Compositional execution semantics for business process verification

Emmanouela Stachtiari, Panagiotis Katsaros

Department of Informatics, Aristotle University of Thessaloniki, Thessaloniki, Greece.

Abstract

Service compositions are programmed as executable business processes in languages like WS-BPEL (or BPEL in short). In such programs, activities are nested within concurrency, isolation, compensation and event handling constructs that cause an overwhelming number of execution paths. Program correctness has to be verified based on a formal definition of the language semantics. For BPEL, previous works have proposed execution semantics in formal languages amenable to model checking. Most of the times the service composition structure is not preserved in the formal model, which impedes tracing the verification findings in the original program. Here, we propose a compositional semantics and a structure-preserving translator of BPEL programs onto the BIP component framework. In addition, we verify essential correctness properties that affect process responsiveness, and the compliance with partner services. The scalability of the proposed translation and analysis is demonstrated on BPEL programs of various sizes. Our compositional translation approach can be also applied to other executable languages with nesting syntax.

Keywords: formal verification, programming language semantics, WS-BPEL, BIP

1. Introduction

Businesses rely more and more on distributed, value-adding software applications in order to offer enterprise functionality to customers. Business Process Modeling (BPM) is a promising paradigm for integrating software components into a single executable unit, termed as process. The Service-Oriented Architecture (SOA) suits to the BPM paradigm, with respect to the composition of services into processes, which can be also deployed as services. Among existing languages for the specification of such processes, BPEL stands out by providing high-level primitives, and constructs for the definition of complex synchronous and asynchronous web service interactions. The used web services are autonomous and loosely-coupled components that possibly span different organizations. For the wide adoption of business process programming, it is essential to ensure reliability in order to avoid errors that may cause critical losses to the involved organizations. Additionally, the program has to fulfil correctness goals such as process responsiveness and compliance with partner services.

One approach towards ensuring reliability is by testing the process with emulating its interactions [1]. In this case, an adequate coverage of the program's control flow has to be achieved by selecting the appropriate test inputs. On the other hand, formal verification guarantees full coverage of execution paths for all possible inputs. Such an analysis has to be based on a formal specification of the language execution semantics, which involves nesting of service interactions using concurrency, isolation, compensation and event handling constructs.

Many works attempt to verify correctness by model checking a *formal model*, which is an abstract representation of the service composition program [2]. However, the original structure of the source program is not reflected in the formal model, thus rendering impossible to exactly locate the verification findings in the program's code. This is an inherent problem of most formalisms, which lack sufficiently expressive composition primitives for a model representation that preserves the service composition structure. The BIP (Behavior, Interaction, Priority) component framework [3] provides a minimal set of primitives adequate for preserving the service composition structure. It consists of an executable modeling language for layered transition systems, which has formally defined operational

Email addresses: emmastac@csd.auth.gr (Emmanouela Stachtiari), katsaros@csd.auth.gr (Panagiotis Katsaros)

semantics and mathematically proven expressiveness [4]. The BIP models can be formally verified with the BIP tools [5].

We use BIP to introduce a compositional semantics for BPEL, i.e. a semantics in which the processing for each BPEL construct is placed locally to a corresponding BIP component. Such a definition tackles the combinatorial problem of defining semantics for each possible combination of nested BPEL constructs. Compositional semantics can be defined for executable languages with nesting syntax if the execution semantics of enclosing and nested constructs can be defined independently from each other. To achieve such a definition in our approach, the semantics of nesting constructs are defined based on abstractions built-in by construction for the nested ones, while the latter are combined using coordination primitives that do not alter their semantics (just restrict their execution traces). A structure-preserving translator into the BIP language has been implemented that covers all activities of the BPEL standard. The translator transforms the BPEL programs into BIP models that contain the code needed for the verification of essential correctness properties. The check of whether the properties are met takes place by exploration of the reachable state space. If a property is violated, we are able to obtain a counterexample execution trace that contains the processing steps of BPEL activities, which lead to the error location.

In [6], we presented a first version of our translator for a limited set of BPEL constructs with more emphasis on the translation algorithm. The verification of a functional property for a showcase application scenario was also demonstrated along with evidence for its violation in the form of a counterexample. Here, we expose:

- the complete execution semantics of BPEL through a new methodology for compositional definition;
- the verification of a wide range of important correctness properties;
- the testing of our translator in mid-scale programs and their verification.

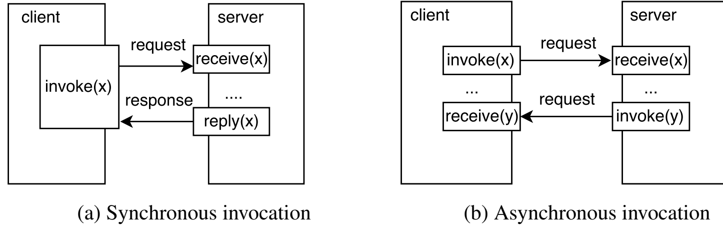
We note that the translation times were found to have a statistically significant *linear relation* to the number of states of the generated BIP model. The translator, the verification utilities for the properties of interest, as well as the BPEL programs of our experiments are available online in [7]. Verification is only one of the possible uses of our BPEL process models, which can be also used e.g. for test case generation based on the produced execution paths [8]. Moreover, in an independent research work [9], our approach was extended towards enabling the configuration of information flow policies for BPEL processes.

In Section 2, we discuss the design problems and the correctness of BPEL processes through a motivating example. Section 3 introduces the structure of our BIP model and the principles of the compositional approach for the definition of the BPEL execution semantics. These principles determine the interface and the behavior of BIP components, which allow implementing the semantics of the various BPEL activities. Section 4 encodes the BPEL execution semantics into safety properties that are *enforced in our model by construction*. Our modeling approach covers all activities of the BPEL standard, but the presentation is restricted to the most important activities and details for more activities are exposed in Appendix B. In Section 5, we present the verification of essential correctness properties that have been previously introduced in Section 2 and the formalization of additional useful correctness properties. Section 6 discusses the principles of the translation of BPEL programs in BIP. Section 7 shows results from the translation and analysis of mid-scale BPEL applications and the paper concludes with a critical review of the related work in Section 8 and our remarks for the exposed contributions in Section 9.

2. Correctness of BPEL processes: a motivating example

BPEL process implementations are based on web services (partner links) whose interfaces expose *service operations* written in the WSDL 1.1 language. Synchronous operations accept an input and block the invoker for the output, or a fault, to be returned. On the contrary, in asynchronous operations the invoker dispatches the input and forgets it. Thus, through the use of two asynchronous operations it is possible to apply a request-response interaction pattern that does not block the invoker. In this approach, a service is invoked with the first operation and the response is returned with a second operation, referred to as *callback*, exposed by the invoker. The use of asynchronous operations generally allows for complex service interaction patterns, such as parallel operation invocations, but it raises the need to effectively manage communication *sessions*, i.e. the stateful chains of dual service interactions. The assignment of messages to the correct session takes place by message *correlation*.

Atomic behavior in processes is realized with *basic activities*, such as the `invoke`, `receive`, and `reply`, which are used respectively to (i) invoke, (ii) receive input, and (iii) send output (or fault), with respect to specific service



(a) Synchronous invocation (b) Asynchronous invocation

Figure 1: Client and server side activities for synchronous and asynchronous invocations between processes.

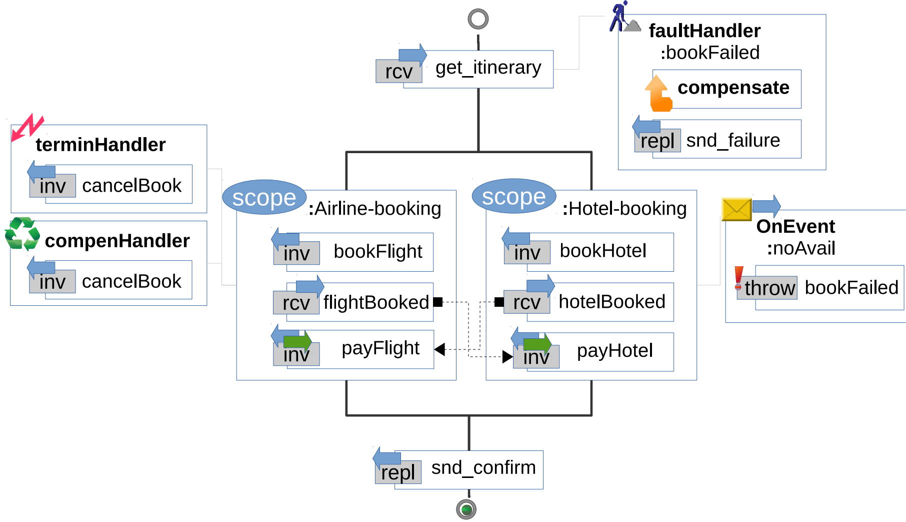


Figure 2: The *TravelBooking* process interacts with the *BookAirline* and *BookHotel* web services and on behalf of a client.

operations. Figures 1a and 1b show the client-side and server-side activities used for a synchronous (resp. an asynchronous) invocation of an operation `x`. A client-side synchronous invocation is implemented by a request-response `invoke`, while the asynchronous interaction relies on an one-way `invoke` of `x` and a `receive` of the callback operation `y`. Generally, the `assign` activity is used before sending and after receiving a message, in order to copy data between the message and the process's variables. BPEL's *structured activities* define workflows of activities, such as `sequence`, parallel `flow`, and other conditional and repeatable structures. The `scope` activity defines a local context for its enclosed activities, with its own data and error handling through compensation, termination and fault handlers. A scope also defines event handlers for incoming messages and timeouts.

Example 1. A BPEL process for travel booking is presented in Figure 2 with its activities shown in rectangular boxes. The activities for service interactions are labelled with the invoked operations. The bold, the thin and the dotted edges represent respectively relationships for the order of execution, the containment of handlers and the synchronization between activities.

The process provides to its clients the synchronous operation *get_itinerary* that responds with an output or a fault message. When a client wants to book a travel itinerary, a *get_itinerary* request is received along with the preferred hotel, room type and flight details. Two scopes are then executed in parallel that communicate respectively with the *HotelBookWS* and *AirlineBookWS* web services:

- The *Hotel-booking* scope invokes the asynchronous *bookHotel* operation of *HotelBookWS* to reserve the chosen hotel room. For this purpose, it uses an one-way `invoke` and continues its processing, while the response is pending. A `receive` waits for the confirmation in the *hotelBooked* callback operation. When the confirmation is received, the synchronous *payHotel* operation of the *HotelBookWS* is invoked for the payment. The progress of the whole process is then blocked on the synchronous `invoke`, until the receipt of the expected response. In

parallel to the normal flow, the scope also has an event handler that listens to requests for the *noAvail* operation. This is a callback operation that is invoked by the *HotelBookWS* service, if there is no availability for the chosen hotel room. Upon receipt of such a message, the event handler throws a *bookFailed* fault.

- The *Airline-booking* scope invokes the asynchronous *bookFlight* operation of the *AirlineBookWS* to book the flight and upon the confirmation receipt the synchronous *payFlight* operation is invoked for the payment. The booking cancellation of *AirlineBookWS* is invoked, when the scope is abruptly terminated or when it is compensated after having been completed.

Two synchronization links (dotted arrows) are used between the two scopes, in order to exclude the invocation of payment in each scope, before the confirmation is received in the other scope. The process responds to the client with a confirmation, after having completed the payment in both scopes.

Faults in any of the two scopes are propagated to the process level, where they are handled by the process's fault handler. In this case, the enclosed scopes are terminated, if they are still running, or compensated, if they have finished. Afterwards, the process replies to the client with a fault message. △

The main principles for the design of BPEL processes are summarized along the following three axes.

All available input from partner services must be received and handled. Input from partner services is received though incoming invocations. A dedicated `receive` activity should be therefore *reachable* at each point of the control flow, where such an input is expected. In Example 1, the *hotelBooked* callback is invoked, when the *HotelBookingWS* responds with the result of the asynchronous *hotelBook*, and the *noAvail* callback is invoked upon a booking failure. If the input to *noAvail* was neglected, the process would be blocked forever in the `receive` of the *hotelBooked* callback, unless there would be a timeout handler to limit the waiting time. Moreover, with respect to the process's requirements, the input to *noAvail* is essential to throw a fault at the process root level in order to terminate the whole booking attempt.

All the expected input to partner services must be provided. For each partner service, the syntax of its expected input is defined in WSDL, but there is no standard way for specifying the input's semantics and its relation to other events. In Example 1, the *cancelBook* request to the *AirlineBookWS* is sent, to cancel a booking request. The termination and the compensation handlers of the scope must send this cancellation request, otherwise the process will have sent and saved booking results that are not reflected in its state. If the cancellation request is omitted, the *PayFlight* request will still be expected from the partner service, though it will not be sent. However, the relationship between an operation and its callbacks is not explicitly defined and the process's runtime environment cannot detect and handle the missing inbound or outbound responses of the asynchronous requests. Due to this weakness, the responsibility for handling the issue is delegated to the process designer.

The process must hold a global view of the behavior that is composed. When implementing a scope, it is important to consider the environment in which it is executed. In Example 1, if the *Airline-booking* scope was the only scope running within the process, there would not be need for implementing a termination handler, because there would be no other scope to throw a fault while this scope is executed. The termination handling is necessary, due to the parallel execution of the scope with a scope which might fail. If a termination handler can never be triggered, we have a case of *dead code* in the process.

The BPEL processes need to fulfil several *correctness properties* that are application-agnostic, in order to ensure safety of the control flow and the sessions. Some of these properties are BPEL constraints that are identified by *standard types of faults* (e.g. the *conflicting receive* fault). Essential properties that are not BPEL constraints, though they have to be ensured for any BPEL process, are the following:

1. *No blocking*: The process will not be blocked indefinitely for receiving an incoming message.
2. *No dead code*: The process does not include code that cannot be executed.
3. *Process termination*: The process can always terminate.
4. *No incomplete asynchronous request-response patterns*: The process cannot terminate with asynchronous (outgoing or incoming) requests-response patterns that have not been responded.

