Rigorous analysis of service composability by embedding WS-BPEL into the BIP component framework

Emmanouela Stachtiari, Anakreon Mentis, Panagiotis Katsaros
Department of Informatics
Aristotle University of Thessaloniki
Thessaloniki, Greece
$\{$*emmastac,anakreon,katsaros*$\}$*@csd.auth.gr*

***Abstract*—Behavioral correctness of service compositions refers to the absence of service interaction flaws, so that essential service properties like deadlock freedom are preserved and correctness properties related to safety and liveness are assured. Model checking is a widespread technique and it is based on extracting an abstract model representation of the program defining a service orchestration or choreography. During model extraction, the original structure of the service composition cannot be preserved and backwards traceability of the verification findings is not possible. We propose a rigorous analysis within the BIP component framework. Being rigorous means that the analyst is able to reason on which properties hold and why. The BIP language offers a sound execution semantics for a minimal set of primitives and constructs for modeling and composing layered components. We formally define the WS-BPEL 2.0 execution semantics and we provide a structure-preserving translation (embedding) of WS-BPEL to BIP. Structure preservation is feasible, due to the formally grounded expressiveness properties of BIP. As a proof of concept, we apply the developed embedding to a sample BPEL program and present the analysis results for a safety property. By exploiting the BIP model structure we interpret the analysis findings in terms of the service interactions stated in the BPEL source code. A significant benefit of BIP is that it applies compositional reasoning on the model structure to guarantee essential correctness properties and avoid, as much as possible, the scalability limitations of conventional model checking.**

***Keywords*-service composition; WS-BPEL; formal analysis**

I. Introduction

One of the key promises of Service Oriented Computing (SOC) is the delivery of services as composable software units amenable to integration in a multitude of applications and composite services. However, building composite applications is still a challenge if we consider the various functional and non-functional correctness properties that services should offer and the compatibility issues that arise.

Most efforts to address this problem focus on the compatibility of service descriptions and their semantic interoperability [2]. Although compatibility of services at this level is necessary, it is not sufficient for combining them in a correct composition. Adequate technical approaches are needed to reveal behavioral incompatibilities, i.e. flaws in service interactions that cause unexpected service behavior. Correctness depends on several concerns, including the choice between synchronous and asynchronous interaction, atomicity, concurrency and so on. Languages for service orchestrations and service choreographies provide high-level primitives and constructs to efficiently define complex service interactions. However, they lack support for guaranteeing behavioral correctness, which is related to safety and liveness goals for the composite service.

Numerous works try to address this problem with verification by model checking [4]. These approaches work on an abstract model representation extracted from the service composition program. The lack of standardized and formally-defined execution semantics for the used composition languages can lead to debatable analysis results, as a consequence of implicit and possibly divergent assumptions during the model extraction (numerous differences between various WS-BPEL implementations are reported in [12]). Moreover, the structure of the service composition in the program source is lost and backwards traceability of the verification findings is not possible. This happens because formalisms based on automata, petri-nets or process algebras lack a sufficiently expressive set of composition primitives, appropriate for a model representation that preserves the service composition structure.

We propose a rigorous approach for the analysis and design of service compositions based on the BIP (Behavior, Interaction, Priority) component modeling framework [3]. In the core of the BIP framework lies a powerful executable modeling language with formally defined operational semantics and mathematically proven expressiveness properties [6]. In BIP, systems are modeled by superposing three distinct layers. The lower layer (behavior) consists of a set of atomic components representing transition systems. Interactions between components are specified by connectors in the middle layer. Each connector is defined as a relation between ports equipped with synchronization types. The highest layer (priorities) is used to enforce scheduling policies applied to interactions.

Our approach elaborates on recent trends in the field of language technology [11], [10] to introduce a structure-preserving translation, for embedding WS-BPEL 2.0 service compositions into BIP. The embedding is defined as a structural representation of the BPEL primitives and constructs, in terms of the BIP language elements. The translation involves an explicit definition of the BPEL execution semantics in the BIP semantic model. For this purpose, we are based on the WS-BPEL standard [1] and other related works [8], [16], [14]. In effect, two shortcomings of conventional model checking are addressed: structure preservation and explicit composition execution semantics. As a proof of concept for the effectiveness of the proposed technique, we present a sample BPEL program with an expected safety property and we provide the analysis results obtained with the BIP tools.

The rest of the paper is organized as follows. Section II summarizes the WS-BPEL abstract syntax and execution semantics. Section III introduces the BIP language and analysis framework that provides novel verification techniques, which avoid the problem of state space explosion. In section IV, we present the structural representation of BPEL processes in BIP. Section V highlights our structure-preserving embedding of WS-BPEL into BIP and section VI reports the analysis results for a BPEL application. Section VII considers related work and section VIII exposes the lessons learned. The paper concludes with summarizing remarks and future research directions.

