aspect, An SCA software architecture is generally described in an XML SCDL file. The latter expresses all the relations in a composite. In this language, the basic deployment markup of a Web application is the composite. An SCDL composite is an assembly of heterogeneous components. Each SCDL component is based on a common set of abstractions such as services, references and properties. In the context, services and references describe, respectively, what a component provides and what a component requires from its external environment. These services and references can be matched with bindings. Hence, each SCDL markup can be specified in Wright as follows:

- An SCDL composite can be translated to a Wright configuration;
- An SCDL component can be translated to a Wright component;
- An SCDL component's reference can be translated to a Wright port with the same name;
- An SCDL component's service can be translated to a Wright port with the same name;
- An SCDL wire connects two SCA components. Hence, we propose to translate an SCDL wire to a Wright connector that proposes two roles.

Concerning the translation of the WS-BPEL behavioral descriptions, we propose to translate each WS-BPEL process by a CSP process. In this translation, each primitive activity is translated to a CSP event. Since WS-BPEL provides three kinds of activities, we suggest translating each activity by a specific event as follows:

- An <invoke> activity is used to initialize an appeal of an operation Oper. This activity can be modeled in CSP by an initialized event as follows : _invokeOper
- A <receive> activity is used to wait for a message from an external operation Oper. This observed activity can be modeled in CSP by an observed event as follows: receiveOper
- A <reply> activity is used to initialize a response to an external operation Oper. This activity can be modeled in CSP by an initialized event as follows: _replyOper

In addition to these communication primitive activities, WS-BPEL provides typical structured activities such as sequence, flow, terminate, if, switch, while, repeatUntil etc. These control structures can express a causal relationship between multiple invocations by means of control and data flow links. For the WS-BPEL control structures, we propose the following translation rules:

- The <sequence> construct is used in WS-BPEL wherever a series of activities needs to occur sequentially, although they may be contained one or more times within looping or concurrent construct activities. This <sequence> construct can be modeled in CSP by a set of events separated by the prefixing operator (->).
- Concurrency in WS-BPEL permits us to model the concurrent transitions in the message sequence charts. In WS-BPEL, this is specified using the <folw> construct. However, the concurrency in CSP is modeled by the parallel composition operator (| |). This operator is formally given as: If P and Q are processes then (P | | Q) represents the concurrent execution of P and Q. Hence, using the CSP parallel operator (| |), we can model the WS-BPEL flow activities by a set of concurrent processes.
- In WS-BPEL, the conditional branching introduces decision points to control the execution flow of a process. Each conditional structure such as <if> or <switch> can be modeled in CSP by the adequate choice operator:
  - ([ ]) deterministic choice operator: if the choice between these activities is an external choice. In other words, if these activities are observed (receive activity).
  - (|~|) nondeterministic choice operator: if the choice between these activities is an internal choice. In other words, if these activities are initialized (invoke or reply activity).
- In WS-BPEL, as in most programming languages, loops are used to repeat activities. Each looping structure such as <forEach>, <while> or <repeatUntil> can be modeled in CSP by a recurrent process as follows: P=… -> P.

3.2.Validation

The objective of this section is to validate our SCDL/BPEL transformation approach to Wright / CSP on an online Banking system. This system offers its customers remote banking services. To put it simply, we have retained the following services: visualization of balances, transfer of funds and loan request. The proposed modeling includes the following two components (see Figure 2):

- The AccountBank component provides a service for client authentication and a reference that triggers the three services offered above.

- The DataBase component provides a service that executes requests from the AccountBank component.