The absence of dead code is important, in order to save memory space for the process execution and for the caching of operations by the CPU. A code segment may be unreachable, because of a logical error, such as conditions that are always false, or due to obsolete event handlers for messages that are not sent any more by the partner service. The process termination is essential for most processes that shouldn't run forever. A process may not be able to terminate,

due to a possible livelock, i.e. an execution path of tasks that are executed infinitely often. The last property concerns the behavioral compliance between the process and the partner services with respect to asynchronous invocations. That is, if an invocation has been received (or sent) by the process, and this invocation is part of a request-response pattern, then a response will always have to be sent (resp. receive).

The main approach to address the aforementioned correctness and other application-specific issues is through testing [1] and interactive simulation. In an effective testing approach the developer has to create test cases that cover the program's control flow up to an acceptable level, as well as to instrument the program with assertions to be checked. The test case generation includes the definition of input/output data for the emulation of process interactions. When an assertion violation is encountered the developer has still to explore the execution path the leads to the violated assertion. This is possible through interactively simulating the BPEL process for the specific test case. Testing is an iterative procedure, since every time that the process is changed towards correcting an error, all test cases have to run again to ensure that no other assertion is violated. Moreover, due to the complex synchronous and asynchronous web service interactions, the cost for effectively testing a BPEL process may be much higher than that for testing other types of applications.

In this paper, we introduce a modelling approach based on the BIP component framework along with the verification of the essential properties that is discussed in Section 5.1. Moreover, our approach can address additional verification needs that are detailed in Section 5.2. Within the development cycle of BPEL processes, correctness verification has to take place after process specification. As opposed to testing, verification is the only way to check functional properties against a process model with full coverage. If a property is violated, the needed corrections for the BPEL program are identified based on the automatically generated counterexample execution trace [6]. If all essential correctness properties have been verified, then non-functional aspects can be addressed, such as the verification of security, e.g. through an extension of our approach in [9], or timing properties, e.g. through statistical model checking [10]. The verification procedure is iterated until all properties are met.

3. BIP model for BPEL processes

3.1. The BIP component framework

BIP (Behavior-Interaction-Priority) [3] is a formal framework for building complex systems by coordinating the behavior of a set of atomic components. Behavior is defined as a transition system, extended with data and functions in C/C++. The description of coordination between components is layered. The first layer describes the interactions between components. The second layer describes dynamic priorities between interactions. BIP has a clean operational semantics, summarized in [11], that describes the behavior of a composite component as the composition of the behaviors of its atomic ones. A direct relation is thus established between the underlying semantic model (transition systems) and its implementation.

In BIP, atomic components are finite-state automata extended with variables and ports. Variables are used to store local data. Ports are action names, and may be associated with variables. They are used for interaction with other components. States denote control locations at which the components await for interaction. A transition is a step, labelled by a port, from a control location to another. It might be associated with a guard and an action, that are respectively a boolean condition and a computation defined on local variables.

Connectors relate ports from different subcomponents by assigning to them a synchronization attribute, which may be either trigger (▲) or synchron (●). The connectors represent sets of interactions that are non-empty sets of ports which have to be jointly executed. If all connected ports are synchrons, one interaction is executed and that is only if all connected components allow the transitions of those ports (rendezvous). If a connector has one trigger, the possible interactions consist of all the subsets of the connected ports which include the trigger port (broadcast). For every interaction, the connector might provide a guard and a data transfer that are, respectively, an enabling condition and an exchange of data across the ports involved in the interaction. Additionally, connectors can export ports for building hierarchies of connectors.

An *architecture* can be viewed as a BIP model, where some of the atomic components are considered as *coordinators*, while the rest are *parameters*. When an architecture is applied to a set of components, these components are used as *operands* to replace the parameters of the architecture. Figure 3 shows a simple BIP model for mutual exclusion

between two tasks. It has two components B_1, B_2 modelling the tasks and one coordinator component C_{12}. Initial states of the components are noted with an arc. The four binary connectors synchronise each of the actions b_1, b_2 (resp. f_1, f_2) of the tasks with the action b_{12} (resp. f_{12}) of the coordinator.

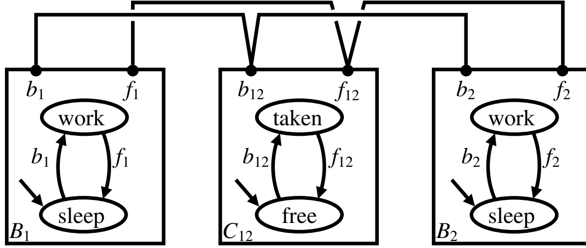


Figure 3: Mutual exclusion model in BIP

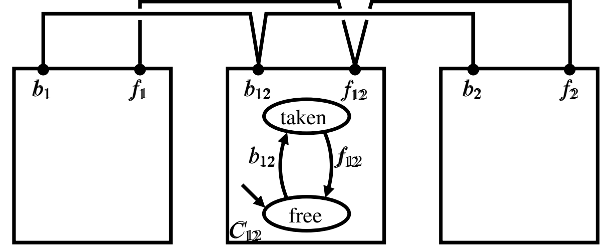


Figure 4: Mutual exclusion architecture

Figure 4 shows an architecture that enforces the mutual exclusion property on any two components with interfaces $\{b_1, f_1\}$ and $\{b_2, f_2\}$, satisfying the assumption of entering the critical section (e.g. in state `work`) when their b_i port is invoked.

Composition of architectures is based on an associative, commutative and idempotent architecture composition operator '$\oplus$' [12]. If two architectures $\mathcal{A}_1$ and $\mathcal{A}_2$ enforce respectively safety properties Φ_1 and Φ_2, the composed architecture $\mathcal{A}_1 \oplus \mathcal{A}_2$ enforces the property $\Phi_1 \wedge \Phi_2$, i.e. both properties are preserved by architecture composition.

Although the architecture in Fig. 4 can only be applied to a set of precisely two components, it is clear that an architecture of the same *style* - with n parameter components and $2n$ connectors - could be applied to any set of operand components satisfying the aforementioned assumption. We use *architecture diagrams* [13] to specify such *architecture styles*. An architecture diagram consists of a set of *component types*, with associated cardinality constraints representing the expected number of instances of each component type and a set of *connector motifs*. Connector motifs define sets of BIP connectors, are non-empty sets of *port types*, each labelled as either a trigger or a synchron. Each port type has a *cardinality* constraint representing the expected number of port instances per component instance and two additional constraints: *multiplicity* and *degree*, represented as a pair $m : d$. Multiplicity constrains the number of instances of the port type that must participate in a connector defined by the motif; degree constrains the number of connectors attached to any instance of the port type.

Cardinalities, multiplicities and degrees are either natural numbers or intervals. The interval attributes, '`mc`' (multiple choice) or '`sc`' (single choice), specify whether these constraints are uniformly applied or not. Let us consider, a port type p with associated intervals defining its multiplicity and degree. We write '$\mathtt{sc}[x, y]$' to mean that the same multiplicity or degree is applied to each port instance of p. We write '$\mathtt{mc}[x, y]$' to mean that different multiplicities or degrees can be applied to different port instances of p, provided they lie in the interval.

3.2. *BIP components and model structure for the BPEL activities*

Every BPEL process is defined by a single topmost scope that encloses variable declarations and stateful web service interactions through exchanged messages. Such a process is translated into BIP by preserving the structure of the used activities. Each activity is represented by a BIP component, called *activity component*, which explicitly defines the processing of that activity by the BPEL engine. *Basic activities*, like service invocations, are represented by the atomic BIP components of Table 1, apart from the `assign` activity, which is represented by one or more `copy` components. *Structured activities* are modelled as BIP compounds that enclose lower level activity components, as well as additional atomic components from those listed in Table 2. For every structured activity a set of connectors is generated, which is referred to as the *glue*, and may be accompanied by coordinating components that are attached to the glue. These coordinating components are introduced in Section 4 and they are not shown in the overall model structure.

`PROC` is the topmost component of the BIP model for a BPEL process. Its enclosed components specify the process's normal behavior (`norm`) and its associated fault-handlers (`faulthlrs`), that are executed as a response to a thrown fault. One more component is used to store the shared `data`, such as the process variables. Other shared data are the *partner links* and *correlation sets*, which identify services and communicating sessions respectively. The `norm`

BIP comp.	Behavior description
receive	handles a message receipt
reply	handles reply to a message
invoke	invokes a service operation
compensate	activates compensation of one or more scopes
valid	validates a variable's value w.r.t. its definition
empty	null behavior
exit	activates abrupt interruption of the process
throw	generates a fault
rethrow	re-throws a caught fault
copy	sets value to a variable, partner link or property

Table 1: Atomic components for basic activities of BPEL.

BIP comp.	Behavior description
data	manages access to the scope's data
timer	fires timer events
listn	handles multiple message receipts
links	manages access to synchronization links
rdlnk	reads a set of synchronization links
wrlnk	sets a synchronization link
loopctrl	controls loop execution
condctrl	controls conditional execution

Table 2: Atomic components for the BPEL semantics.

component encloses a primary activity (`act`) and event-handlers (`evhlrs`), that are scopes activated by timers or message receipts.

The overall model structure is further exemplified using phrase structure rules, where each rule refers to a compound (enclosing component shown in “<” and “>”) in the left part and its constituents (enclosed components) in the right part. The high level structure discussed so far is reflected by the following two rules:

⟨*PROC*⟩ ::= ⟨*norm*⟩ ⟨*faulthlrs*⟩ data

⟨*norm*⟩ ::= ⟨*act*⟩ ⟨*evhlrs*⟩

The components shown in the right-hand side of the `PROC` rule are also used in `scope` components, along with two more handlers: the compensation-handler (`comphlr`) that specifies behavior for the reversal of the scope's effects, and the termination-handler (`termhlr`) that controls the forced termination of the scope.

⟨*scope*⟩ ::= ⟨*norm*⟩ ⟨*faulthlrs*⟩ ⟨*termhlr*⟩ ⟨*comphlr*⟩ data

The lower-level structure of the mentioned handlers is given by the following rules:

⟨*evhlrs*⟩ ::= ((listn | timer) ⟨*evscope*⟩)+ | empty

⟨*faulthlrs*⟩ ::= (catch ⟨*act*⟩)+

⟨*comphlr*⟩ ::= ⟨*act*⟩

⟨*termhlr*⟩ ::= ⟨*act*⟩

with the “|” representing the allowed alternatives and the “+” showing the occurrence of one or more components of the preceding type. The `evscope` activity component has the same structure with the `scope`, though it has a different glue. This allows to address the prescription of the BPEL standard, which foresees a special scope for event handlers that treats the handler's starting activity (modelled by a `listn` or `timer`) as if it were part of its main activity.

The execution of parallel activities can be synchronized through synchronization links: an activity can defer other activities through outgoing (source) links and can be deferred through incoming (target) links. The components `source` and `target` handle the associated links using one `wrlnk` and one or more `rdlnk`.

⟨*source*⟩ ::= rdlnk+ wrlnk

⟨*target*⟩ ::= wrlnk rdlnk+

The rules and the details for modelling each activity are as follows:

⟨*act*⟩ ::= ⟨*source*⟩? ⟨*target*⟩? (receive | reply | invoke | compensate | valid | empty | exit | throw | rethrow
| ⟨*assign*⟩ | ⟨*seq*⟩ | ⟨*flow*⟩ | ⟨*loop*⟩ | ⟨*pick*⟩ | ⟨*if*⟩ | ⟨*scope*⟩)

⟨*assign*⟩ ::= copy+

⟨*seq*⟩ ::= ⟨*act*⟩+

⟨*flow*⟩ ::= links ⟨*act*⟩+

⟨*loop*⟩ ::= loopctrl ⟨*act*⟩+

⟨*pick*⟩ ::= ((timer | receive) ⟨*act*⟩)+

⟨*if*⟩ ::= condctrl ⟨*act*⟩+

where "?" specifies optional occurrence of the preceding component type. Structured activities enclose one or more activity components and control their execution either sequentially (`seq`, `if`, `loop`), in parallel (`flow`), or by deferred activation (`pick`).

3.3. Interface and behavior of activity components

The observable behavior of activity components [14] fulfills certain *assumptions (model abstractions)*. This enables the compound components to be defined based on the assumed observable behavior of the combined components i.e. *the glue is not tied to the combined behaviors*. For each activity component, its assumed observable behavior is ensured by construction, throughout the incremental building of the BIP model.

The fact that the glue is not tied to the combined behaviors allows the processing of each activity, i.e. the handling of control and dataflow events, to be *placed locally to the corresponding activity component*. The dataflow events comprise ports for reading/writing data to components that manage shared data (Sections 3.3.1, 3.3.2). The control events include: (i) *commands*, that tell an activity what to do (e.g. start or terminate), and (ii) *notifications*, that are generated by the activity (e.g. upon finishing or throwing a fault). The *basic interface* of activity components consists of ports for the propagation of distinct commands and notifications.

Command ports of activity components are fired *by their outer* components, with the aim to:

- start them (*start*);
- disable them (*dsbl*), for some execution scenario;
- terminate them (*term*), due to a fault in a parent scope;
- compensate them (*rvs*), during the compensation of a whole scope.

On the other hand, notification ports are fired by activity components, when they:

- throw a fault (*fault*), which can be any BPEL- or WSDL-defined fault;
- have finished (*fin*), either successfully (not disabled/terminated or having thrown a fault) or unsuccessfully;
- complete their compensation (*rvsd*).

The observable behavior of an activity component with respect to some goal, is formed by hiding the actions that are of no interest for this goal. Thus, the component might have different observable behaviours for different goals. Let us consider the behavior of the `empty` activity component in Figure 5 and that of the activity component in Figure 6, whose τ and *fault* actions are hidden. We define the bisimulation relation $R = \{(e0, s0), (e1, s1), (e1, s3), (e1, s4), (e2, s2)\}$ between the states of the two behaviors, i.e. one relation such that the related states imitate each other's observable actions leading to states that again are related. We say that any two behaviors are *observationally equivalent*, if and only if there is a bisimulation relation R with $(e0, s0) \in R$, where $e0$ and $s0$ are their respective initial states. It can be easily shown that the two behaviors in Figures 5 and 6 are *branching bisimilar* [15], which is an observational equivalence notion that preserves the branching structure of behaviors. This means that R preserves the computations *together with the potentials in all intermediate states* that are passed through, even if hidden actions are involved.