II. BPEL ABSTRACT SYNTAX AND EXECUTION SEMANTICS

BPEL processes are stateful, loosely coupled interactions of web services that exchange information via messages. The only restriction imposed on these interactions is that the structure of the exchanged information (e.g message field names and types) must adhere to a provided specification that conforms to the WSDL schema [7]. Figure 1 outlines the syntax of BPEL. A process is defined by the topmost scope, which contains variable declarations (*var*) for storing the process state. Other declarations include partner links (*plink*) for defining service connections, correlation sets (*corrset*) for selecting the message receiver activity and message exchange rules (*msge*) for disambiguation of replies to pending requests.

Incoming events are processed by event handlers (*evhndlr*) declared in a scope. The channel characteristics of the received message, such as the partner link (ID_{pl}) used for transmission, the operation (ID_{op}) responsible for message processing and the correlation set (ID_{cs}) acknowledged by the event handler, determine which of the available handlers receives the message. Due to the asynchronous communication model of web services, there is no guarantee that a message is received on time. Timeout conditions (E_{date}, $E_{duration}$) for waiting a message are defined with *onAlarm* rules and an associated activity is started when the timer fires.

In a dialog between web services, it is often the case that a participant in a conversation either expects to receive a specific message (*receive*) or needs to reply to a previously received one (*reply*). To initiate a conversation, a process uses the *invoke* activity, which defines the message to be sent and optionally the expected reply.

⟨*process*⟩ ::= ⟨*scope*⟩

⟨*scope*⟩ ::= ID 'isolated'? ⟨*decl*⟩* ⟨*evhndlr*⟩* ⟨*fcts*⟩ ⟨*act*⟩

⟨*decl*⟩ ::= ⟨*var*⟩ | ⟨*plink*⟩ | ⟨*corrset*⟩ | ⟨*msge*⟩

⟨*evhndlr*⟩ ::= ('onEvent' ⟨*channel*⟩ ID_{var}
| 'onAlarm' timeExpr) ⟨*scope*⟩

⟨*channel*⟩ ::= ID_{pl} ID_{op} ID_{cs}

⟨*timeExpr*⟩ ::= (E_{date} | $E_{duration}$) $E_{duration}$?

⟨*fcts*⟩ ::= (ID_{fault}? ⟨*act*⟩)+ ⟨*act*⟩ ⟨*act*⟩

⟨*link*⟩ ::= (ID_{link}+ E_{bool})? (ID_{link} E_{bool})*

⟨*links*⟩ ::= ID+

⟨*from*⟩ ::= literal | ID_{pl} uri | ⟨*from_to*⟩

⟨*to*⟩ ::= ID_{pl} | ⟨*from_to*⟩

⟨*from_to*⟩ ::= ID_{var} part? | xpath

⟨*act*⟩ ::= ⟨*bact*⟩ ⟨*link*⟩?

⟨*bact*⟩ ::= (('receive' | 'reply') ⟨*channel*⟩ ID_{var})
| 'invoke' ⟨*channel*⟩ ID_{var} ID_{var}?
| 'assign' (⟨*from*⟩ ⟨*to*⟩)+
| 'wait' (E_{date} | $E_{duration}$)
| 'compensateScope' ID_{scope}+
| ⟨*scope*⟩ | 'empty' | 'exit' | 'compensate'
| 'throw' ID_{fault} | 'rethrow'
| ('sequence' | 'flow' ⟨*links*⟩?) ⟨*act*⟩+
| ('while' | 'repeatUntil') E_{date} ⟨*act*⟩
| 'forEach' E_{int} E_{int} E_{int}? ⟨*scope*⟩
| 'pick' ('onMessage' ⟨*channel*⟩ ⟨*act*⟩)+ ('onAlarm' ⟨*timeExpr*⟩ ⟨*act*⟩)*
| 'if' E_{bool} ⟨*act*⟩ ('elseif' E_{bool} ⟨*act*⟩)* ('else' ⟨*act*⟩)?

Figure 1: Abstract syntax of BPEL

Normal execution of web services is on occasions interrupted by faults. Handlers (*fcts*) for anticipated faults are declared inside a scope and attempt to compensate or otherwise respond to an abnormal situation. Faults are propagated towards the root of the scope hierarchy until an appropriate handler is found. Unhandled faults cause the process to terminate.

Control flow structures common in many programming languages, such as assignment (*assign*), sequential execution of commands (*sequence*), conditional execution (*if, pick*), repetition (*forEach, while, repeatUntil*) and exceptions (*throw, rethrow, compensate, compensateScope*) are expressed in BPEL by corresponding activities (*act*). Other activities include process termination (*exit*), temporary pause of execution (*wait*) and *empty* that does not performs any computation.