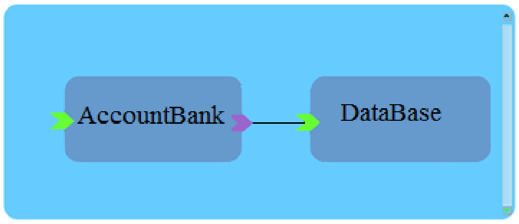


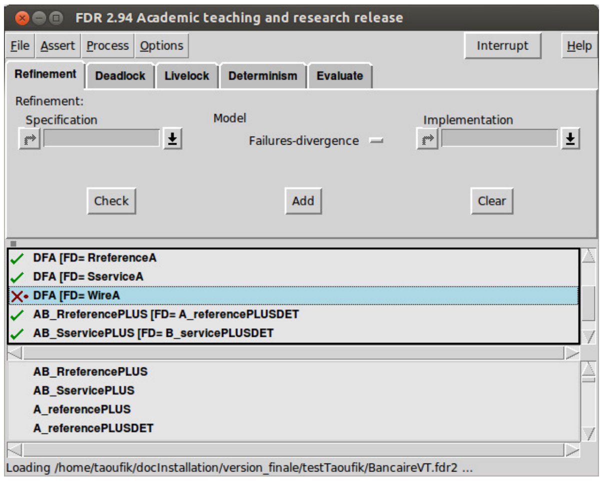
Figure2. Graphical description of the Banking system

Our SourceForge repository [15] presents the complete description (SCDL and BPEL files) of this Banking system. Listing 1 shows the Wright configuration corresponding to the translation of the SCDL / BPEL component assembly of this Banking system.

```
Configuration BankingSystem
Component AccountBank
  Port client = receiveAuthenticate -> _replyAuthenticate -> TICK
  Port reference = TICK |~| (_invokeVisualisation -> MovementsTicket -> TICK |~|
_invokePret -> PretTicket -> TICK |~| _invokeTransfert-> TransfertTicket ->
(TICK |~| _invokeVisualisation -> MovementsTicket -> TICK))
  Computation = client.receiveAuthenticate -> _client.replyAuthenticate ->
(TICK |~| (_reference.invokeVisualisation -> reference.MovementsTicket -> TICK
|~|_reference.invokePret-> reference.PretTicket -> TICK |~|
_reference.invokeTransfert-> reference.TransfertTicket -> (TICK |~|
_reference.invokeVisualisation -> reference.MovementsTicket-> TICK)))

Component DataBase
  Port service = receiveVisualisation -> _replyVisualisation -> TICK []
receiveTransfert->_replyTransfert -> TICK [] receivePret -> _replyPret -> TICK
  Computation = service.receiveVisualisation -> _service.replyVisualisation ->
TICK [] service.receiveTransfert -> _service.replyTransfert -> TICK []
service.receivePret -> _service.replyPret-> TICK

Connector Wire
  Role Rreference = TICK |~|(_invokeVisualisation -> MovementsTicket -> TICK |~|
_invokePret -> PretTicket -> TICK |~| _invokeTransfert-> TransfertTicket ->
(TICK |~| _invokeVisualisation -> MovementsTicket -> TICK) )
  Role Sservice = receiveVisualisation -> _replyVisualisation -> TICK []
receiveTransfert->_replyTransfert -> TICK [] receivePret -> _replyPret -> TICK
  Glue= Rreference.invokeVisualisation -> _Sservice.receiveVisualisation ->
Sservice.replyVisualisation -> _Rreference.MovementsTicket -> TICK []
Rreference.invokeTransfert -> _Sservice.receiveTransfert ->
Sservice.replyTransfert -> _Rreference.TransfertTicket -> TICK []
Rreference.invokePret -> _Sservice.receivePret -> Sservice.replyPret ->
_Rreference.PretTicket -> TICK

Instances
  A : AccountBank
  B : DataBase
  AB :Wire
Attachments
  A.reference As AB.Rreference
  B.service As AB.Sservice
End Configuration
```

Listing 1. Wright configuration associated to the Banking system.

Using our Wr2fdr tool [20] and the FDR2 model-checker, we have formally verified the consistency of two components (AccountBank and DataBase) and the non-blocking of the Wire connector (see Figure 3). However we detected an architectural error related to the non-compatibility between these two components via the Wire connector.



```
Failures-divergence

✓ DFA [FD= RreferenceA
✓ DFA [FD= SserviceA
✗ DFA [FD= WireA
✓ AB_RreferencePLUS [FD= A_referencePLUSDET
✓ AB_SservicePLUS [FD= B_servicePLUSDET

AB_RreferencePLUS
AB_SservicePLUS
A_referencePLUS
A_referencePLUSDET

Loading /home/taoufik/docInstallation/version_finale/testTaoufik/BancaireVT.fdr2 ...
```