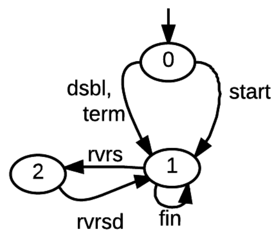


Figure 5: `empty` activity behavior.

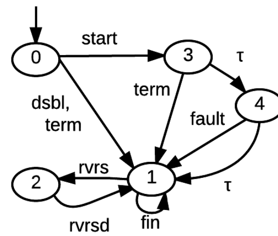


Figure 6: Example of activity behavior.

One general assumption for defining the glue is that the behavior of all activity components is branching bisimilar with the behavior of the `empty` component. This assumption is ensured by construction for all activity components; in atomic components, this is easily implemented in their behavior, which consists of limited control locations, while in compounds, the glue and possibly additional coordinating components enforce the externally observable behavior.

3.3.1. The state of service interactions

Since the lifetime of service interactions may span the execution of multiple activity components, the `data` components have to accommodate variables for sharing the service interactions' state. These variables store: (i) the (url) location of partner link services, that may change dynamically, (ii) sets of *correlation properties* that are instantiated and accessed by activity components for service interactions (`receive`, `reply`, `invoke`, `listen`), (iii) the enabled Inbound Message Activities (IMAs) for routing messages to the listening `receive` components, and (iv) the open IMAs, that identify incomplete inbound synchronous requests. More details for the representation of the service interactions' state and the detection of associated faults are given in Appendix A.

3.3.2. BPEL variables

The variables of a BPEL process store the messages' content or other business specific information that has to be shared among the activities (of a scope or globally), which may influence the control flow. Their data types are either XML types or WSDL message types with partitions, called *parts*. In our model, XML typed variables and variable parts are represented as local variables in the `data` components, which are accessed by activity components with dataflow processing, such as the `copy` and `receive` components. The `read` and `write` ports of activity components are used to read and assign variables of `data`, respectively.

For the values of BIP variables , we have adopted a data abstraction approach (details are given in Appendix A), which allows to identify (i) variables that have not been initialized, (ii) pairs of variables that hold the same value, and (iii) variables that are not assigned within a loop body.

3.4. Atomic BIP components

We adopt the common implementation approach of BPEL engines, which serialize the execution of basic activities. For this purpose, we enforce a mutual exclusion management in which the components for basic activities perform their execution (i.e. critical section) one by one; they remain blocked until they get the lock through their *allow* port, and invoke their *done* port to release the lock.

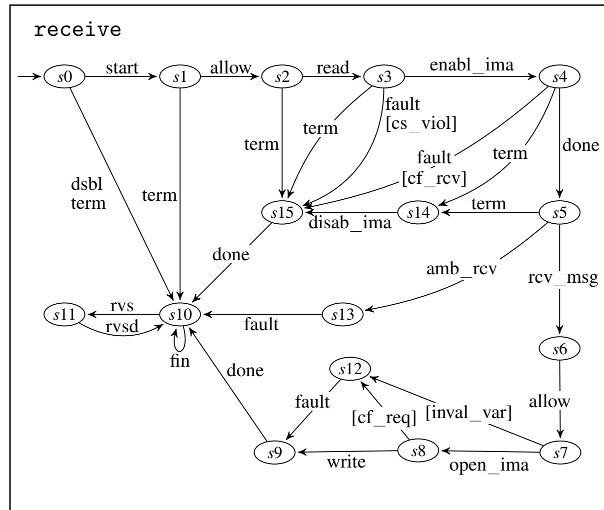


Figure 7: `receive` component for synchronous operation.

As an example, we show in Figure 7 the `receive` component for a synchronous operation and we discuss its differences from a `receive` component for an asynchronous operation. The `receive` is completed in two consecutive processing phases, one for the establishment of message listening, and one for the processing of the received message. The actions in these phases are included in two critical sections and the component remains idle in-between. Specifically, the component includes ports in order to:

- read the expected partner link and correlation sets (*read*),

- establish message listening (*enabl_ima*),
- stop message listening (*disab_ima*),
- detect an ambiguous receipt (*amb_rcv*),
- receive the message (*rcv_msg*),
- enlist the request identifier (*open_ima*),
- store the message and the correlation sets (*write*).

Component actions are guarded by the boolean variables shown in brackets, which represent detected faults. Possible faults are the *correlation violation* (cs_viol), *conflicting receive* (cf_rcv) and *conflicting request* (cf_req). Moreover, the *ambiguous receive* fault is detected when the *amb_rcv* port interacts with another listening component and the *invalid variable* (inval_var) is thrown if the message does not match the expected structure.

The `receive` component for asynchronous operations does not have the *open_req* port, since there is no need to enlist the requests of asynchronous operations; thus, no conflicting request faults are thrown in this case. In Appendix C, we expose the details of the other atomic components.

4. Compositional semantics definition

The execution semantics is enforced onto the composed behaviors of BIP compounds through coordination defined by the glue and additional components. A compositional definition is enabled by the principle that the coordination is not tied to the combined behaviors (shown in Section 3.3) and preserves the execution semantics of combined behaviors (no additional behavior is introduced, as explained below).

The acceptable behavior for the BIP compounds representing structured activities is captured in the form of safety properties defined over ports. To ease readability, all properties are defined using natural language statements. A set of *general safety properties* for any compound `C` with n enclosed components $\mathtt{A}_1 \ldots \mathtt{A}_n$ is the following:

- if `C` is disabled (*dsbl* port), so do all $\mathtt{A}_i$.
- if `C` is terminated (*term* port), so do all the $\mathtt{A}_i$ that can be terminated.
- `C` is finished (*fin* port) only if all $\mathtt{A}_i$ are finished.
- `C`'s compensation is completed (*rvsd* port) only if all $\mathtt{A}_i$ have been compensated.

Moreover, the order of compensation (*rvs* port) for all $\mathtt{A}_i$ is the reverse order of their *start* port activation, if there is an imposed order (e.g. for `sequence`). Otherwise, the compensation of all $\mathtt{A}_i$ is started simultaneously.

The aforementioned safety properties, as well as the invariants for basic activities and additional safety properties specific to each structured activity aim to formally capture the informally defined BPEL semantics from [16]. The invariants for the basic BPEL activities are enforced within atomic BIP components by design, such as in Figure 7 and in the components of Appendix C. For the structured BPEL activities, we introduce architecture styles that enforce safety properties - like those mentioned - associated with their semantics. This means that each property is built-in by construction within the used architecture style, i.e. it is implied by the behavior of the coordinating component(s) plus the used glue.

In the BIP compound for a structured activity, the architecture styles are instantiated into concrete architectures by defining a mapping from the styles' parameters to the compound's enclosed components (operands). This involves also a mapping of the parameters' ports to operands' ports. According to the results presented in [12], the safety properties of combined architectures are preserved in the compound; this result is also valid when the architectures are composed hierarchically. Moreover, all compounds only interfere with the lower level components by applying synchronization on their ports and synchronization always preserves the component invariants. In this way, we follow the principle of compositional semantics definition mentioned in the first paragraph of this section.

The following properties and the used architecture styles are specific to the most important structured activities. Details for the other activities are exposed in Appendix B.

4.1. BIP compound for the `flow`

Definition 4.1. A `flow` compound encloses one `links` component and n components $\mathtt{act}_1 \ldots \mathtt{act}_n$ that contain k `rdlnk` and m `wrlnk` components in total. The following properties have to be satisfied:

- if `flow` is started, so do all $\mathtt{act}_i$.

- some $\mathtt{rdlnk}_i$ can read a link (from `links`), only if some $\mathtt{wrlnk}_j$ has set the link (to `links`).

For the properties of the `flow` compound, two architecture styles were combined, namely the *Parallel* style in Figure 8 and the *Synch. links mngmt* style shown in Figure 9. The *Parallel* style enforces the first property of Def. 4.1 and the general safety properties that hold for any compound. The *Synch. links mngmt* enforces the second property of Def. 4.1.

All architecture styles are applicable to operands that: (i) have at least the ports assumed for the replaced parameter, and (ii) are branching bisimilar with the behavior assumed for the replaced parameter. For example, for the parameters $\mathtt{A}_i$ of the *Parallel* style (Figure 8) we assume the ports of the basic interface and the behavior of the `empty` component. The style is applicable to $\mathtt{act}_1 \ldots \mathtt{act}_n$ that fulfill these assumptions for $\mathtt{A}_i$. The style's coordinator `P` mediates the interactions of the basic interface of all $\mathtt{A}_i$ with the environment, so that the coordination fulfills the general assumption of Section 3.3: the observable behavior of the compound is branching bisimilar with the `empty` component. If the compound is started (`P`.*start*), all $\mathtt{A}_i$ are started due to a rendezvous connector ($n : 1$ means that the connector connects all $\mathtt{A}_i$). Therefore, the first property of Def. 4.1 is enforced, when the style is used in the `flow` compound. Furthermore, the connectors in Figure 8 that connect the *dsbl*, *term* and *fin* ports enforce the general safety properties.

The *Synch. links mngmt* style manages the access of *m* parameters `WRlnk` and *n* `RDlnk` to the synchronization links of the `Lnk` parameter. Each `WRlnk` sets a link through the *set* port, whereas each `RDlnk` reads a set of links through the *get* port. Ports *set* and *get* are connected to `Lnk` through the WRVAR (resp. RDVAR) connectors that enable exchange of data. A `RDlnk` can read a set of links (`RDlnk`.*get*), only if some `WRlnks` have set these links (`WRlnk`.*set*), since each `Lnk`.*get* port is assumed to be guarded with this condition. Therefore, the second property of Def. 4.1 is enforced, when the style is used in the `flow` compound. Since for the `RDlnk` and `WRlnk` parameters we assume a trivial behavior with a single state, the style is applicable to the `rdlnk` and `wrlnk` components of `act` (*set* and *get* ports are exported by `act`).

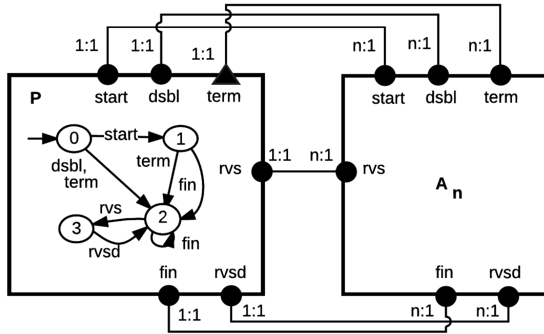


Figure 8: *Parallel* style

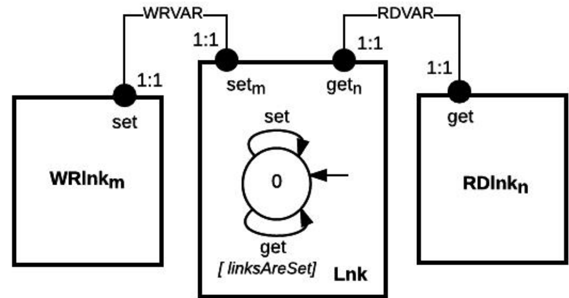


Figure 9: *Synch. links mngmt* style

4.2. BIP compound for the `scope` and PROC

Definition 4.2. A `scope` compound encloses the components (i) `norm`, (ii) `faulthlrs`, (iii) `termhlr`, (iv) `comphlr` and (v) `data`. Let us also consider that the enclosed components (i) to (iv) contain *k* `rddat` and *m* `wrdat` components in total, which read (resp. assign) variables stored in `data`. The following properties have to be to satisfied:

1. if `scope` is started, then `norm` is started.
2. if `norm` throws a fault and `norm` is terminated while `termhlr`, `comphlr` are disabled, then `faulthlrs` is started only if `norm` has been terminated.
3. if `scope` is terminated while `norm` is executed, `norm` is terminated and also `faulthlrs`, `comphlr` are disabled, then `termhlr` is started only if `norm` has been terminated.
4. `scope` is finished successfully, only if `norm` has finished, without being disabled, terminated or having thrown a fault.
5. `scope` is finished unsuccessfully, only if `faulthlrs` or `termhlr` have been executed and finished.
6. if `faulthlrs` throws a fault, the fault is thrown by `scope`, then `faulthlrs` is terminated only if `scope` is terminated (i.e., by the enclosing `scope` that will handle the fault).

7. `comphlr` is started, only if a successfully finished `scope` is compensated.
8. `scope` is finished only if `norm`, `faulthlrs` and `termhlr` are finished and while `comphlr` is not being executed.
9. the values of variables in `data` are modified only if they are assigned by some `wrdat`.

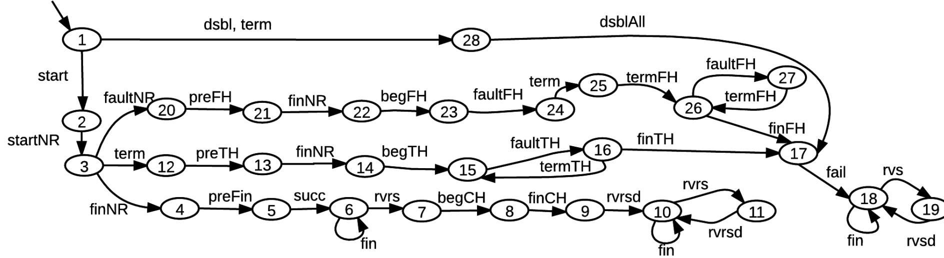


Figure 10: Coordinator of the *Scope* style

For the properties of Def. 4.2 two architecture styles were combined, namely the *Scope* and one *Data mngmt*. The *Scope* style is used only in `scope`, in order to fulfill properties (1) to (8). The style has a coordinator that is shown in Figure 10 and takes as parameters the `norm` (NR), `faulthlrs` (FH), `termhlr` (TH) and `comphlr` (CH). The coordinator's role is to export the `scope`'s basic interface and to coordinate the parameters with respect to their basic interface. This happens under the assumption that `faulthlrs`, `termhlr` and `comphlr` are branching bisimilar with the `empty`. The coordinator enforces property (1) by starting `norm` (*startNR* port) after the scope is started (*start* port). For property (2), a fault thrown by `norm` ($faultNR$ port) is followed by the preparation needed for starting fault handling (*preFH* port), that involves terminating `norm` and disabling `faulthlrs` and `comphlr`. Afterwards, `norm` must first finish ($finNR$ port) before `faulthlrs` is started. Property (3) is treated accordingly. For property (4), the coordinator enables the *succ* port after `norm` has finished without being terminated or thrown a fault. Similarly, for property (5), the coordinator enables the $fail$ port after either `faulthlrs` ($finFH$) or `termhlr` ($finTH$) is finished. For property (6), the coordinator enables $fault$ after `faulthlrs` has thrown a fault ($faultFH$). Then, it waits for the `scope`'s termination (by the enclosing `scope` that will handle the fault) before invoking the termination of `faulthlrs` (*termFH*). Property (7) holds because the coordinator disables `comphlr` whenever the scope is not going to finish successfully, during the preparation for termination (*preTH* port) and fault handling (*preFH* port). Finally, property (8) is enforced by starting `comphlr` (*startCH*) after `scope`'s compensation is invoked, provided that `scope` has finished successfully.