Services are executed in parallel by either the *forEach* or the *flow* activity. Identical copies of an activity's scope execute in parallel and optionally with concurrency guarantees, when the attribute *parallel* of the *forEach* activity is set to true. Links in a *flow* activity, serve the purpose of explicit synchronization between the concurrent activities in

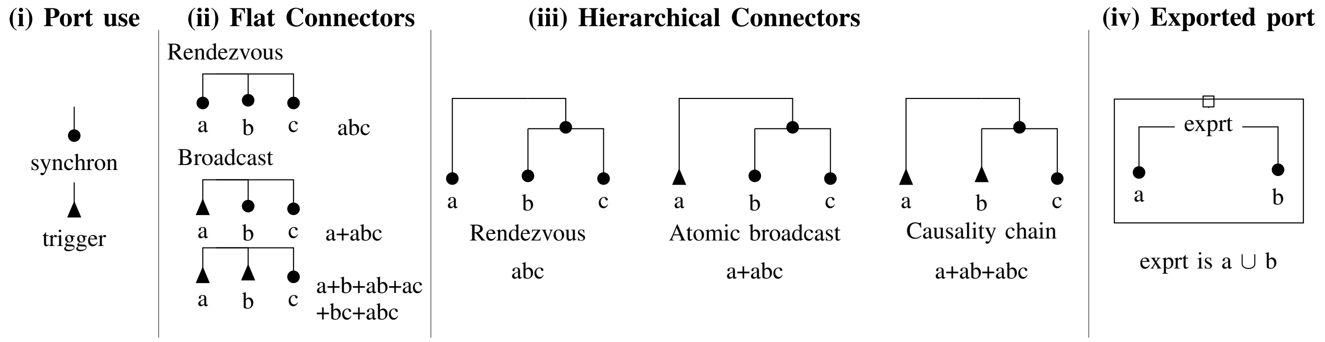


Figure 2: Flat and hierarchical BIP connectors

the *flow*. The execution of an activity with dependency on a link is suspended until the activity it depends on assigns a value to the link.

Activities have access to variables declared in their enclosing scopes. Concurrency guarantees for access on the declared variables and partner links are provided for scopes with *isolated* attribute set to yes.

III. THE BIP LANGUAGE

BIP allows hierarchical construction of *compound* components from *atomic* ones by using *connectors* and *priorities*. Atomic components are transition systems that consist of a set of local variables and a set of transitions. Transitions are defined as a relation over the *control states* that is labeled by *ports*. Ports may be exported for the communication of local data and the synchronization with other components. Transitions are associated with guards. For a transition to be executed, its guard should evaluate to true and some interaction involving its port must be enabled.

Connectors define possibly guarded interactions on the ports of the connected components. An interaction specifies ports with associated transitions that must or may be enabled together, i.e. transitions are executed synchronously. Two possible uses of the connected ports (Figure 2*i*) express strong or weak participation in interactions: *synchron* (must) or *trigger* (may). In connectors that consist of synchrons (rendezvous in Figure 2*ii*) all ports must be enabled. In connectors with one or more triggers, at least one of them should be enabled (e.g. broadcasts in Figure 2*ii*).

Connectors can export their ports for building hierarchies of connectors (Figure 2*iii*). Furthermore, a port may be used to export any subset of a union of ports (Figure 2*iv*). Such a port is enabled if some port in the union is enabled [1]. Connectors can use data variables, in order to compute transfer functions associated with interactions and to update the ports' data. Computations take place iteratively either upwards (*up*) or downwards (*down*) through the connectors' hierarchy levels, but computed values are not stored between the execution of two interactions (connectors are stateless).

Compound components are defined by assembling constituent components (atomic or composite) using connectors. *Priorities* are defined as rules for choosing in a pairwise manner between simultaneously enabled interactions within a BIP component.

A BIP source file includes type definitions for ports and connectors and the description of the system's model component. *Port types* characterize the number and type of data carried by the ports and *connector types* define templates that are parameterized by a list of ports.

BIP has been formally proved [6] that it can encompass directly any coordination mechanism, which means that the coordination mechanisms of the service composition languages can be embedded in BIP by preserving their structure and without any combinatorial explosion in the translation. Finally, state of the art compositional verification techniques [5] are provided that scale linearly with respect to the number of model components, as opposed to the exponential cost of conventional model checking approaches.

IV. STRUCTURAL REPRESENTATION OF BPEL PROCESSES IN BIP

BPEL activities are transformed into BIP components. Atomic components are used for non–decomposable activities (all basic BPEL activities except *assign*) whereas activities with complex structure (e.g *sequence* and *flow*) are modeled by compound components. Moreover, we employ atomic components to model access to shared variables and links used for explicit activity synchronization.