Figure 3. Verification with FDR2 model-checker

4. TRANSLATION OF WRIGHT TO ADA

The means to establish automatic connections between Wright and Ada are limited. For example, Naumovich et al. [18] offer a manual translation of Wright into Ada without explanation rules. In our previous work [6] we established a set of simple rules allowing translating Wright software architecture into Ada. In this study, a significant improvement of our translation rules was proposed. The main structural concepts treated in this paper are: configuration, component, connector, port, role, computation, glue, attachments, process, initialized event, observed event, successfully terminated event, prefixing operator, deterministic choice operator and nondeterministic choice operator. To achieve this, we proposed an Ada package called ArchWright allowing the representation in Ada of the main structural concepts coming from the Wright ADL. For this purpose, we benefited from using the Ada composite type "record" that groups one or more fields. A field can be of any type, even a record. In addition, the typing possibilities offered by the Ada language are profitably used for the translation of the Wright behavioral aspects.

4.1. Component Translation

A Wright component is a computation element with multiple ports. Each port represents a point of interaction between the component and its environment. A CSP process expresses the component properties and expectations seen through this port and is part of its behavior. The computation specification provides a complete description of the behavior and properties of the component showing how ports are grouped and used.

In this work, we propose to implement a Wright component with an Ada record compound with two fields:

- Ports: it represents the component's ports. It is modeled by an array of CSPTask, where the CSPTask is the task type proposed to implement in Ada the CSP process (see Section 3.4).

- Computation: it represents the component computation. It can be modeled by a single CSPTask.

Listing 2 illustrates the formalization of the Wright component by an Ada record type. This new type represents a base type for all Wright component instances.

```
type CompWright (portsNumber : natural) is record
  Ports: array (portsNumber) of CSPTask;
  Computaion: CSPTask;
end record;
```

Listing 2. Formalization of Wright component in Ada

Table 1 illustrates an example of an Ada implementation of a Wright component instance which proposes three ports: Input, Left and Right.

Table 1. Translation of Wright component instance in Ada

Wright Component	Implementation in Ada

`Component Filter`
`   Port Input=…`
`   Port Left =…`
`   Port Right=…`
`   Computation=…`
`Instances`
`   SplitFilter : Filter`
	`SplitFilter : CompWright (3);`
	`       -- it have three ports`
	`SplitFilter := CompWright'(`
	` Ports =>(1 => …,  --CSPTask of first port`
	`         2 => …,  --CSPTask of second port`
	`         3 => …); --CSPTask of third port`
	` Computation => …;--CSPTask of computation`
	`);`

4.2. Connector Translation

In Wright, connectors define patterns of interaction between components. A Wright description of a connector consists of a set of connector roles and connector glue. Each role specifies, by a CSP process, the behavior of a participant in the interaction. However, the connector's glue describes how the participants work together to create an interaction.

In this work, we propose to formalize a Wright connector with an Ada record compound with two fields:

- Roles: it represents the connector roles. It is modeled by an array of CSPTask, where the CSPTasck is the task type proposed to implement the CSP process (See Section 3.4).

- Glue: it represents the glue of the connector. It can be modeled by a single CSPTask.

Listing 3 illustrates the formalization of the Wright connector by an Ada record type. This new type represents a base type for all Wright connector instances.

```
type ConnectorWright (rolesNumber : natural) is record
  Roles: array (rolesNumber) of CSPTask;
  Glue: CSPTask;
end record;
```

Listing 3. Formalization of Wright connector in Ada

Table 2 shows an example of an Ada implementation of a Wright connector instance called Pipe that proposes two roles: Source and Sink.