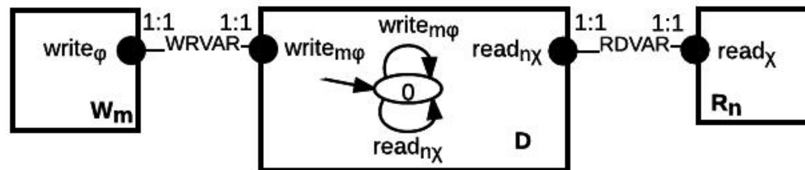


Figure 11: The *Data mngmt* style

The *Data mngmt* style in Figure 11 is used to fulfill property (9). The style has one parameter `D` that stores some variables, m parameters `W`, and n parameters `R`. The `W` have φ ports for assigning variables, whereas the `R` have χ ports for reading variables. The style is applied to each `scope` by mapping the `wrdat` and `rddat` components to the `W` and `R` parameters, while the `data` component is mapped to `D`. Property (9) is fulfilled by connecting `data` exclusively with the `wrdat` components that are enclosed by the `scope`.

The `PROC` compound uses the *Proc* style, whose coordinator is different from the coordinator of the `Scope` style with respect to the following four aspects: (i) it does not have the branches starting with *term* and *disbl*, since `PROC` is a root component and cannot receive events that come from enclosing components, (ii) the $faultFH$ is not followed by the `faulthrs` termination, but immediate interruption of the process occurs instead, (iii) if an `exit` component

is executed within `norm` or `faulthlrs` (i.e. invoking coordinator's *exit* port), then the interruption of the process occurs, and (iv) it does not need to enable the ports *succ*, *fail*, *rvs* and *rvsd*.

5. Verification of correctness properties

In the verification procedure of Figure 12, we attach a fault injection component, as well as observer automata (monitors) [17] for observing the state of the BPEL process model, whereas the state space exploration takes place with one of the BIP tools [5]. The building of monitors, that is discussed later, requires information given by the user in a configuration file. The procedure steps are shown with rectangles and the input and output data for each step with dotted lines. More specifically, the individual steps are as follows:

***Input*:** (i) the BPEL program and its WSDL definition files , (ii) the configuration file
***Output*:** the verification verdict

Step 1 *BPEL-to-BIP translation.* The BIP model is built together with the monitors and the fault injection component through the translation of the BPEL program and its WSDL definition.

Step 2 *Verification.* The verification output is produced by executing the BIP state space exploration tool. The output contains diagnostic messages appended by the monitors, the activity components and the BIP exploration tool.

Step 3 *Analysis of verification.* The verification verdict for each property is produced. For some properties, it suffices to inspect the diagnostics in the verification output, while for others a post-processing program is used[1]. If a property is found to be violated, a counterexample execution trace is generated through post-processing the verification output with a dedicated program.

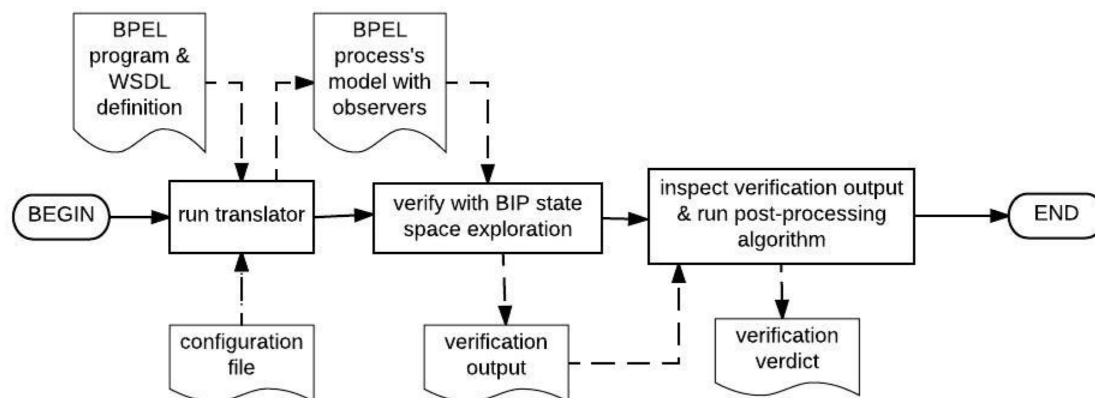


Figure 12: The BPEL process verification procedure.

In the following subsection, we discuss specifically the verification of the essential correctness properties from Section 2. Moreover, the verification of other important properties (standard BPEL faults, properties specific to the application functionality and compliance with a session specifications) is discussed in Section 5.2.

5.1. Essential properties

No blocking
To detect blocking by *receive* activities, we compose the process model with a fault injection component that consists of two non-deterministically selected states: the state representing availability of partner links for communication (AVAIL state) and the state in which partner links cannot send messages (NAVAIL state). This component also exports

[1]The post-processing program and the configuration file template are available online in [7].

a *send* port, enabled at the AVAIL state, which is synchronized using a rendezvous connector with the *rcv_msg* ports exported by the `receive` components. If the process model can be blocked by an incomplete `receive`, then the state-space exploration will detect deadlocks in the execution paths, in which the NAVAIL state was reached. Indeed, the *rcv_msg* ports will not be triggered if the NAVAIL state is reached and a deadlock will be met unless each `receive` is stopped by a `timer` component. The attachment of the fault injection component only affects the model by restricting the execution traces to those observed when a failure in the process environment exists. Thus, the execution semantics of the process is not changed.

Process termination
We detect livelocks caused by eternal loops that prevent process termination. In particular, we detect execution paths in which the variables used in the loop exit conditions are not updated during the loop's execution. For each evaluation of an exit condition, the values of variables are compared with the values they had in previous evaluation. This check is integrated within the `loopctrl` components, which print a diagnostic message during state exploration when reaching states where the aforementioned condition is detected.

No dead code
Dead code consists of basic activities, which are not started in at least one execution path. Such activities are detected using a script that processes the output of state-space exploration. During the exploration, each basic activity component prints its id twice: in one message at the initial state and in a second message once it is started. The output is post-processed by a script that adds the ids of the messages printed at the initial state in a hash set and removes the ids of the messages at starting. The ids that remain in the hash set after the output is processed indicate the components that are dead code. Note that the post-processing algorithm scales linearly with the number of printed messages, since the cost for adding and retrieving each id in the hash set is O(1) in the average case.

No incomplete asynchronous request-response
In order to verify that every asynchronous request-response pattern is responded in all execution traces, we introduced two monitoring components for observing the incoming (resp. outgoing) patterns. The monitor of outgoing patterns observes the dispatch of the first message (request) to partner services and the receipt of the second message (response). Upon process termination, if the dispatched requests are more than the received responses, the monitor outputs the property's violation. Similarly, the monitor of incoming patterns prints a message, if more receipts of requests are observed than the dispatched responses. The attachment of monitors does not affect the execution traces, because they are connected with the activity components using broadcast connectors, through which only the activity components can trigger the monitors and not the other way around. Thus, the execution semantics of the process is not changed.

For this check, it is necessary to define which request and response messages are associated, since this cannot be derived from the WSDL description. This mapping is given by the user in a configuration file.

5.2. Additional correctness poperties

Our BPEL process models can be used for verifying properties that are defined in the context of the application's functionality. For example, in [6], we verified a purchase order BPEL process with respect to the property: "If an invoice has been issued, the process must not complete before sending the invoice to the client". Such properties are verified through the use of application-specific observer automata.

The properties related to standard BPEL faults, such as those checked in [18] (message delivery atomicity, no session ambiguity, possible inputs) and [19] (the so-called conflicting receive in BPEL) are checked by the activity components, which throw these faults. Thus, we represent the handling of every such fault by the responsible scope.

Besides that, our model can be checked for compliance with the acceptable message sequences within a session, as it is demonstrated in [20]. This entails the attachment of an automaton representing the language of acceptable message sequences, while observing the exchanged messages identified by the session. The process's compliance is then verified, if the observer automaton is in an accepting state, when the process is terminated. Properties like those in (4) of Section 2 are examples of compliance with a language of sequences with two exchanged messages (asynchronous request and response). An example language for a complete session between the process of Section 2

and the Airline Booking service is shown in the observer automaton of Figure13. Transitions are labelled with send and receive message actions of the BPEL process that are allowed at each state. Valid message sequences lead to the states 4 and 5 which enable the port *valid*. If a message is exchanged that is not allowed, a non-accepting state is reached that is not shown and the message sequence is rejected. Thus, the process is not compliant with the session language specification, if there is an execution path in which it sends, for example, cancelBook twice.

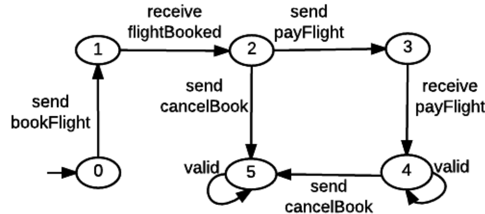


Figure 13: Observer automaton for a language of acceptable messages.

6. BPEL to BIP translation

A code generation embedding [21] was developed, such that BPEL is translated into BIP by parsing BPEL programs and their referenced WSDL descriptions.

Our algorithm implements a single pass syntax-directed translation [22] through a recursive decent parsing of the BPEL XML tree and a postorder call of the BIP code generation function for the tree nodes. The code generation function is invoked only for the XML elements corresponding to BPEL activities. A code fragment, that is generated upon each function call, is part of an incrementally built BIP model. The symbol table structure stores information derived from the parsing of the BPEL XML tree (e.g. visible BPEL variables) and the referenced WSDL descriptions, as well as from the code generation results. The latter is necessary for building a hierarchical model, in which each BIP compound refers to the enclosed BIP components and their exported ports.

The code generation function uses templates of code for BIP components with placeholders. The tokens of XML elements (tree nodes), such as the element tag and attribute values, determine the template to be used. The placeholders are replaced by BIP code that is generated based on: (i) information retrieved from the symbol table, and (ii) the attribute values of the XML element.

```
 1 atomic type copy_<i>()
 2    /* ... sample of data ... */
 3    data int err
 4    data int <var[k]>    /* for k = 1 .. K */
 5    /* ... sample of states ... */
 6    place s0, s1, s2, s3
 7    /* ... sample of transitions ... */
 8    initial to s0
 9    on read_<i>from s1 to s2 do {
10       if <var[k]>==-1 then err=10   fi    /* for k = 1 .. K */
11    }
12    on fault from s2 to s3 provided (err>0 )
13    on write from s2 to s3 provided (err==0 )
14 end
```

Figure 14: Template of BIP code for the copy activity

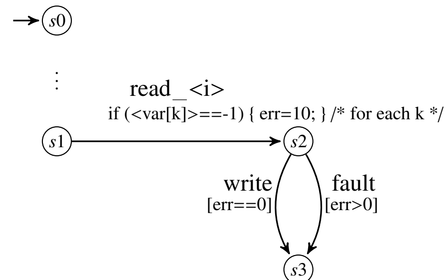


Figure 15: Behavior of the copy template

Figure 14 shows the template of a BIP code fragment that models the copy activity. The template starts with the declaration of the copy atomic component, which includes data (lines 3-4), a set of states (line 6) and transitions (lines 8-13). The template accepts two input parameters: (i) an auto-incremented component identifier *i*, (ii) a list

var[*k*] of size *K* with component variables for storing the message parts. The placeholder for *i* is noted with <i>. The lines ending with /* for k=1..K */ comments (i.e. lines 4 and 10) are repeated for each element in *var*[*k*]. The behavior corresponding to the transitions of this template is shown in Figure 15. Specifically, the `copy` starts from state *s*0. Let us consider that it reaches *s*1, where it reads the message parts that should be copied (port *read_<i>*). After checking whether there are uninitialized message parts, either the message parts are copied to the new variables (port *write_<i>*) or a fault is thrown (port *fault*).

```
 1 compound type assign_<i>()
 2    /* ... enclosed components ... */
 3    component <cp[k]>  C<k>    /* for k = 1 .. K */
 4    /* ... sample of connectors ... */
 5    connector <wrConn>  write_<i>1(
 6       C<k>.write , /* for k = 1 ..K */
 7    )
 8    /* ... sample of exported ports ... */
 9    export port write_<i>1.xpr as write_<i>
10    export port C<k>.read_<cp[k]>  as read_<cp[k]>    /* for k=1..K */
11    export port
12             C<k>.fault,    /* for k=1..K */
13    as fault
14 end
```

Figure 16: Template of BIP code for the `assign` activity

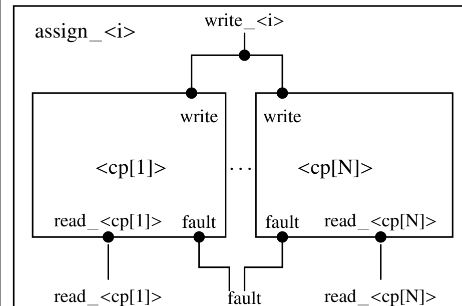


Figure 17: Behavior of the *assign* template

The template of BIP code for the `assign` activity component is shown in Figure 16 and the component's structure is illustrated in Figure 17. The template starts with the declaration of the compound (line 1) and the enclosed `copy` components (line 3). Lines 5-7 show the declaration of a connector that synchronizes all the `copy`.*write* ports. This connector exports a *write* port at the `assign`'s interface (line 9). A number of *read* ports are also exported, one for each included `copy`.*read* port (line 10). On the other hand, a single *fault* port is exported for all included `copy`.*fault* ports (lines 11-13). The template accepts three input parameters: (i) an auto-incremented component identifier *i*, (ii) a list *cp*[*k*] of size *K* with the enclosed `copy` components (iii) the connector *wrConn* used for the assignment of message parts.