Events that occur in all activity types are shown in Figure 3. Parent activities propagate *start, disable, term* and *interrupt* events to their children, while child activities inform their parent about their current state through the *done, disabled, termed* and *fault* ports. These ports constitute the minimum interface of an activity component. Activity specific behavior is defined by activity specific transitions and ports that replace the arc *[execute]* connecting the *ready* and *done* states. Figure 4 illustrates an assign activity

[1] Exporting a union of ports is a new feature introduced in BIP2.0.

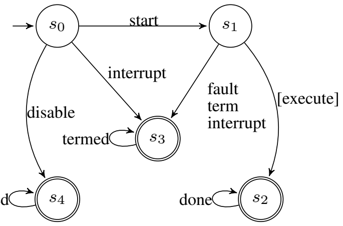


Figure 3: Behavior of non-decomposable activities

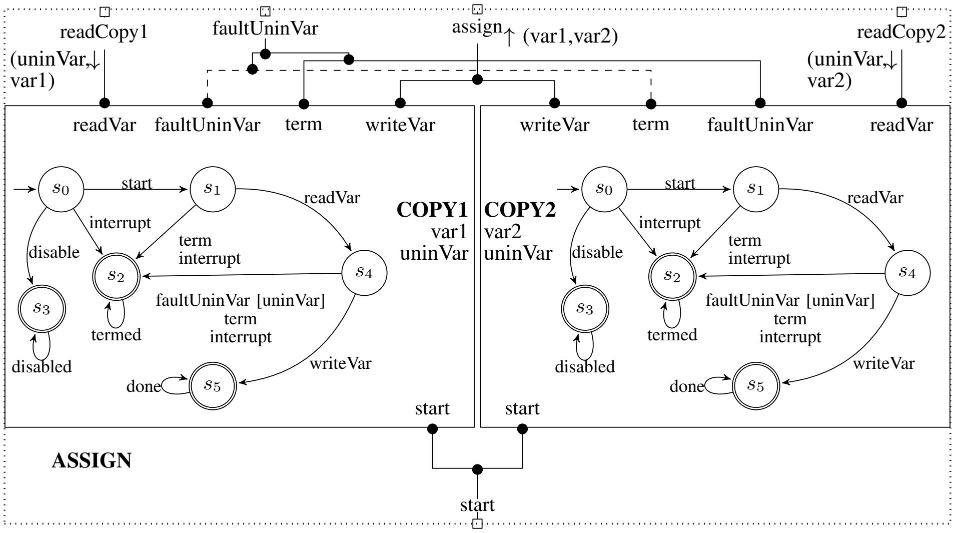


Figure 4: BIP component for an assign activity

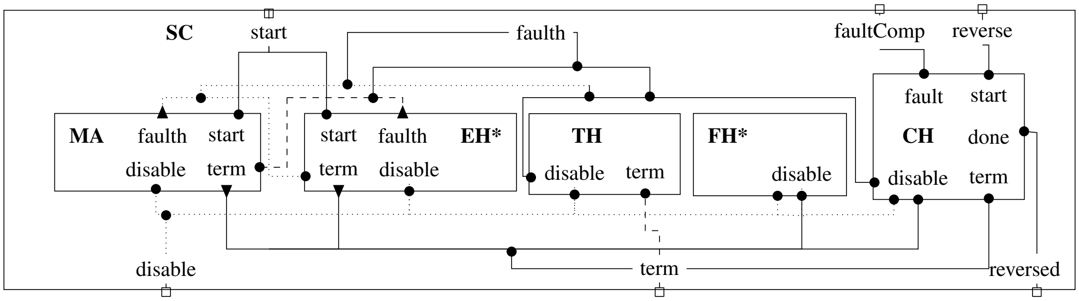


Figure 5: Schematic representation of connectors in a *scope* component.

component that encloses two copy components that copy messages with one part. The copy components concurrently read the *from* part through the readVar port and synchronize on the assignment of the *to* part through the writeVar port. The readCopy1 and readCopy2 ports carry the incoming data and the variable uninVar which signals if some data was not initialized. The assign port carries the outgoing data of the assign component. A fault is thrown by a copy component if the value of uninVar is true, an event that terminates the execution of the assign component. The connectors of the rest of the minimum interface ports are omitted in Figure 4.