Table 2. Translation of Wright connector instance in Ada

Wright Connector	Implementation in Ada

`Connector Channel`
`    Role Source=…`
`    Role Sink=…`
`     Glue= …`
 	
`Instances`
`Pipe : Channel`
	`Pipe : ConnectorWright (2);`
	`          -- it have 2 roles`
	`Pipe := ConnectorWright'(`
	`Roles =>`
	`   (1=>…,      --CSPTask of the Source role`
	`    2 => …);   --CSPTask of the Sink role`
	`Glue=>…    ; --CSPTask of the glue`
	`);`

4.3. Configuration Translation

A Wright configuration can be translated to an Ada concurrent program using the ArchWright package. Table 3 illustrates the principle of the translation of a Wright configuration to an Ada concurrent program. For traceability reasons, we keep the same identifiers used in the Wright specification.

Table 3. Translation of Wright configuration in Ada

Wright Connector	Implementation in Ada

`Configuration FilterPipe`
 	
`Component Filter`
`   Port Input=…`
`   Port Output  =…`
`   Computation=…`
 	
`Connector Pipe`
`    Role Source=…`
`    Role Sink=…`
`   Glue= …`
 	
`Instances`
`Filter1, Filter2 : Filtre`
`Pipe1 : Pipe`
 	
`Attachments`
`…`
`…`
`End Configuration`
	`with ArchWright;`
	`use ArchWright;`
	`procedure FilterPipe is`
	` Filter1 :CompWright(2);-- it have two ports`
	` Filter2 :CompWright(2);-- it have two ports`
	` Pipe1 :ConnectorWright(2);--it have 2 roles`
	` Filter1 := CompWright'(`
	`  Ports =>(1=> … , --CSPTask of first port`
	`          2=> … );--CSPTask of second port`
	`Computation =>…;--CSPTask of computation`
	` );`
	` Filtre2:= Filtre1;`
	`-- Filtre1 and Filtre2 are derived`
	`-- from the same component`
	
	` Pipe1 :ConnectorWright(2);`
	`   -- Pipe1 have 2 roles`
	` Pipe1 := ConnectorWright'(`
	`  Roles => (`
	`     1=>…,--CSPTask of the Source role`
	`     2=>…); --CSPTask of the Sink role`
	` Glue=>…; --CSPTask of the glue`
	` );`
	`begin`
	`…`
	`…`
	`end FilterPipe ;`

4.4. CSP Process Translation

The Wright ADL is one of the first approaches allowing the description of the behavioral aspect of architectural elements. Indeed, the behavior of a Wright component (respectively of a connector) is described locally through the ports (respectively roles) and, generally, through a computation (glue respectively) using a CSP process algebra. In this algebra, a simple process can be a compound with a set of observed and initialized events separate with the CSP prefixing operator (□).

Based on the similarity between the CSP process and the Ada task, we offer an intuitive correspondence provided below for translating a CSP process to an Ada task:

- A CSP process leads to an Ada task;
- A CSP event naturally corresponds to an Ada entry. In order to differentiate between an observed and an initialized event, we propose to use the same prefixed notation used in CSP: an observed event is denoted by (e) and an initialized event is denoted by (_e)
- The recursion operator can be translated by an Ada loop;
- The CSP prefixing operator (□) can be specified by the Ada sequential instruction;
- The CSP successfully terminated event (denoted by TICK or §) can be implemented with the Ada "terminate" instruction.
- The CSP nondeterministic (or internal) choice operator (denoted by Π or | ~|) allows the future evolution of a process to be defined as a choice between two sub-processes, but does not allow the environment any control over which one of the component processes will be selected. This nondeterministic choice can be implemented in Ada with a simple conditional structure (if). If we have multiple composite processes, the Ada conditional structure (case) can be used.
- The CSP deterministic (or external) choice operator ([ ]) allows the future evolution of a process to be defined as a choice between two sub processes, and allows the environment to resolve the choice by communicating an initial event for one of the processes. This deterministic choice can be implemented with the Ada "select" instruction.

Table 4 illustrates these proposed rules allowing the translation of a CSP specification to Ada.