7. Experiments on the verification of BPEL programs

The scalability of our analysis in real-size BPEL programs and the effectiveness of the verification approach were tested using a number of BPEL programs of various sizes from [23] and [24]. The programs mentioned in the rows of Table 3 were first translated and then analyzed with respect to the essential properties of Section 5.1.

For each program, Table 3 summarizes in the corresponding row the statistics for the size of the BIP model, the translation/verification time and the obtained results. The shown times in *ms* were measured on a 64-bit machine with an Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz and 16 GB RAM running Ubuntu 14.04. The first five columns show the number of components, connectors and reachable states (RSS), as well as the CPU time for translating and verifying the program. The worst time taken for the program translation was 16 sec, whereas for the program verification was almost 44 sec. The translation times were found to have a statistically significant *linear relation* to the number of states of the generated BIP model (Appendix D).

The last four columns show the verification results, for each one of the properties of Section 5.1. We note with dash the cases in which a property does not raise some correctness issue or a property's definition is not applicable. For example, we don't need to verify "no blocking" in programs, which do not wait for incoming messages other than the first messages that create new process instances. Also, "process termination" is not relevant to programs that do not have loop activities. The verification result for the two aforementioned properties is noted with either "pass" or "fail". For the two other properties, Table 3 shows the number of violation cases out of the total number of checked cases. For example, for the property "no dead code", it is shown the number of non-reachable basic activities together

Table 3: Statistics and verification results for analyzed BPEL processes.

Process ID	# comp.	# conn.	# RSS	transl. time	verif. time	no block.	process termin.	no dead	no incompl.
AmericanAirlines_12	42	216	12533	6471	21112	-	-	0 of 13	1 of 2
AmericanAirline_59	22	127	69	3632	8	-	-	0 of 4	1 of 1
BookRating_50	20	129	69	4496	18	-	-	0 of 4	0 of 0
BookStore1_52	40	243	1791	5118	1204	-	-	1 of 8	0 of 3
BookStore2_49	42	247	1791	4834	1211	-	-	1 of 8	0 of 3
BuyBook_48	85	454	2437	6441	567	fail	-	2 of 13	0 of 1
BuyBook_51	89	1359	49295	15967	43737	fail	-	29 of 54	0 of 3
BuyBook_53	50	836	13607	9933	3242	fail	-	2 of 25	0 of 5
BuyBook_54	27	326	983	6167	245	fail	-	0 of 15	0 of 5
BuyBook_55	34	341	1707	9796	349	fail	-	0 of 19	0 of 5
DeltaAirline_56	13	90	69	3736	15	-	-	0 of 4	0 of 1
Employee_57	12	89	69	3712	18	-	-	0 of 4	0 of 0
Travel_58	25	243	907	5721	209	fail	-	0 of 13	0 of 3
TravelApproval_41	168	811	32907	11089	24364	-	pass	0 of 27	0 of 0

with the total number of basic activities. The last column shows the number of incomplete asynchronous requests and the total number of asynchronous requests.

The "no blocking" property is violated in all programs in which it was checked. By inspection of the execution traces it was found that only the programs BuyBook_48 and BuyBook_51 could avoid the eternal waiting for messages using timers, but not in all cases. The "process termination" property was relevant only for the TravelApproval_41 program, where it was found to be satisfied. The property "no dead code" was violated in five programs. More specifically, we detected fault handlers that cannot be started. Some of these handlers were included, in order to be invoked, if a partner service is not available. However, such a fault can be thrown only from BPEL engines, which support this non-standard fault. On the other hand, we do not provide built-in support for non-standard faults and for this reason the corresponding fault handlers cannot be started. Finally, two programs were found, in which the property "no incomplete requests" was violated. By inspection, we confirmed afterwards that these programs do not respond in all cases to services that invoke the process.

8. Related Work

In order to place our approach in the broad range of formal methods for the analysis of service compositions, we start with the comparison framework in [2]. In that article, the authors review the pros and cons of 35 related works classified in three categories of semantic models, namely automata or labelled transition systems, Petri-nets and process algebras. The authors conclude that few of the considered formal methods address in a satisfactory way the correctness properties for continuity of service delivery, and they specifically emphasize that only a limited number of proposals support exception handling and compensations, as is the case in our approach.

The works of [25] and [19] are the only BPEL formalization approaches accompanied by available translators (the BPEL2oWFN and the BPEL2PNML tools respectively). A comparison between the two frameworks is presented in [26]. The resulting models of both approaches can be checked by tools that verify temporal properties on Petri-nets. The models in [19] do not represent data dependencies but they can be checked efficiently by tools that analyze the structure of WF-nets. Such examples are the WofBPEL tool in [19], which can check application-agnostic properties and the tool in [27] that checks the conformance of a BPEL process model with respect to message logs. On the other hand, models in [25] are built with an abstraction level that suits the intended analysis goal, with the possibility to represent data dependencies. These models can be checked for general and application-specific properties only by reachability analysis, since they do not comply to the structural requirements of WF-nets. Compared to the two aforementioned translation approaches, our work exposes the way that the translation rules ensure the BPEL semantics

and provides more expressive means (i.e. the observer automata vs temporal logics) for the definition of application-specific properties.

The distinct feature of our approach is a compositional definition of execution semantics for BPEL, such that there is no need to model all possible combinations of nested BPEL constructs. Similar approaches are those in [28], for the BPEL, and in [29], for an artificial variant of OWL-S. The latter, does not feature complex event handling structures like the ones offered by BPEL. Moreover, the main goal of semantics definition was the implementation of an orchestration execution engine, as opposed to ours, which is the verification of correctness properties. In [28], the authors describe a stepwise refinement approach for a structure-preserving modeling of BPEL in Event-B. This work, like ours, is supported by a translation tool and considers both generic and custom correctness properties for verification. We instead follow a constructive approach in building the model by gradually imposing constraints while preserving invariants. In another work [30], a BPEL translation to FIACRE [31] is presented. FIACRE is a formal language for modeling both the behavioral and timing aspects of systems. This work is focused on the translation approach, rather than a compositional semantics definition. No details are provided for the verification of BPEL programs and for the scalability of their translation to programs of various sizes.

For considering the effectiveness of BIP in comparison with the process algebraic approaches we recall [32], where the authors realized a semantic gap between BPEL 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, in order to provide formal semantics for the BPEL activities, they extend π–calculus with a transactional construct. In [33], a two–way mapping is introduced between BPEL 4WS 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 expressive primitives like for example the broadcast connector in BIP.

The difference between the process algebra setting and that of BIP is thoroughly studied in [4]. The components in BIP are characterized by their behavior (labeled transitions) and their composition takes place by means of interaction models and priority models, which essentially perform memoryless coordination of behavior. On the other hand, in process algebras processes evolve with the use of operators for composition. 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. Regarding the compositional semantics definition, the coordination means of process algebras do not preserve the invariants of composed behaviors, since the behaviors evolve.

Moreover, 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 [34]. However, Reo's semantic model, which is the constrained automata, cannot preserve the BPEL process structure due to lack of powerful coordination operators, such as those in BIP. The lack of structure preservation, along with the fact that connectors in Reo are stateful, would complicate the definition of a compositional semantics definition for BPEL in Reo.

Finally, in an independent research work [9], our structural BPEL process representation from [6] was extended towards the verification of service non-interference, which is an interesting application of our compositional semantics framework. To do so, the process designer must provide a configuration file (.xml) with authorization rights, that is, a list of owners and authorized readers - partner link services - for critical data. Then, the configuration synthesis algorithm takes as input the BIP model of the BPEL program and the configuration file and builds information flow dependency graphs by considering all implicit and explicit data dependencies in the system. In case a total configuration file can be generated by the tool, the system information flow is then considered non-interferent with respect to the initially defined configuration. Otherwise, the system is interferent and the system designer has to re-define the initial configuration by utilizing the obtained counterexample. The calculated configuration is optimal, i.e. only data that need to be protected is configured as critical. The ultimate aim is to reduce the security processing overhead like cryptography encryption and decryption, signature calculation, certificate verification, etc.

9. Conclusion

The main contribution of the paper is the compositional semantics definition for business processes. Such a definition tackles the combinatorial problem of defining semantics for each possible combination of nested statements. In particular, we focused on the BPEL language for business processes, which features a multitude of activities nested

using concurrency, isolation, compensation and event handling constructs. Our approach relies on the BIP component framework. More precisely, a hierarchical BIP model is built by applying sets of BIP connectors to certain observable events of the lower-level components. The connectors determine the execution semantics of the BPEL process and its activities; they are based on assumptions for the observable behavior of the combined components that are *ensured by construction*. The semantics is defined by introducing safety properties for the BIP compounds that represent the BPEL activities. These properties are then enforced using architecture styles, as in [12].

We also presented the design of a code generation embedding based on our semantics definition that translates BPEL programs into BIP models. In a number of experiments with real-size programs, we noticed that the translation times have a statistically significant linear relation to the number of states of the generated BIP model. The translator serves the purpose of verifying business processes written in BPEL. We presented the verification of essential correctness properties, as well as other properties that are defined in the context of the application's functionality. However, verification is only one of the possible uses of our semantics framework. As a proof of concept, the reader is referred to [9], where our model was extended towards enabling the configuration of information flow policies for BPEL processes.

As future work, we consider the use of a more fine grain data abstraction that will represent data as a set of constraints on their data range. This would result in a tighter overapproximation of the process's behavior. Another perspective is to deliver a new implementation of our translator using: (i) XPath (XML Path) expressions [35] for parsing BPEL activities, and (ii) XSLT (XML style transformation) expressions [36] for transforming their content into BIP code. Such an implementation will provide enhanced readability and extensibility of the parsing and code generation rules, which will not be embedded in the translation program. Moreover, it is also possible to combine our translator with BIP embeddings for other composition languages. By using BIP as a multilanguage host framework we can analyze the behavioral correctness of service choreographies. However, this involves the formal treatment of problems related to the compositionality of language semantics. Finally, our compositional semantics approach can also be used in studying BPEL processes, which use dynamic partner links [37], i.e. endpoint references that are passed to the BPEL process at runtime. For this purpose, we plan to use Dy-BIP [38], which is a dynamic extension of the BIP component framework.

References

[1] C. A. Sun, Y. Zhao, L. Pan, H. Liu, T. Chen, Automated testing of WS-BPEL service compositions: A scenario-oriented approach, IEEE Transactions on Services Computing PP (2015) 1–1.

[2] M. H. Beek, A. Bucchiarone, S. Gnesi, Formal methods for service composition, Annals of Mathematics, Computing and Teleinformatics 1 (2007) 1–14.

[3] A. Basu, B. Bensalem, M. Bozga, J. Combaz, M. Jaber, T. H. Nguyen, J. Sifakis, Rigorous component-based system design using the bip framework, IEEE Software 28 (2011) 41–48.

[4] S. Bliudze, J. Sifakis, A notion of glue expressiveness for component-based systems, in: Proceedings of the 19th International Conference on Concurrency Theory, CONCUR '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 508–522.

[5] BIP tools, BIP tools, `http://www-verimag.imag.fr/BIP-Tools,93.html`, 2017.

[6] E. Stachtiari, A. Mentis, P. Katsaros, Rigorous analysis of service composability by embedding WS-BPEL into the bip component framework, in: proc. of the IEEE 19th International Conference on Web Services (ICWS), 2012, pp. 319–326.

[7] BPEL2BIP, BPEL process modelling tools, `http://depend.csd.auth.gr/research/BpelProcessModelling`, 2017.

[8] S. Jehan, I. Pill, F. Wotawa, Bpel integration testing, in: International Conference on Fundamental Approaches to Software Engineering, Springer, 2015, pp. 69–83.

[9] N. Ben Said, T. Abdellatif, S. Bensalem, M. Bozga, A robust framework for securing composed web services, in: Revised Selected Papers of the 12th International Conference on Formal Aspects of Component Software - Volume 9539, FACS 2015, Springer-Verlag New York, Inc., New York, NY, USA, 2016, pp. 105–122.

[10] S. Bensalem, M. Bozga, B. Delahaye, C. Jegourel, A. Legay, A. Nouri, Statistical model checking qos properties of systems with sbip, in: Proceedings of the 5th International Conference on Leveraging Applications of Formal Methods, Verification and Validation: Technologies for Mastering Change - Volume Part I, ISoLA'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 327–341.

[11] A. Basu, S. Bensalem, M. Bozga, P. Bourgos, M. Maheshwari, J. Sifakis, Component assemblies in the context of manycore, in: Formal Methods for Components and Objects, 10th International Symposium, FMCO 2011, Turin, Italy, October 3-5, 2011, Revised Selected Papers, 2011, pp. 314–333.

[12] P. Attie et al, A general framework for architecture composability, Formal Aspects of Computing 18 (2016) 207–231.

[13] A. Mavridou, E. Baranov, S. Bliudze, J. Sifakis, Architecture diagrams: A graphical language for architecture style specification, in: Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016., 2016, pp. 83–97.

[14] K. C. Wong, W. M. Wonham, Hierarchical control of discrete-event systems, Discrete Event Dynamic Systems 6 (1996) 241–273.

[15] R. J. van Glabbeek, W. P. Weijland, Branching time and abstraction in bisimulation semantics, J. ACM 43 (1996) 555–600.

[16] OASIS, Web services business process execution language version 2.0, 2007. URL: `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.

[17] N. Halbwachs, F. Lagnier, P. Raymond, Synchronous observers and the verification of reactive systems, in: Proceedings of the Third International Conference on Methodology and Software Technology: Algebraic Methodology and Software Technology, AMAST '93, Springer-Verlag, London, UK, UK, 1994, pp. 83–96.

[18] F. Montesi, M. Carbone, Programming services with correlation sets, in: Proceedings of the 9th International Conference on Service-Oriented Computing, ICSOC'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 125–141.