A. *Scope*

A pictorial representation of the scope component (SC) is shown in Figure 5. We use different line styles to unambiguously indicate connected ports. A scope is modeled by a compound component that contains (i) the activity of the scope (MA) (ii) a component modeling event handlers (EH), that reacts to received messages and timer events (iii) a termination handler (TH) which is invoked when a fault in some other activity is raised causing the termination of the scope (iv) a component modeling fault handlers (FH) which executes an activity in response to a fault raised in the scope and (v) a compensation handler contained in the (CH) which is invoked when a fault in some other activity is raised, *if* the scope is completed . In addition, a scope contains a data access manager (DAM) component which models access to variables, partner links and correlation sets declared in the scope. The constructs for TH, FH and CH are expressed in the *fcts* rule of BPEL's abstract grammar. For isolated scopes, the DAM component models concurrent read and write access to the variables and the partner links it manages.

Normal execution begins with an invocation of the *start* port that subsequently invokes the *start* port of the MA and EH components. If a fault occurs in the BPEL process, which has been propagated to an enclosing scope, either all constituent components are disabled (*disable* port) or the *term* port of the scope is invoked. Upon the invocation of the *term* port, either (i) the *term* port of MA or EH is invoked triggering the *disable* port of the FH and CH components ,since exception handling is no longer needed (ii) the *term* port of the TH component is invoked to terminate the execution of its contained activity, or (iii) the *term* port of the CH component is invoked. The *start* port of the TH component is triggered when all constituent components of the scope have enabled the *done* or *termed* port signaling they have ended.

The *done* port signals that the MA and EH components have completed their activity without faults or an occurred fault has been handled by the FH component without throwing another fault. The *reverse* port of the scope is enabled if no faults have occurred and the *done* port is enabled. It is requested by the FH, CH or TH of the enclosing scope and it invokes the *start* port of the CH component. Faults raised during compensation handling are propagated through the *faultComp* port to the component which requested the compensation.

Faults with associated fault handlers invoke the *faulth* port. All components except the one which raised the fault are either disabled or terminated. Finally, the fault handler is started. Faults without an associated fault handler invoke the *fault* port which propagates the error to the enclosing scope. In either case, the scope is considered finished.

The exit BPEL activity abruptly terminates a BPEL process. When such an activity in the scope is executed, the *exit* port is invoked and the *interrupt* port of the constituent components are also invoked, effectively terminating all components in the scope. Abrupt termination is propagated through the *exit* port to the enclosing component, eventually reaching the root of the component hierarchy.

B. Event handlers

Scopes may contain handlers for many message types. A different component type is declared for each available message type. In BPEL, an event handler is instantiated for each received message. However, the number of received messages is known only at run-time. For state exploration purposes it suffices to assume the minimum number of handlers materializing all possible interactions. We model the handler of each message type as a compound component enclosing two instances of the same component type that are executed in parallel.

C. Parallel execution

Flow activities coordinate the parallel execution of enclosed activities. The BIP flow activity component contain components corresponding to the encompassed activities and an atomic component coordinating access to the link values used for synchronization. An activity that conditions the execution of other activities (i.e. link source) has an attached atomic component that assigns a value to the associated link, when the activity is completed. On the other hand, an activity that is conditioned upon at least one other activity's execution (i.e. link target) has an attached atomic component that initiates the activity after the link value is set.

D. Variables

In our analysis all execution paths are explored, hence the concrete values of BPEL variables can not affect program correctness. To identify attempts of accessing an uninitialized BPEL variable, a corresponding BIP variable registers whether the former is initialized.

E. Message exchange activities and correlation sets

Constraints for enabling a message exchange activity are imposed by partner links, message exchange rules and correlation sets. In BIP, these constraints are modeled by boolean variables stored in the DAM component. Moreover, the BIP representation of correlation sets determines violations of correlation rules.

V. Code generation embedding for BPEL

We opt for a code generation embedding approach [10], such that BPEL is integrated into BIP by a translator that parses BPEL documents and the referenced WSDL descriptions, effectively converting them into a BIP model.

Translation is achieved by a depth first traversal of the process XML tree as shown in the algorithm of Figure 6. A post-order visit of the children nodes produces BIP code fragments, temporary stored in local variables. XML node properties (e.g node name, attribute names and values) determine the appropriate template applied for translating the node into BIP code. Templates contain static text and placeholders. Placeholders are replaced by node attribute values and code fragments produced by the translation of the descendant nodes. The implementation of the BIP model for the BPEL process is produced by recursive composition of the code fragments.

