Testing and Verification of Service Compositions

by

Adel Khaled

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

In

Software Engineering and Intelligent Systems

Department of Electrical and Computer Engineering

University of Alberta

Abstract

Service Oriented Architecture (SOA) delivers on the promise of dynamic applications that are better aligned with business goals. Building SOA systems, whether choreography or orchestrated, involve the interaction and collaboration of different parties through service calls. SOA architectural paradigm introduces a set of challenges that make SOA system verification different from conventional software systems. In this research, we explore the opportunities in the area of testing and verification of SOA systems built on top of the two main standards 1) *Web Service Choreography Description Language* (WS-CDL) and 2) *Web Service Business Process Execution Language* (WS-BPEL). In this regard, we introduce a formal language based on pi-calculus, Chor-calculus, for the formal modeling and verification of WS-CDL programs. This approach enables the static verification of choreographies using existing pi-calculus model-checker tools and sets the ground for enabling the runtime monitoring of choreographies for behavioral correctness. We validate the calculus for its expressiveness by evaluating the language support for representing workflow, and service interaction, patterns. We demonstrate the use of the HAL toolkit to verify the correctness properties of choreographies. On the other hand, we introduce a set of mutation operators for WS-BPEL and use mutation testing to verify the behavioral correctness of WS-BPEL programs. The fault models can be employed in mutation testing to assess the effectiveness of test suites or measure the accuracy of runtime monitors that are capable of verifying the correctness. We introduce a language to capture the trace behavior of BPEL programs and describe the fault models. We also propose a runtime verification system that consumes the specification language and detects temporal faults.

Preface

Chapter 2 of this thesis has been published as A. Khaled and J. Miller, “Using π-calculus for Formal Modeling and Verification of WS-CDL Choreographies”, IEEE Transactions on Service Computing, 2015. Most of the work in Chapter 3 has been published as A. Khaled, J. Miller, "Fault-Based Mutation Processes for WS-BPEL 2.0 Programs", International Journal Web Engineering and Technology, Vol. 6, No.2, pp. 141 – 170, 2010.

Table of Contents

List of Tables

List of Figures

1 Introduction

Service-Oriented computing encompasses many things, including design paradigms and principles, design patterns, distinctive architectural models and related concepts, technologies and frameworks [43]. Service-Oriented computing (SOC) utilizes services as fundamental elements for developing applications [68]. Service-Oriented Architecture is an architectural paradigm that promotes building applications from existing services [8]. SOA aims to enhance the agility and cost-effectiveness of an enterprise while reducing the burden of IT on the overall organization [43]. A service is the unit of solution logic upon which SOA systems are composed. Service composition is the aggregation of one or more services to automate business processes. Web services are the most widely used form of services.

SOA systems involve multiple parties collaborating to deliver a higher-level business process. Each party contributes to the process by exposing one or more consumable services that encapsulate some reusable business logic. Generally speaking, an SOA program can be described using one of two modeling styles:

- **A Global Perspective** or commonly known as *Choreography* describes the collaboration of the participants from a global perspective. This description is not biased towards any of the parties and is less centralized. Each participant will have the same view of the process; the Web Service Collaboration Description Language (WS-CDL), a W3C standard, is a specification to describe choreographies [4].

- **A Private Perspective** or commonly known as *Orchestration* describes the collaboration of the participants from the perspective of the service provider. This description is specific to the participant and is different from that of other participants in the same process. The Web Services Business Process Execution Language (WS-BPEL), a W3C standard, defines a model and grammar for capturing the specifications of a business process from a single perspective. WS-BPEL is the technology of choice for describing SOA orchestration [3].

The success of any software system depends on its functional correctness and reliability. The importance of software testing and quality delivery has been continuously increasing over the past years and SOA systems are no exception. However, SOA systems are composed of system components, services that are owned and controlled by third parties. Researchers are continuously working to devise new testing techniques to fit within this architectural model. Testing techniques for SOA systems are either: applied during the software development phase, where the software system is verified against specifications; or are monitored at runtime after its release to detect any deviations in the system's behavior from expectations. In this research, we focus on testing techniques that are applied during the software development phase.

This thesis is organized into two main sections. Section 2 introduces a formal language, Chor-calculus, for the formal modeling and verification of WS-CDL choreographies. Section 3 discusses a new set of mutation operators for WS-BPEL programs and how to leverage the operators in mutation testing. Section 4 provides a conclusion.

2 Using π-calculus for Formal Modeling and Verification of WS-CDL Choreographies

Service-Oriented applications are realized by composing and aggregating existing web services. Orchestration and Choreography are two interaction models for building SOA applications and several standards exist to capture and describe such interactions. The Web Service Choreography Description Language (WS-CDL) is a standard for modeling choreographies. In this paper, we propose a calculus (Chor-calculus) for formal modeling of WS-CDL and we use this language for generating WS-CDL programs. This approach enables the static verification of choreographies using existing pi-calculus model-checker tools and sets the ground for enabling the runtime monitoring of choreographies for behavioral correctness. We validate the calculus for its expressiveness by evaluating the language support for representing workflow, and service interaction, patterns. We demonstrate the use of the HAL toolkit to verify the correctness properties of choreographies.

2.1 Introduction

Service-Oriented Architecture is an architectural paradigm that promotes building applications from existing services [1]. SOA aims to enhance the agility and cost-effectiveness of an enterprise while reducing the burden of IT on the overall organization [2]. A service is the unit of solution logic upon which SOA systems are composed. Service composition is the aggregation of one or more services to automate business processes. Web services are the most widely used form of services. A web service has a well-defined contract that describes the format of messages and the message exchange patterns supported by the service.