[19] C. Ouyang, E. Verbeek, W. M. van der Aalst, S. Breutel, M. Dumas, A. H. ter Hofstede, Formal semantics and analysis of control flow in WS-BPEL, Science of Computer Programming 67 (2007) 162 – 198.

[20] P. Parizek, J. Adamek, Checking session-oriented interactions between web services, in: Proceedings of the 2008 34th Euromicro Conference Software Engineering and Advanced Applications, SEAA '08, IEEE Computer Society, Washington, DC, USA, 2008, pp. 3–10.

[21] P. Hudak, Modular domain specific languages and tools, in: Proceedings. Fifth International Conference on Software Reuse, 1998, pp. 134–142.

[22] A. V. Aho, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[23] M. B. Juric, A Hands-on Introduction to BPEL, `http://www.oracle.com/technetwork/articles/matjaz-bpel1-090575.html`, 2017.

[24] M. B. Juric, A Hands-on Introduction to BPEL, Part 2: Advanced BPEL, `http://www.oracle.com/technetwork/articles/matjaz-bpel2-082861.html`, 2017.

[25] N. Lohmann, A feature-complete petri net semantics for WS-BPEL 2.0, in: Web Services and Formal Methods, 4th International Workshop, WS-FM 2007, Brisbane, Australia, September 28-29, 2007. Proceedings, 2007, pp. 77–91.

[26] N. Lohmann, E. Verbeek, C. Ouyang, C. Stahl, Comparing and evaluating petri net semantics for bpel, International Journal of Business Process Integration and Management 4 (2009) 60–73.

[27] W. M. Van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, E. Verbeek, Conformance checking of service behavior, ACM Transactions on Internet Technology (TOIT) 8 (2008) 13.

[28] I. Aït-Sadoune, Y. A. Ameur, Stepwise development of formal models for web services compositions: Modelling and property verification, Trans. Large-Scale Data- and Knowledge-Centered Systems 10 (2013) 1–33.

[29] B. Norton, S. Foster, A. Hughes, A compositional operational semantics for owl-s, in: Proceedings of the 2005 International Conference on European Performance Engineering, and Web Services and Formal Methods, International Conference on Formal Techniques for Computer Systems and Business Processes, EPEW'05/WS-FM'05, Springer, 2005, pp. 303–317.

[30] E. Fares, J. Bodeveix, M. Filali, Design of a BPEL verification tool, in: Web Services and Formal Methods - 8th International Workshop, WS-FM 2011, Clermont-Ferrand, France, September 1-2, 2011, Revised Selected Papers, 2011, pp. 95–110.

[31] B. Berthomieu, J.-P. Bodeveix, M. Filali, H. Garavel, F. Lang, F. Peres, R. Saad, J. Stoecker, F. Vernadat, P. Gaufillet, et al., The syntax and semantics of fiacre, Repport LAAS (2007).

[32] R. Lucchi, M. Mazzara, A pi-calculus based semantics for WS-BPEL, Journal of Logic and Algebraic Programming 70 (2007) 96–118.

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

[34] S. Tasharofi, M. Vakilian, R. Z. Moghaddam, M. Sirjani, Modeling web service interactions using the coordination language reo, in: Proceedings of the 4th International Conference on Web Services and Formal Methods, WS-FM'07, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 108–123.

[35] S. DeRose, J. Clark, XML Path Language (XPath) Version 1.0, W3C Recommendation, W3C, 1999. Http://www.w3.org/TR/1999/REC-xpath-19991116.

[36] J. Clark, XSL Transformations (XSLT) Version 1.0, W3C Recommendation, W3C, 1999. Http://www.w3.org/TR/1999/REC-xslt-19991116.

[37] S. Carey, SOA Best Practices: The BPEL Cookbook - Making BPEL Processes Dynamic, `http://www.oracle.com/technetwork/articles/carey-090553.html`, 2017.

[38] M. Bozga, M. Jaber, N. Maris, J. Sifakis, Modeling dynamic architectures using dy-bip, in: Proceedings of the 11th International Conference on Software Composition, SC'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 1–16.

Appendix A. Variables for process state

A.1. The state of service interactions

The lifetime of service interactions spans across the execution of multiple activity components. Therefore, we use variables in the `data` components of scopes that allow sharing information about the state of service interactions. These variables store:

- the (url) location of remote services of partner links that influences the routing of incoming messages. Partner links may or may not be initialized (by the `scope` or `copy` components) when they are accessed, in which case a fault is thrown.
- the correlation sets; these are sets of *correlation properties*, i.e. variables that are instantiated and accessed by activity components for service interactions (`receive`, `reply`, `invoke`, `listen`). A fault (correlation violation) is thrown upon attempts to (i) initialize already initialized sets, (ii) access uninitialized sets, (iii) send a message not matching the initialized sets.
- the information for routing messages to each listening `receive` component[2]; this information consists of a partner link, a service operation, as well as a list of correlation set values and their mappings to message parts.

  A fault is thrown if the routing information of `receive` components are conflicting (i.e., they match the same messages) or ambiguous (i.e., they can both match a message).
- a request identifier for each incomplete incoming synchronous request[3]; this identifier is set by the `receive` component and encodes its associated partner link, service operation and possibly some internal transaction identifier[4]; each `reply` component attempts to erase the request identifier to which it replies. A fault is thrown if: (i) a `receive` component attempts to set a duplicate request identifier (conflicting request), (ii) a `reply` does not find its associated request identifier (missing request), or (iii) a `scope` about to end detects an incomplete synchronous request (missing reply).

A.2. BPEL variables

The variables for a BPEL process may have been defined globally or within scopes. They store the content of messages or any other information that is shared among the activities of a scope, and are often used in conditions that influence the control flow. Their data types are either XML types or WSDL message types with partitions, called *parts*. In the BIP model, these variables are stored within the `data` components using separate BIP variables for each of their parts; they are read and assigned by activity components such as `copy`, `receive`, `listn`, `invoke`, `reply` and `loopctrl` using their `read` and `write` ports.

For the values of BPEL variables, we have adopted a data abstraction approach using symbols. This allows identifying variables that have not been initialized, and assignments with the same expression which is needed to detect every time that the variables change value. The same default symbol is assigned to all variables that have not been initialized. Each activity component with assignment semantics (e.g. `copy`, `receive`) evaluates the symbol to be assigned - let us call it right-value - to some BPEL variable, referred to as left-value. First, the symbols used in the right-value are being read:

- if they correspond to BPEL variables, then they are retrieved from data components;
- if they are BPEL message inputs or external data, then their values are represented by distinct symbols.

To the left-value is then assigned the hash code for the string given by concatenating the retrieved symbols with the static parts of the right-value.

[2] this refers to enabled IMA according to the BPEL terminology
[3] this refers to open IMA according to the BPEL terminology
[4] this refers to the BPEL `messageExchange` attribute

Appendix B. Compositional semantics for BIP compounds

B.1. BIP compound for `sequence`

Definition B.1. A `sequence` composite encloses n components $\texttt{act}_1 \ldots \texttt{act}_n$. The following safety properties have to be satisfied:

- if `sequence` is started, so does $\texttt{act}_1$.
- $\texttt{act}_i$ is started only if $\texttt{act}_{i-1}$ is finished.

The safety properties of Def. B.1 and the general properties are fulfilled by the *Sequential* style of Figure 18. The style has a coordinator `P` which enforces a sequential order of execution between two parameters `A1` and `A2`. Thus for the coordination of $n > 2$ parameters, *Sequential* style must be applied hierarchically n-1 times. The first and second property of Def. 4.1 are implemented by two rendezvous between the `P`.*start* and `A1`.*start* ports (resp. of `A1`.fin and `A2`.*start* ports). Note that *fin* ports are linked by connectors twice in `A1`: first, in order to synchronize the *fin* and *start* of successive components, and second, to synchronize the *fin* of both components at the end of their processing. The rendezvous between the `A1`.rvs and `A2`.$rvsd$ ports enforce the order of compensation, which is the reverse of the order of execution. In case `A1` *abort* ports, it can cause `A2` to be disabled by triggering one of the *abort* ports. This coordination, though it is not applied to `sequence`, it is needed for using the style to other components with similar semantics (e.g. the `act` composite).

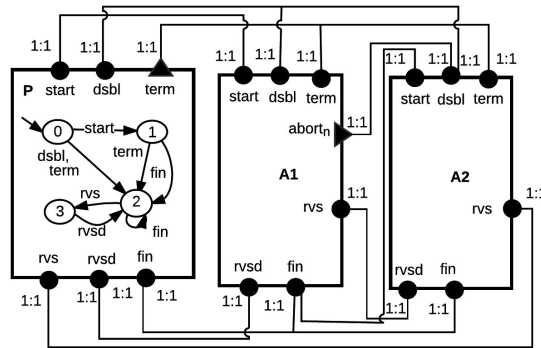


Figure 18: *Sequential* style

The connectors in Figure 18 that connect the $dsbl$, $term$ and fin ports are used to enforce the general safety properties in Section 4. These connectors exist also in the Prioritized Alternative, Non-Prioritized Alternative, Repetitive and Parallel Repetitive styles, though they are omitted from the diagrams. The connectors of rvs and $rvsd$ ports for these styles are placed as shown in Figure 8. In all these style there is one coordinating component which is branching bisimilar to the `P` component shown in Figure 8. This ensures branching bisimilarity between the observable behavior of the styles and the `empty` component.

B.2. BIP compound for `if`

Definition B.2. An `if` composite encloses one `condctrl` component and n components $\texttt{act}_1 \ldots \texttt{act}_n$. Let us consider the `condctrl` ports $acc_1 \ldots acc_n$ and $rej_1 \ldots rej_n$, for accepting an activity component (resp. rejecting it). The following safety properties have to be satisfied:

- $\texttt{act}_i$ is started only if it is accepted by the `condctrl`.
- if `condctrl` rejects an $\texttt{act}_i$, then $\texttt{act}_i$ is disabled.

The *Conditional alternative* style of Figure 19 is used to enforce the safety properties of Def. B.2. The style has one `CN` and n `A` parameters. The coordinator `P` plays the same role as in the previously described styles. The connectors that enforce the general safety properties are omitted from Figure 19. Parameters `A` are assumed to be branching bisimilar with `empty`, whereas `CN` is assumed to be branching bisimilar to the behavior with which it is depicted in Figure 19. The first property of Def. B.2 is fulfilled by n broadcast connectors between each `CN`.$selct_i$

and $\mathtt{A}_i$.*start* for $i = 1 \ldots n$. The second property is satisfied by a single broadcast connector between CN.rej and all $\mathtt{A}_i$.*disable*. Broadcast connectors were chosen, so that CN is not blocked after the termination of the `if` composite, upon which all $\mathtt{A}_i$ are terminated while CN doesn't enable termination. As a result, CN must be able to perform, for example, $selct_i$ even if $\mathtt{A}_i$ cannot start. Finally, it is worth to mention that the effects of disabling some $\mathtt{A}_i$ do not depend on race conditions, thus the same results are produced whether $\mathtt{A}_i$ is disabled before, during, or after the execution of some $\mathtt{A}_j$.

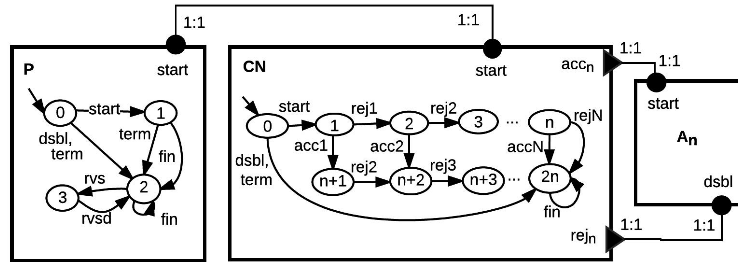


Figure 19: Conditional alternative style

B.3. BIP compound for `pick`

Definition B.3. A `pick` composite encloses n components $\mathtt{act}_1 \ldots \mathtt{act}_n$ and n components $\lambda_1 \ldots \lambda_n$ (each being either `listn` or `timer`). λ components export the ports ev and off, for receiving an event (i.e. message or timing event) and, respectively, stop waiting. The following safety properties need to be satisfied:

- if `pick` is started, so do all $\mathtt{L}_i$.
- $\mathtt{act}_i$ is started only if the event of $\mathtt{L}_i$ arrives.
- At most one $\mathtt{act}_i$ is started.

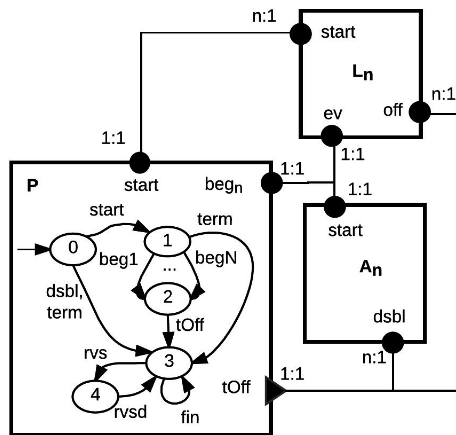


Figure 20: *Alternative* style

The *Alternative* style (Figure 20) is used to enforce the safety properties of Def. B.3. The style has n M parameters, n A parameters and a coordinator P. The behavior of L and A is assumed to be branching bisimilar to `empty`. The first property of B.3 is fulfilled by a single rendezvous that connects the P.*start* and the $\mathtt{L}_i$.*start* ports, while the second is realized by the rendezvous between the $\mathtt{L}_i$.*ev* and $\mathtt{A}_i$ ports.*start*. The third property is satisfied by the coordinator P and its rendezvous connection with the $\mathtt{L}_i$.*ev* port, which allows only one such port to be executed. Subsequently, P disables the A components and closes the L components through a broadcast connector triggered by the $tOff$ port. Note that the A that started will not be disabled, since it is not in the initial state. The broadcast connector allows the interaction to be executed, even if this A is not able to participate.

B.4. BIP compound for `loop`

Definition B.4. A `loop` composite encloses one `loopctrl` component and n components, $\texttt{act}_1 \ldots \texttt{act}_n$, standing for n repetitions of the loop body activity. Let us consider the `loopctrl`'s ports $beg_1 \ldots beg_n$, *end* and *break*, for beginning a loop's execution, receiving its end and breaking from loop, respectively. The following safety properties have to be satisfied:

- $\texttt{act}_i$ is started only if `loopctrl` begins the i–th loop execution.
- if *break* occurs, the $\texttt{act}_i$ of all remaining loop executions are disabled.