The execution semantics of a BPEL program is obtained by composing the structural representations in BIP of the individual BPEL constructs. The semantics of each construct is independent of the semantics of the referenced constructs. For example, the semantics of the flow statement is independent of the semantics of the contained activities. Compositionality of semantics in BIP is possible due to the separation of behavior from the interactions.

```
$text$ = function TRANSFORM($node$)
  $vars$ = [ ]    # stores BIP code fragments
  for each $child \in node.children$
    $vars.append$(TRANSFORM($child$))
  return APPLY_TEMPLATE($node, vars$)
end function
```

Figure 6: Transformation of XML subtrees of a BPEL process to BIP code fragments

Figure 7 shows the template of a BIP code fragment that models the semantics of the copy statement. It accepts two input parameters: (i) a counter which is incremented every time a template is applied and (ii) the multitude of message parts affected by the copy statement. Template lines that refer to placeholders with values derived from a collection, are repeated for every value in the collection when the template is evaluated. For example, line 2 of the template is

```
atomic type copy_noIgnMissing_fromToVar<i>(int id)
 data int var<k>= id    # for k = 1 .. parts
 data bool uninVar
 export port e0b0 <sp>   # for sp ∈ ports of the minim. interface
 export port e0b0 faultUninVar
 export port readVar(
   var<k>,    # for k = 1 .. parts
 )
 export port writeVar(
   var<k>,    # for k = 1 .. parts
 )
 place INIT, READY, COPY, DONE, TERMED, DISABLED
 initial to INIT
   # .. a sample of state transitions
 on readVar from READY to COPY
 on faultUninVar from COPY to TERMED provided uninVar==true
 on writeVar from COPY to DONE
 end
```

Figure 7: Template for translating a *copy* statement with *ignoreMissingFromData* set to false

appended four times when the copy operation affects four message parts.

The template producing code for an assign activity component is presented in Figure 8. It accepts three input parameters: (i) a counter which is incremented every time a template is applied (ii) a list of component types instantiated for each encompassed copy statement and (iii) a list containing the multitude of message parts affected by each copy statement. The connector types for transferring data to the copy component, named as RV<parts[cp]>in the assign template, are produced by the template shown in Figure 9. The template is parametrized by the number of message parts accessed by the copy component.

```
compound type assign<i>
 component <copy[cp]> C<cp>   # for cp = 1 .. size(copy)
   # ... sample connectors ...
 connector RDV<size(copy)> start1(
   C<cp>.start,    # for cp = 1 .. size(copy)
   )
 connector RV<parts[cp]> readCopy<cp>1(C<cp>.readVar)
   # for cp = 1 .. size(copy)
 connector WV<total> assign1(
     # total is the sum of the "parts" list
   C<cp>.writeVar,    # for cp = 1 .. size(copy)
   )
 connector RDV<size(copy)> faultUninVar1(
   C1.faultUninVar,
    C<cp>.term,    # for cp = 2 .. size(copy)
   )
   # ... sample of exported ports ...
 export port readCopy<cp> is readCopy<cp>1.xpr
     # for cp = 1 .. size(copy)
 export port assign is assign1.xpr
end
```

Figure 8: Template of assign component

```
connector type RV<parts>(e<parts>b p)
 define [p]
 data int tmp<i>   # for i = 1 .. parts
 data bool tmp<parts + 1>
 on p down {
    p.msg<i>= tmp<i>;    # for i = 1 .. parts + 1
 }
 export port e<parts>b xpr(
    tmp<i>,    # for i = 1 .. parts
    tmp<parts>)
end
```

Figure 9: Template for a connector used for transferring data in the copy component

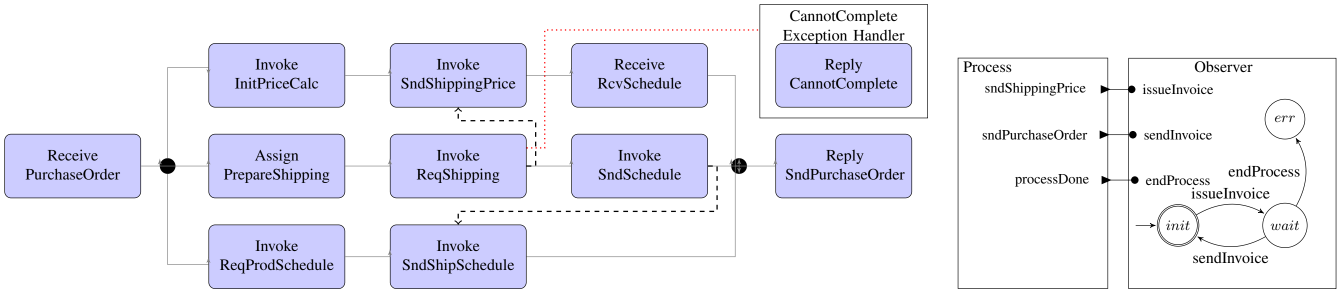
VI. Application

The application described in the BPEL specification contains many of the BPEL structures and concepts and has a simple and comprehensible business logic. We use this application in order to demonstrate the benefits of preserving the structure of the BPEL interactions during the analysis of the generated BIP model. We aim at the verification of model properties and in case of a correctness property violation, we should be able to trace back the location in the original source, where the error is manifested.