Table.4. Translation of CSP specification to Ada

CSP Specification	**Implementation in Ada**

`P = _request ->  result -> §`	`task P is`
	` entry _request;`
	` entry  result;`
	`end P ;`
	`task body P is`
	` accept _request;  accept  result;`
	`end P;`

`P1 |~|P2`	`if internCondition then   P1`
	`else     P2`
	`end if;`

P1 |~| P2 |~| P3	**case** internCondition **is** **when** 1 => P1 ; **when** 2 => P2 ; **when** 3 => P3 ; **end case** ;

P1 [ ] P2 [ ] TICK	**select** P1;
	or P2;
	or terminate ;
	end select;

4.5. Attachments Translation

After the instances declaration, the attachments statement completes the Wright configuration. These attachments show which components participate in which interaction, binding a component's port with a connector's role.

Each association between a component's port and a connector's role can be implemented in Ada by a sequence of entries call of the events specified in the interconnected port/role. The general form of the entries call (event) is specified as follows:

- If the port entry corresponds to an initialized event: we call this entry, then we call the similar entry of role (Component.port._event; Connector.role._event;).
- Else, if this entry corresponds to an observed event, we call this entry after the call of the similar entry in the role (Connector.role.event; Component.port.event;).

Table 5 shows an illustrative example of an attachment translation.

Table 5. Translation of Wright attachment to Ada

Wright Configuration	**Ada implementation**

Configuration ClientServer
…
Component Client
Port caller =_request →
result → §
…
Connector cs
Role client =_send →
receive → §
Role server = …
…
Instances
c : Client
cls: cs
 	
Attachments
Client. caller **As** cls.client
…
End Configuration
	with ArchWright;
	use ArchWright;
	procedure ClientServer **is**
	-- CSP Task of the caller port
	task Client_caller **is**
	entry _request;
	entry result;
	end Client_caller ;
	task body Client_caller **is**
	accept _request; **accept** result;
	end Client_caller ;
	-- CSP Task of the client role
	task Cls_client **is**
	entry _send;
	entry receive;
	end Cls_client;
	task body Cls_client **is**
	accept _send;**accept** receive;
	end Cls_client;
	-- CSP Task of the server role
	begin -- principal program
	Client_caller._request;
	Cls.client._send;
	Cls.client.receive;
	Client_caller.result ;
	end ClientServer ;

4.6. Validation

Using the translation rules presented in the previous sections on the Banking system (See Section 3.2), we obtained an Ada concurrent program available at [15]. Using an Ada analysis tool such as SPIN, SMV, FLAVERS and INCA [8], we can detect several errors in our Banking system. In particular, using the FLAVERS toolset we detect that our source AccountBank component has a deadlock problem between its global behavior and its interface behaviors. Indeed, its global behavior allows the call of two successive rendezvous of the AccountBank._invokeVisualisation entry; however, the behavior of this reference interface cannot accept these rendezvous. To cover this error, we can replace the global BPEL of the AccountBank component by another compatible with its interfaces.

5. Related Works

The approach shared by most of the existing works in the field of Web services architectures consistency verification is the use of techniques and general tools such as B [1] and CSP [14]. To achieve this, numerous works offer more or less systematic translations of source architecture to the target model. In this section, only the works related to the behavioral verification of the SCDL/WS-BPEL architectures are mentioned.

Yeung proposes in [28] to translate the WS-BPEL web service to CSP to verify the behavior properties of web services architecture. In this paper, formal verification can be carried out based on the notion of CSP trace-refinement and can take advantage of the FDR2 model checking. Another formal approach [9] similar to the one reported by Yeung [28] proposes to use the FSP formal language to check if a web service composition implemented inWS-BPEL satisfies a web service composition specification captured by Message Sequence Charts (MSCs). Both theWS-BPEL process and the MSC are translated to FSPs. Each FSP represents a finite labelled transition system. Using the LTSA model checking tool, this FSP target specification can check the safety and progress properties as well as properties expressed in the LTL logic. In [10], Foster et al. use this LTSA to check the compatibility of web service compositions in WS-BPEL. Since the semantics of Petri Nets is formally defined by mapping each WS-BPEL process to a Petri net, a formal model of WS-BPEL can be obtained. This approach has been followed in several works. For example, in [26] Verbeek et al. formalize some WS-BPEL activities used for the orchestration of Web services as a class of Petri Net called workflow nets. For this class of Petri nets, a verification tool named Wolfan has been developed. This tool can verify properties such as the termination of a workflow net and detection of nodes that can never be activated. The authors of [27] propose to map most of the basic and structured activities of WS-BPEL and the Web Service Choreography Interface (WSCI) to Coloured Petri Nets (CPN). Choosing this CPN as a verification method relates to the work of Tsai and Xu [22]: CPN, as the intermediate formal basis, can be transformed to the input languages of existing analysis tools such as SPIN, SMV, SMC, and IOTA. In [12] Hamel et Al. propose to use the Event-B method to check the structural and behavioural properties of an SCA component assembly. To achieve this, the B-Invariant and B-Event are profitably used to formalize the patterns proposed by Barros [4]. Using the ProB animator, Hamel et Al. validate their formal approach on an event-B specification.