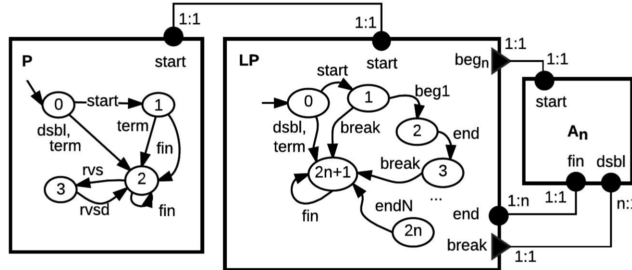


Figure 21: *Repetitive* style

Note that we use a different activity component to maintain each repetition's state, since the loop's activity compensation will have to run for each repetition separately.

The *Repetitive* style (Figure 21) is used to enforce the safety properties of Def. B.4. The style contains one parameter `LP`, n parameters `A` and a coordinator `P`. The first property is satisfied by the broadcast from `LP`.beg_i to the $\texttt{A}_i$.*start* port, whereas the second property is realized by the broadcast from `LP`.*break* to all $\texttt{A}_i$.*dsbl* ports.

Definition B.5. A `parallel loop` composite encloses one `loopctrl` component and n components $\texttt{scope}_1 \ldots \texttt{scope}_n$, standing for n parallel repetitions of the loop body `scope`. Let us consider the `loopctrl`'s ports $beg_1 \ldots beg_n$, for beginning i parallel loop executions, and *break* for breaking from loop, respectively. Also, the ports $fail$ and *succ* used by `loopctrl` to receive the successful (resp. unsuccesssful) completion of every `scope`. The following safety properties have to be satisfied:

- if `loopctrl` begins i loop executions, $\texttt{scope}_1 \ldots \texttt{scope}_i$ are started.
- if `loopctrl` breaks from loop, all terminatable $\texttt{scope}_i$ are terminated.
- if $\texttt{scope}_i$ completes, `loopctrl` is notified whether completion was successful or not.

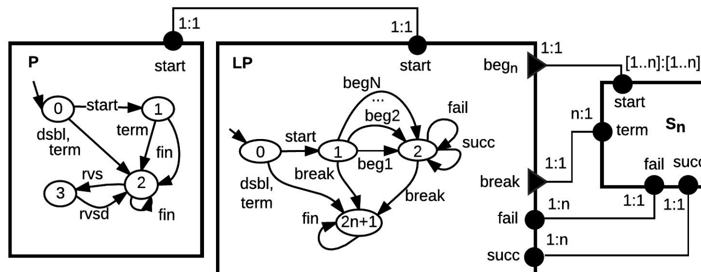


Figure 22: *Parallel Repetitive* style

The *Parallel repetitive* style (Figure 22) is used to enforce the safety properties of Def. B.5. The style contains one parameter `LP`, n parameters `S` and a coordinator `P`. The first property is satisfied by the broadcast from `LP`.beg_i to the *start* ports of $\texttt{S}_1 \ldots \texttt{S}_i$. Number i is evaluated by `loopctrl`. The second property is fulfilled by the broadcast from `LP`.*break*, which is executed by `loopctrl` in 3 cases: (a) if $i = 0$ in state 1 (b) if the number of completed `S` in state 2 is the minimum needed for the loop and (c) if all `S` in state 2 are completed. It depends on the `loopctrl` whether it

counts all completed S to to the minimum needed, or only those that finished successfully. Finally, the third property is fulfilled by the rendezvous connectors between each $\mathtt{S}_i.succ$ and $\mathtt{LP}.succ$ (resp. $fail$)

B.5. BIP compound for `act`

Definition B.6. An `act` composite is used to enclose a specific activity component, say `acti`, one `target` and/or one `source` component. Let us consider the `target`'s port *abort*, for preventing the execution of `actc` due to target links, and the `source`'s ports: $read_1 \ldots read_m$, for reading values used in the evaluation of m source links (also evaluating the links), and $set_1 \ldots set_m$, for setting the evaluation' results to the m links. The following safety properties have to be satisfied:

- `actc` is started only if (`target` has finished) and (`target` has not prevented `acti`).
- `source` is started only if `acti` has finished.
- `source` evaluates a link only if (`acti` has not been disabled or terminated) and (neither `acti` or `source` has thrown a fault).

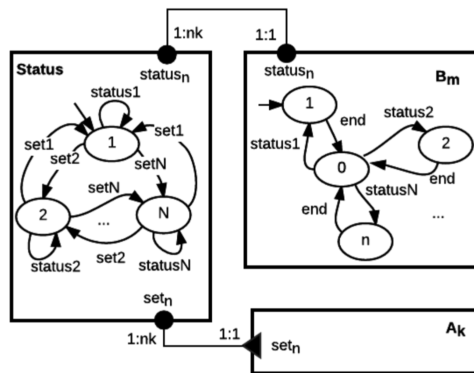


Figure 23: *Status mngmt* style

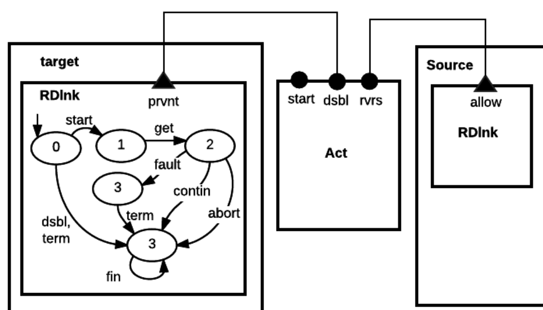


Figure 24: Reading of links in `act` component

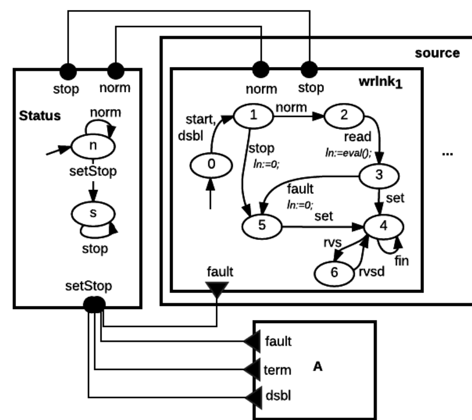


Figure 25: Writing of links in `act` component

For the properties of Def. B.6 two architecture styles were combined, namely the *Sequential* and the *Status mngmt* styles. The *Sequential* style is used for the first two properties of Def. B.6, that are ensured through the sequential execution of `target`, `acti` and `source` and the ability of `target` to abort the sequence. *Status mngmt* (Figure 23) is used to enforce the third property of Def. B.6. The style has m parameters `B`, and k parameters `A`, which get and set the status, respectively. A coordinator `Status` is used to maintain n status values using a different state for each

status value. There are no assumptions for A, while B are assumed to be branching bisimilar to the behavior shown in Figure 23. According to the third property of Def. B.6 there will be two statuses: the *norm*, which is set by default, and the *stop* which is set when acti is disabled, or terminated, or if some component (acti or source) has thrown a fault. For applying the style, source will be used as operand for B, whereas both source and acti will be operands for A. Figure 24 shows the result of applying the two sequential architectures, whereas Figure 25 shows the result of applying the Status mngmt in act composite.

B.6. Architectures of the components in PROC and scope

The scope encloses composite components with applied architectures that coordinate their constituent components. We describe the architectures used in each of the norm, evhlrs, faulthlrs and comphlr, though we omit the expected safety properties and implementation details.

The norm composite encloses a main activity (act) and a component that contains event handlers (evhlrs). Two coordinating components are attached, namely the scinit and the scfin, that perform initialization and finalization actions at the beginning (resp. at the end). of norm. For example, scfin checks whether open IMAs are left after act and evhlrs have finished. The *Sequential* style is applied, so that scinit and act are executed sequentially, whereas the *Parallel* style is applied to the *Sequential*'s result and the evhlrs. Finally, the *Sequential* style is applied again on the the *Parallel*'s result and scfin. The scinit and scfin are weakly bisimilar to empty, thus they are valid operands for these styles. One extra connector is used, that turns off event handlers (evhlrs.$turnOff$ port), so that no new events are handled after act has finished (act.fin). Note that if norm is inside PROC it will contain the process' starting activity, which in this case must first finish before evhlrs is started. This enables evhlrs to use input received by the starting activity (e.g. for the definition of expected events).

The evhlrs composite encloses a single empty component handles in the trivial case where there are no expected events to handle. In all other cases, it encloses n event receiving components (i.e. a timer or listn) and $m = 2 \times n$ evscope components that handle the incoming events. Each event receiving component is associated with two evscope components that will handle one event occurrence each. For our verification purposes, it is sufficient to consider just two occurrences of each event, thus we use only two evscopes per event. Two evscopes can materialize all the interleavings that are necessary to capture concurrency issues due to parallel handling of the same event. All event receivers are started when evhlrs is started (i.e. *Parallel* style) and they start one of their associated evscopes upon receiving an event. Event receivers can receive subsequent events, until evhlrs is turned off (i.e. all the non-started evscopes are then disabled). Afterwards, evhlrs is considered finished when all evscopes are finished. The evscopes can be concurrently compensated. The behavior of evscope is quite similar to scope, since they share the same structure and connectors. Their difference lies in that evscope treats the event receiver as if it were an enclosed component even though it is not. According to that, evscope will handle any fault thrown by the event receiver and allows the event receiver to write to variables and access the correlation sets of the evscope.

The faulthlrs composite encloses n catch and their associated n act components. Also, a scfin component is attached, as the one used in norm. The *Alternative* style is applied so that only one catch can trigger its associated act when faulthlrs is started. Specifically, upon faulthrs.*startFH* a rendezvous of all catch.*start* ports is invoked in which each catch exports a string that characterizes the faults that it can handle. The rendezvous connector performs a computation which decides on a catch that triggers its act (i.e. all other catch and act are then disabled). The fault occurrence is stored as a local variable in catch, so that it can be thrown again by faulthlrs if a rethrow component is executed within act. The *Sequential* style is applied so that scfin is started after the result of *Alternative* style is finished.

The comphlr composite encloses an act and a scfin coordinated by the *Sequential* style. The termhlr composite encloses a single act, thus it has no coordination needs.

Appendix C. Models for basic activities and other atomic components

The models for basic activities and the other atomic components of Table 2 are presented here. Note that in the components' figures we omit to include all the `term` ports, in order to keep them uncluttered.

invoke

The `invoke` component performs a service invocation and it has two variants based on whether it invokes a one-way or a request-response operation. Figure 26 shows the ports included in both variants with solid line and the ports that are specific to each variant with dashed and dotted lines, respectively. The *intern* port is used for the component's internal transitions. The used ports are:

- *read*, to read the partner link and the variables (and correlation sets), for preparing the message.
- *snd_msg*, to send the invocation message;
- *rcv_msg*, when the invocation's response is received
- *write*, to store the message and the retrieved correlation sets.

The component throws possible *correlation violation* (cs_viol), *uninitialized partner role* (unin_role), *uninitialized variable* (unin_var) and *selection failure* (slct_fail) faults during the preparation of the request message. From these faults, the first three can be detected, while the last one is non-deterministically thrown. For the response message, the component throws also the *mismatched assignment* (mis_assg) fault non-deterministically. Also, if the received response corresponds to a fault message (flt_msg) then this fault is thrown by the `invoke`.

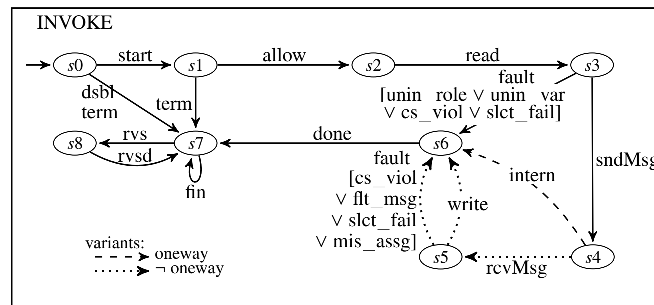


Figure 26: The `invoke` component with two variants.

reply

The `reply` component (Figure 27) handles the response of a synchronous operation.
The used ports are:

- *chk_ima*, to check that such a request is waiting for response.
- *read*, to read variables and correlation sets for preparing the message
- *close_ima*, to remove request from the waiting list.
- *snd_msg*, to send the response message;

The component detects and throws possible *correlation violation* (cs_viol) and *uninitialized variable* (unin_var) faults. Also, it throws non-deterministically the *mismatched assignment* (mis_assg), *selection failure* (slct_fail), and *missing request* (miss_req) faults.

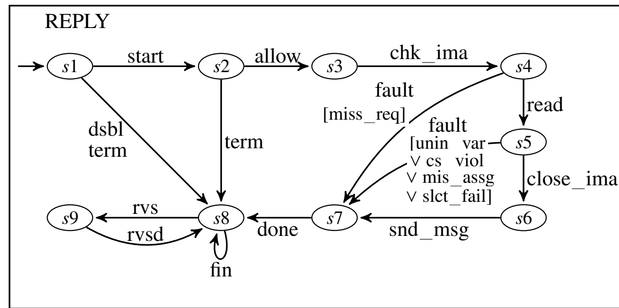


Figure 27: The `reply` component.

listn

The `listn` component is ommited, since it is similar to the `receive`. Namely, `listn` is different in that: (i) it can receive multiple messages with one open listening endpoint, and (ii) it can be turned off, when it shouldn't receive any more messages.

copy

The `copy` component (Figure 28) handles the assignment to a BPEL variable (or a partner link). The component reads the source variable stored in some `data` component (*read* port) and evaluates its value. If no fault occurs, it assigns the value to the target variable, found in some `data` component (*write* port). Note that if the `copy` assigns from a literal value, the *read* port has no effect.

The component detects and throws possible *uninitialized variables* (unin_var), and *uninitialized partner role* (unin_role) faults. Also, it throws the *mismatched assignment* (mis_assg), *selection failure* (slct_fail), and *invalid selection* (inval_slct) faults non-deterministically. These faults may or may not be thrown based on the source and target variables' variant, which can be a variable, a partner link, a literal value or an expression.