The process handles purchase orders issued by clients and spawns three concurrent execution branches that involve: production of the ordered items, shipper selection and invoice preparation. The shipment date is required for scheduling the item production while the transportation cost is needed for the invoice preparation. Data interdependencies are explicitly expressed by links that synchronize the execution of the three branches. The process is completed when the invoice is sent to the client. In Figure 10i we show the individual activities that compose the service and their interactions. Solid lines denote control dependencies, dashed lines show data dependencies between activities and dotted lines show the flow of exception handling.

Let us consider the following safety property: "If the invoice has been issued, the process must not complete before sending the invoice to the client". We construct an observer automaton [15], which monitors the execution of the process model and reports an error upon property violation. Figure 10ii shows the process model and the observer automaton. The invoice is issued upon completion of the SndShippingPrice invoke activity, which triggers the issueInvoice port of the observer and sets its current state to *wait*. The observer advances to the *err* state if the process is completed without having sent the invoice to the client.

The BIP state exploration tool reported a counterexample which violates the considered property. Due to the preserved process structure the obtained trace is mapped to the following activity execution sequence: PurchaseOrder, InitPriceCalc, PrepareShipping, ReqShipping, SndShippingPrice and finally the ReqProdSchedule activity which raised an uninitializedPartnerRole fault. Because the process does not have an appropriate fault handler it is abruptly terminated, without having sent the invoice to the client.



(i) Process flow

(ii) Interactions between the BIP model and the observer component

Figure 10: PurchaseOrder BPEL application and analysis of a safety property

VII. Related Work

We have introduced a structure-preserving embedding of WS-BPEL into BIP based on a structural representation of BPEL orchestrations. In order to place our proposal in the wide spectrum of formal methods for the analysis of service compositions, we adopt the comparison framework of [4]. In that article, the authors review the pros and cons of 35 published related works classified in three categories of semantic models, namely automata or labeled transition systems, Petri-nets and process algebras. Evaluation is based on the provided support for the analysis of three service composition characteristics: (i) language features guaranteeing the continuity of service delivery termed as connectivity support, (ii) correctness and (iii) Quality of Service (QoS).

The authors conclude that few of the considered formal methods address the connectivity characteristic in a satisfactory way and they specifically emphasize that exception handling and compensations are supported by a limited number of proposals. Only six of these works explicitly propose a model checking approach for analyzing correctness with respect to safety and liveness properties. Our approach can be compared with these related works as follows: (i) BIP offers more advanced correctness analysis techniques [5] that avoid the scalability limitations of conventional model checking and (ii) our language embedding for WS-BPEL 2.0 and the transformation of [9] for BPEL4WS 1.1 are the only works, which attempt to preserve the structure of the service composition program. Regarding QoS, BIP has been also used in the analysis of timing behavior and it comes with a tool [3] that supports performance evaluation.

For considering the effectiveness of BIP in comparison with related process algebraic approaches we recall [13], where the authors realized a semantic gap between WS-BPEL 2.0 and the π–calculus. They recognize that the notion of global state over BPEL computations, the message passing and the combination of sequencing with concurrency create interleaving and name binding behavior that cannot be faithfully represented in π–calculus. As a consequence of these findings, they extend π–calculus with a transactional construct, in order to provide formal semantics for the BPEL activities. In [9], a two–way mapping is introduced between BPEL4WS and the LOTOS process algebra. The authors claim that LOTOS has the expressive power to structurally represent BPEL processes, due to its compositionality semantics. However, LOTOS lacks a primitive analogous to the broadcast connector of BIP.

In order to explain the difference between the process algebra setting and the BIP approach, we refer to [6]. In process algebras we can use a series of operators with which processes evolve. In BIP components are characterized by their behavior (labeled transitions) and composition of behaviors takes place by means of interaction models and priority models, which essentially perform only memoryless coordination of behavior (the behavior is not modified when components execute some transition). With respect to a notion of expressiveness that characterizes the ability of some framework to coordinate components, process algebras have been shown to be less expressive than BIP.

Finally, it is worth comparing BIP with high-level modeling languages for component-based systems. One language that has been used for modeling compositional construction of web services is Reo [17]. Reo formal semantics has been defined with constrained automata, which however cannot preserve service composition structure, due to lack of powerful coordination operators, such as those in BIP. Another difference is that connectors in Reo are statefull, allowing for coordination behavior that does not preserve essential information about atomic behavior in components.

VIII. Lessons Learned

By modeling service coordination in BIP separately from the activities behavior, it was feasible to develop a model representation and translation with several attractive features:

- compositionality and reusability of model building blocks
- use of a limited number of connectors that sets the resulting model comprehensible

- direct mapping of the analysis traces into the BPEL process code due to the preservation of the service composition structure
- compositional translation of BPEL programs without retaining globally scoped data; this renders the translation amenable to parallel execution.