6. Conclusion

In this paper, an approach for verifying the behavioral coherence of SCDL/WS-BPEL component-service architectures has been proposed. To achieve this, it has been proposed to map SCDL/WS-BPEL to the Wright ADL, thus allowing the checking of the standard properties supported by this ADL. Indeed, the Wright ADL defines eleven standard properties related to the consistency of software architecture, among which four, assimilated to behavioral contracts, are automated by the Wr2fdr tool. The latter contracts can be checked with the FDR2 model checker. As a second step of our verification approach, the translation of the Wright specification to an Ada concurrent program has been put forward, which allows:

- The validation of an abstract software architecture using dynamic analysis tools associated with Ada, such as test data generation and debugging. Indeed, the execution of abstract software architecture with representative test data allows the validation of the future system in an early time.

- The verification of specific properties on an abstract software architecture using several verification tools supporting the Ada concurrent language. Indeed, many are the tools that allow the analysis of Ada concurrent programs. For example, SPIN, SMV, FLAVERS and INCA [8] are four complementary approaches permitting the static analysis of Ada concurrent programs. Indeed, vis-à-vis the specification of properties to be verified, the SMV and SPIN tools promote the property-oriented state, while INCA and FLAVERS promote the property- oriented trace.

Our verification approach is validated on a set of uses cases available at our SourceForge repository [15]. Currently, we are extending this work by an automation of these translation rules using the ATL [16] model transformation language.

REFERENCES

[1] Abrial J. R.: The B-Book, Assigning Programs to Meanings. Cambridge University Press. (1996).

[2] Allen R.: A Formal Approach to Software Architecture. Phd Thesis, Carnegie Mellon University, (1997).

[3] Andrews T., Curbera F., Dholakia H., Goland Y., Klein J., Leymann F., Liu K., Roller D., Smith D., Thatte S., Trickovic I., and Weerawarana S.: Business Process Execution Language for Web Services, (2005).

[4] Barros O.: Business Process Patterns and Frameworks: Reusing Knowledge in Process Innovation. Business Process Management Journal, (2007). Vol.13, N°1, pp. 47- 69.

[5] Beugnard A., Jézéquel J. M., Plouzeau N. and Watkins D.: Making components contract aware. In IEEE Software, (July 1999), pp 38-45.

[6] Bhiri M. T., Fourati F., Kmimech M., Graiet M.. Transformation exogène de Wright vers Ada. Technique et Science Informatiques, Vol. 31, No. 7, (2012), pp.839-868

[7] Cobleigh J. M., Clarke L. A. and Osterweil L. J.: FLAVERS: a Finite State Verification Technique for Software Systems'. IBM Systems Journal, (2002), vol. 41, n° 1, pp. 140-165.

[8] Dwyer M. B., Pasarean S. C. and Corbett J. C.: Translating ADA programs for Model checking: A tutorial', Technical Report 12, Kansas State University. (1998).

[9] Foster H., Uchitel S. Magee J. and Kramer J.: Model-based verification of web service compositions. In Proceedings of 18th IEEE International Conference on Automated Software Engineering, pp. 152-163, Montreal, Canada, (October 2003). IEEE.

[10] Foster H., Uchitel S. Magee J. and Kramer J and Magee J.: Compatibility verification for web service choreography. In Proceedings of the IEEE International Conference on Web Services, pp. 738-741, San Diego, CA, USA, (June 2004). IEEE.