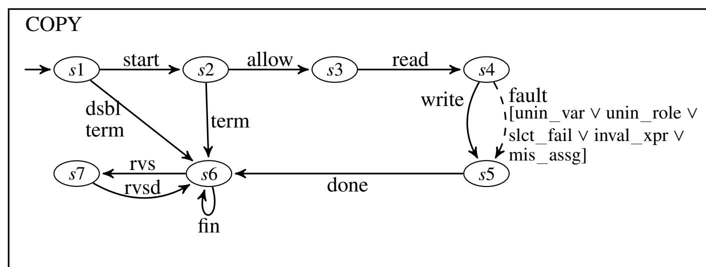


Figure 28: The `copy` component with two variants.

compensate

The `compensate` component (Figure 29) controls the compensation of one or many scopes. Such a component reside only within the `act` of the components `faulthlrs`, `comphlr` and `termhlr`. The used ports are:

- *compensate*, to start the compensation of some scope(s).
- *infault*, to receive a fault from the compensated scopes.
- *endrvrs*, to be notified about the end of compensation.
- *termcomp*, to abruptly terminate the compensation.

The component rethrows, through the *fault* port, any fault of the compensated scope. Thus, these faults can be handled by the `compensate`'s enclosing scope.

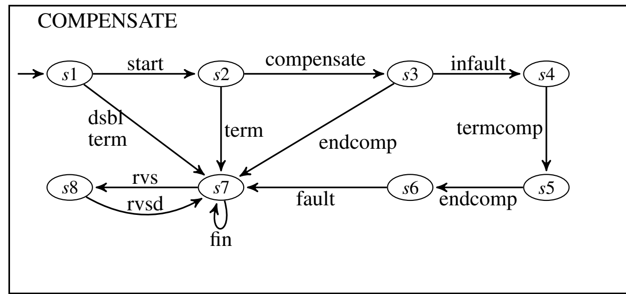


Figure 29: The `compensate` component.

throw , rethrow

The `throw` component (Figure 30) is used to throw an explicit internal fault, through the *fault* port. On the other hand, the `rethrow` component (Figure 31) has a slightly different role; It is placed within `faulthlrs` in order to to rethrow the fault that was originally caught. For this reason, the `rethrow` component uses the *rethrow* port that has different semantics from the *throw* port and it is handled differently by the `scope`'s glue: initially the transferred fault is unknown and it is assigned at the scope level, after it is retrieved by the scope's controller which stores the caught faults.

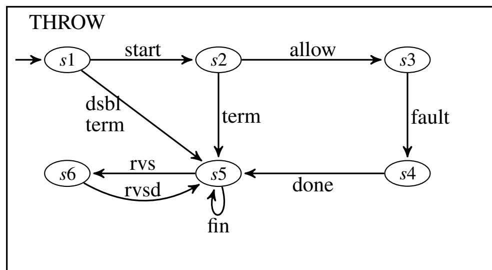


Figure 30: The `throw` component.

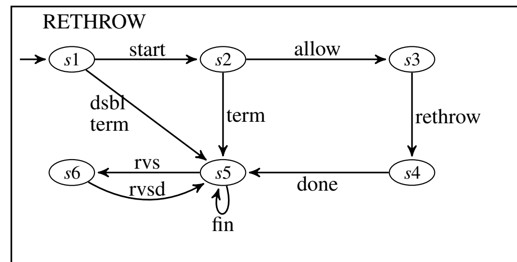


Figure 31: The `rethrow` component.

exit

The `exit` component (Figure 32) fires the *exit* port, that causes the whole process's interruption.

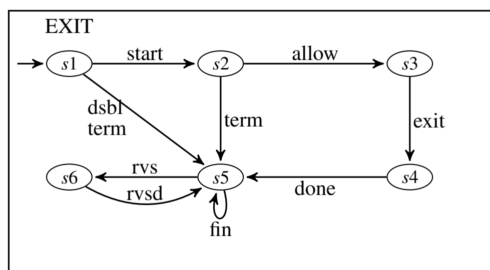


Figure 32: The `exit` component.

valid

The `valid` component (Figure 33) validates a variable against its data type (XML schema). Since our modelling assumes only symbolic values for variables, the component cannot detect the *invalid variables* (inval_var) fault, thus the fault is thrown non-deterministically.

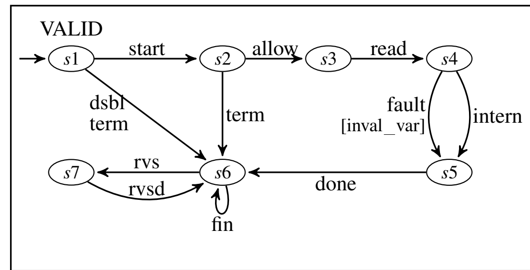


Figure 33: The `valid` component.

timer

The `timer` component (Figure 34) is used to model the BPEL activities *onAlarm* and *wait* that handle the firing of a time-out. The time-out is specified using either a duration (relative timestamp) or a date (absolute timestamp). Also, the time-out for *onAlarm* may be periodic, if a time period is given. All the `timer` components of the model are synchronized (*tick* port) so that they update their remaining time accordingly: upon each *tick*, one `timer` expires (whichever is closer to expire), while this `timer's` remaining time is reduced by the other `timers` remaining times. The used ports are:

- *read*, to read the variables for evaluating the timer's expressions.
- *tick*, to synchronize with other `timers`.
- *trigger*, when there is a time-out.
- *tOff*, when it is turned off.

The component can detect a possible *uninitialized variable* (unin_var) fault, while it throws the *invalid expression value* (inval_xpr) fault non-deterministically.

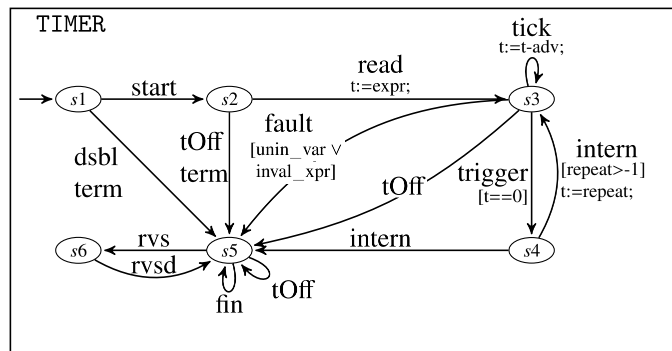


Figure 34: The `timer` component.

loopctrl

The `loopctrl` component for the *while* and *repeatUntil* activities are shown in Figure 35 and Figure 36 respectively. Both components have the same interface, though their behaviors differ, based on whether the first execution depends on a decision or not. The used ports are:

- *read*, to read the variables for evaluating the conditions.
- *trigger*, to start the loop body activity;
- *done_in*, when the loop body activity is finished.
- *tOff*, to exit the loop.

The component can detect a possible *uninitialized variable* (unin_var) fault, while it throws the *invalid expression value* (inval_xpr) fault non-deterministically.

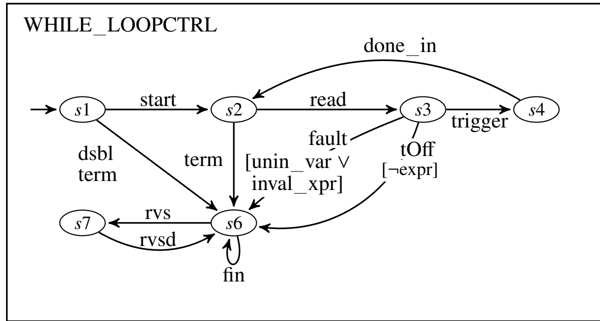


Figure 35: The `loopctrl` component for the `while` activity.

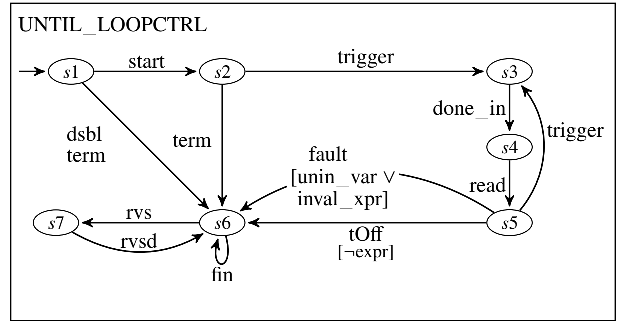


Figure 36: The `loopctrl` component for the `repeatUntil` activity.

The `loopctrl` for the *forEach* and the parallel *forEach* activities are shown in Figure 37 and Figure 38 respectively. The components have the same interface, though their behaviors differ slightly. The used ports are:

- *read*, to read the variables for evaluating the expressions.
- *trigger*, to start the loop body scope;
- *succ*, when the loop body scope is successfully finished.
- *fail*, when the loop body scope is finished but not successfully.
- *tOff*, to exit the loop.

The component can detect a possible *uninitialized variable* (unin_var) fault, while it throws the *invalid expression value* (inval_xpr) fault non-deterministically.

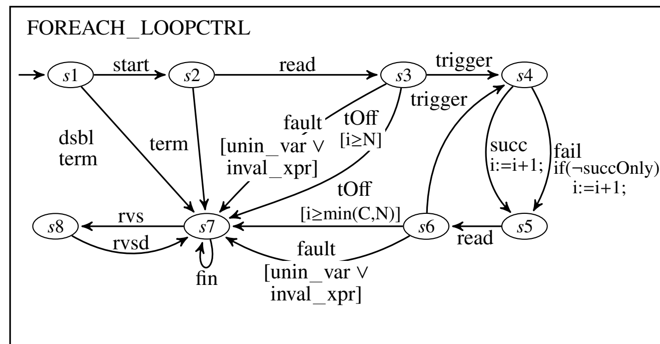


Figure 37: The `loopctrl` component for the `forEach` activity.

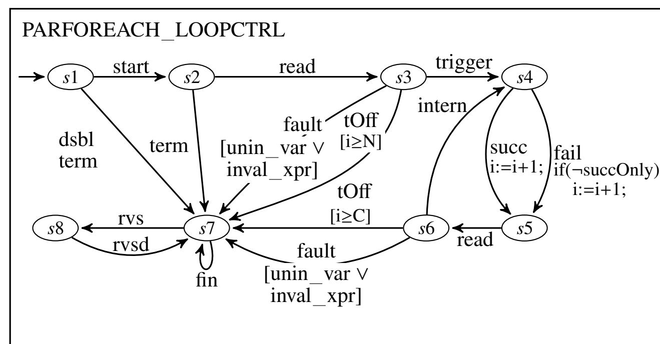


Figure 38: The `loopctrl` component for the parallel `forEach` activity.

condctrl

The `condctrl` component (Figure 39) handles a decision for the execution of one out of N components. The used ports are:

- *read*$_i$, to read the variables for evaluating expression i.
- *trigger*$_i$, to start component i;
- *tOff*$_i$, to disable component i.

The component can detect a possible *uninitialized variable* (unin_var) fault, while it throws the *invalid expression value* (inval_xpr) fault non-deterministically.

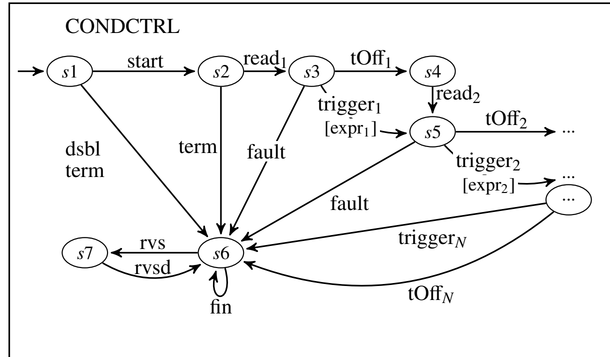


Figure 39: The `condctrl` component.

Appendix D. Translation times for test programs

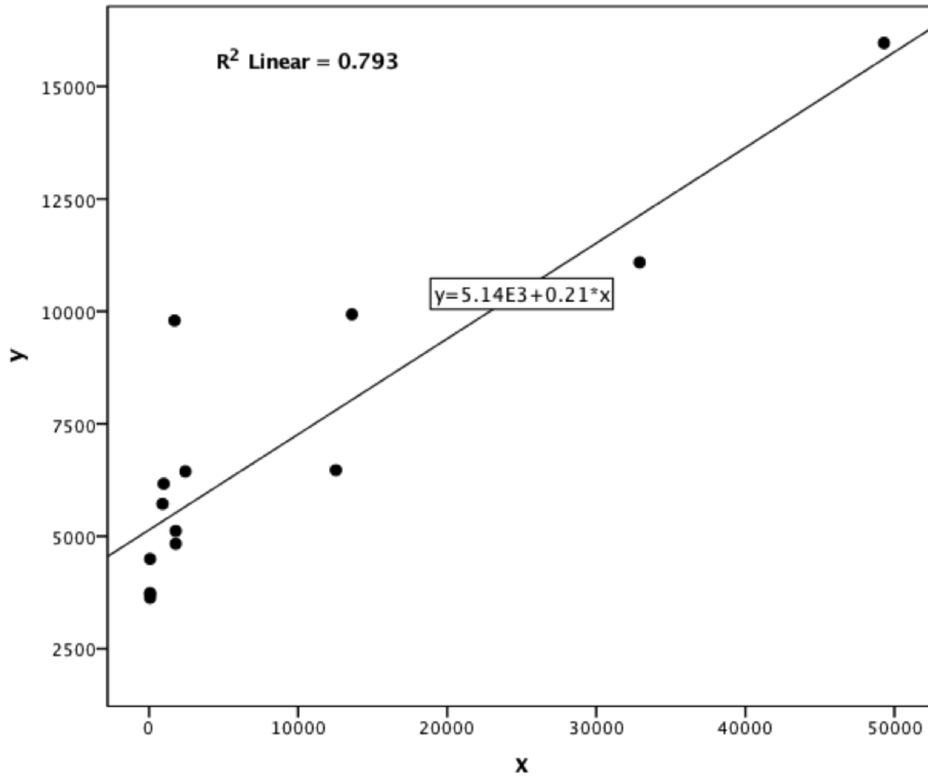


Figure 40: Regression analysis of translation times for test programs
(*x* axis is the number of states, *y* axis in *ms*)