IX. Conclusion

We introduced a structural representation of WS-BPEL 2.0 compositions in the BIP component framework. This contribution involved the formal definition of the BPEL execution semantics, in terms of the BIP primitives and constructs. BPEL programs are transformed into BIP models by a language embedding, which preserves the service composition structure. As opposed to other formal representations and model checking approaches that work on a flattened model representation, with the presented embedding we can interpret the verification findings by referring to the original composition program. Moreover, BIP supports innovative formal analyses by reasoning on the model structure thus avoiding as much as possible the scalability limitations of model checking techniques.

As future work, we will further develop our language embedding to support bidirectional transformation. Correctness of our embedding will be addressed by advanced techniques for testing observational equivalence with various BPEL engines. However, BIP models for composite services will be eventually orchestrated by the BIP execution engine. Since BIP can subsume various composition mechanisms while preserving their structure, it should be possible to combine embeddings for different composition languages. By using BIP as a multilanguage host framework we could also include analysis for service choreographies. However, this involves the formal treatment of problems related to the compositionality of language semantics.

Acknowledgment

This work was performed in the framework of the TRACER(09SYN-72-942) project, which is funded by the Cooperation Programme of the Cooperation Programme of the Hellenic Secretariat for Research & Technology.

References

[1] A. Alves and A. Arkin, “Web services business process execution language version 2.0,” OASIS Committee Draft, May 2006.

[2] Z. Azmeh, M. Driss, F. Hamoui, M. Huchard, N. Moha, and C. Tibermacine, “Selection of composable web services driven by user requirements,” in *ICWS*, 2011, pp. 395–402.

[3] A. Basu, B. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis, “Rigorous component-based system design using the BIP framework,” *IEEE Software*, vol. 28, pp. 41–48, 2011.

[4] M. H. Beek, A. Bucchiarone, and S. Gnesi, “Formal methods for service composition,” *Annals of Mathematics, Computing and Teleinformatics*, vol. 1, no. 5, pp. 1–14, 2007.

[5] S. Bensalem, A. Griesmayer, A. Legay, T.-H. Nguyen, J. Sifakis, and R. Yan, “D-finder 2: Towards efficient correctness of incremental design,” in *NASA Formal Methods*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011, vol. 6617, pp. 453–458.

[6] S. Bliudze and J. Sifakis, “A notion of glue expressiveness for component-based systems,” in *CONCUR 2008 - Concurrency Theory*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, vol. 5201, pp. 508–522.

[7] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, “Web services description language (WSDL) 1.1,” Mar. 2001.

[8] D. Fahland, *Complete Abstract Operational Semantics for the Web Service Business Process Execution Language*, ser. Informatik-Berichte / Institut fur Informatik, Humboldt Universitat zu Berlin. Berlin: Inst. fur Informatik, 2005, no. 190.

[9] A. Ferrara, “Web services: a process algebra approach,” in *ICSOC ’04: Proceedings of the 2nd International Conference on Service Oriented Computing*. New York, NY, USA: ACM Press, 2004, pp. 242–251.

[10] C. Hofer, K. Ostermann, T. Rendel, and A. Moors, “Polymorphic embedding of DSLs,” in *Proceedings of the 7th international conference on Generative programming and component engineering*, ser. GPCE ’08, 2008, pp. 137–148.

[11] P. Hudak, “Modular domain specific languages and tools,” in *in Proceedings of Fifth International Conference on Software Reuse*. IEEE Computer Society Press, 1998, pp. 134–142.

[12] A. Lapadula, R. Pugliese, and F. Tiezzi, “Using formal methods to develop WS-BPEL applications,” *Science of Computer Programming*, vol. 77, no. 3, pp. 189–21, 2012.

[13] R. Lucchi and M. Mazzara, “A pi-calculus based semantics for WS-BPEL,” *Journal of Logic and Algebraic Programming*, vol. 70, no. 1, pp. 96–118, 2007.

[14] C. Ouyang, E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas, and A. H. M. ter Hofstede, “Formal semantics and analysis of control flow in WS-BPEL,” *Sci. Comput. Program.*, vol. 67, no. 2-3, pp. 162–198, 2007.

[15] M. Phalippou, “Executable testers,” in *Protocol Test Systems, VI, Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems, Pau, France*, 1993, pp. 35–50.

[16] C. Stahl, “A petri net semantics for BPEL,” Humboldt-Universität zu Berlin, Informatik-Berichte 188, Jul. 2005.

[17] S. Tasharofi, M. Vakilian, R. Z. Moghaddam, and M. Sirjani, “Modeling web service interactions using the coordination language reo.” in *WS-FM*. Springer, 2007, pp. 108–123.