[11] Haddad I., Kmimech M., Sakka Rouis T., and Bhiri M. T.: Towards a practical approach to check service component architecture. In 11th International Conference on Semantics, Knowledge and Grid, (2015), pp. 65–72. IEEE.

[12] Hamel L., M. Graiet G., M. Kmimech M.: Formal modeling for verifying SCA composion. RCIS conference, (2015), pp. 193-204.

[13] Hammami M., Bhiri M. T., Kmimech M. and Graiet M.. : Traduction de Wright en CSP pour la vérification par le model-checking. 6 ème conférence francophone sur les architectures logicielles, Montpellier. (2012). [14]Hoare C. A. R.: Communicating Sequential Processes. Prentice Hall (1985).

[15] https://sourceforge.net/projects/SCA2WrightToAda

[16] Jouault F., Allilaire F., Bézivin J. and Kurtev I.: Atl: A model transformation tool. Sci. Comput. Program. (2008) vol. 72, n° 1-2, pp. 31-39.

[17] Leymann F.: Web services ow language (WSFL 1.0), May 2001.

[18] Naumovich G., Avrunin G. S., Clarke L. A. and Osterweil L. J., Applying static analysis to software architectures'. ACM SIGSOFT 1997, Software Engineering Notes, vol. 22, n° 6, (1997), pp. 77-93.

[19] OASIS. Service Component Architecture Assembly Model Specification Version 1.1. Oasis. https://www.oasis-open.org/standards, (July 2017).

[20] Sakka Rouis T., Bhiri M. T., Kmimech M and Moussa F.: Wr2Fdr Tool Maintenance for models Checking. Proceeding in 16th International Conference on Intelligent Software Methodologies, Tools, and Techniques (2017), Kitakyushu, Japan.

[21] Thatte S.: XLANG: Web Services for Business Process Desig. Corporation, 2001.

[22] Tsai J. J. P. and Xu K.: An empirical evaluation of deadlock detection in software architecture specifications, Annals of Software Engineering, 7, (1999). pp. 95-126.

[23] Tremblay G. and Chae J.: Towards specifying contracts and protocols for web services. In H. Mili and F. Khendek, editors, Proceedings of the MCeTech Montreal Conference on eTechnologies, Montreal, Canada, (January 2005). pp. 73-85.

[24] The Formal Systems Website, vendor for the FDR2. http://www.fsel.com/

[25] Taft S. T., Duff R. A., Brukardt R. L., Ploedereder E. and Leroy P.: Ada 2005 Reference Manual. (2005)

[26] Verbeek H. M. W. and. Van Der Aalst W. M. P.: Analyzing BPEL processes using Petri nets. In D. Marinescu, editor, Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination, Workow and Business Process Management, pp. 59-78, Miami, FL, USA, (October 2005).

[27] Yang Y., Tan Q., Yu J., and Liu F.: Transformation BPEL to CP-nets for verifying web services composition. In Proceedings of the International Conference on Next Generation Web Services Practices, Seoul, Korea, (August 2005). IEEE.

[28] Yeung W. L.: Mapping ws-cdl and BPEL into csp for behavioural specification and verification of web services. ECOWS, IEEE Computer, (2006), pp. 297-305.

AUTHORS

Taoufik Sakka Rouis is a doctor in computer science and member at the LIPAH laboratory (Tunisia). He received his Ph.D’s degree in Computer Science from the University of Manouba. His main research interests are component-based software engineering and formal verification of service-component architectures.

Moahamed Tahar Bhiri is a professor at the Faculty of Sciences of Sfax (Tunisia). He led numerous studies on the subject of Object, Component and transformation models. He uses the formal models such as Wright, Acme, B, Event-B and CSP to check the consistancy of the UML2.0 diagrams. Currently, he is interested in the integration of competition in the Event-B method.

Mourad Kmimech is an associate professor at the High School of Computer Science and Mathematics, Monastir University. He holds a Ph.D. (2010) in software engineering from the University of Pau and Pays of the Adour, France. His research interests include service computing, service component architecture (SCA), services reliability/flexibility and formal verification.