SOA systems involve multiple parties collaborating to deliver a higher-level business process. Each party contributes to the process by exposing one or more consumable services that encapsulate some reusable business logic. Generally speaking, an SOA program can be described from two perspectives:

1. A Private Perspective, commonly known as Orchestration, describes the collaboration of the participants from the perspective of a single participant. This description is specific to the participant and is different from that of another participant of the same process. The Web Services Business Process Execution Language (WS-BPEL), a W3C standard, defines a model and grammar for capturing the specifications of a business process from a single perspective. WS-BPEL is the technology of choice for describing SOA orchestrations [3].

2. A Global Perspective, commonly known as Choreography, describes the collaboration of the participants from a global perspective. This description is not biased towards any of the parties and is less centralized. Each participant will have the same view of the process; the Web Service Collaboration Description Language (WS-CDL), a W3C standard, is a language to describe choreographies [4].

While formal modeling and reasoning about orchestrated systems is given attention in the literature [5], [6], [7], [8], [9], [10], [11], [12], little has been dedicated towards formalizing choreographies. In this paper, we focus on choreographies and specifically

the WS-CDL standard. One major issue of WS-CDL is the lack of tools to enable design time, static validation and verification of the correctness properties of choreographies such as deadlocks and liveness [13]. Although WS-CDL seems to borrow terminology from pi-calculus such as channel passing (link passing), there is no obvious association to any existing formal language [13], [14]. To tackle the aforementioned issues, we propose a formal method for capturing, as far as possible, the intention of the choreography. The formal language, which we call Chor-calculus, is based on pi-calculus. Our selection of pi-calculus is because of the language's simplicity and expressiveness to capture link mobility (or channels in WS-CDL) and its ability to describe the runtime behavior of concurrent systems (analogous to WS-CDL). We design and describe Chor-calculus based on the semantics of WS-CDL. To evaluate the expressiveness and completeness of the calculus, we analyze the support of the language to express both workflow [15], and service interaction, patterns [16]. Chor-calculus enables us to reason about the correctness properties of the choreography; and we provide a demonstration, using the HAL toolkit (HD-Automata Laboratory) [17], [18], of this reasoning. With the use of Chor-calculus, the generated WS-CDL code from a formally verified model detects design flaws as early as possible in the software development life cycle; and thus, increases the reliability of SOA systems built on top of the code.

The contributions of this work are three-fold:

1. We introduce Chor-calculus, a new formal language based on pi-calculus and the semantics of WS-CDL.

2. A mapping between Chor-calculus and WS-CDL is provided which allows the generation of WS-CDL programs from a Chor-calculus specification.

3. We demonstrate the use of the HAL toolkit to assert the correctness properties of choreographies described in Chor-calculus.

The rest of the paper is organized as follows: the next section briefly describes WS-CDL. In section 2.3, we present Chor-calculus, a formal language for modeling WS-CDL choreographies. Section 2.4 describes the mapping between Chor-calculus and WS-CDL. In section 2.5, we provide an evaluation of Chor-calculus. Section 2.6 introduces an

illustrative example to demonstrate how to use Chor-calculus for choreography modeling and WS-CDL code generation. Finally, sections 2.7 and 2.8 provide an overview of related literature in the field and a conclusion.

2.2 WS-CDL Overview

WS-CDL is an XML based specification that describes from a global viewpoint the common and observable collaboration (behavior) of participants in the form of a message exchange [4]. We can think of a behavior in the context of the web service platform as an operation on a portType (WSDL 1.1) [19] or interface (WSDL 2.0) [20]. A role type (roleType element) in WS-CDL enumerates a set of behaviors. A participant (participantType element) in a choreography adopts one or more roles, and thus a participant is committed to exhibit all the behaviors identified by the adopted roles. A relationship (relationshipType element) associates two roles to indicate a legal collaboration between participants. A channel (channelType element) realizes a point of collaboration between two participants by specifying when and how information is exchanged [4]. Channel instances can further be exchanged between participants during interactions to enable dynamic interaction scenarios (mobile links). Channels impose restrictions on the type of communication that can happen through the channel such as the role/behavior of the collaborating participant. Furthermore, they are used to identify and correlate conversations.

WS-CDL introduces three categories of activities: control-flow, WorkUnit and basic activities [13]. Under control-flow, there are three types of activities: *Sequence*, *Parallel* and *Choice*. These activities are composite activities in the sense they enclose other activities and govern their order of execution. A *Sequence* activity encapsulates one or more activities executed in a sequence. A *Parallel* activity encapsulates one or more activities executed concurrently. A *Choice* activity describes the execution of one activity among a set of activities. A *WorkUnit* describes the conditional and possibly repeated execution of the enclosed activity. The following activities fall under the third category of activities (basic activities): *NoAction*, *Silent*, *Interaction*, *Assign* and *Perform*. *NoAction* and *Silent* activities describes a point in the choreography where one role performs an action behind the scenes that doesn't affect the choreography. The *Perform*

activity is used to call another choreography to be executed within the context of the executing choreography. The *Interaction* activity is the point in the choreography where information exchange happens between the participants. This can be thought of as a service invocation that can fall under one of three message exchange patterns: Request-Only, Respond-Only, or Request-Response. The *Assign* activity is used to assign one variable to another within a role. The *Assign* activity does not support assigning variables that belong to different roles. Variable assignment across roles requires an explicit interaction between participants with the designated roles. This is referred to as interaction-based information alignment.

A WS-CDL definition is included in a *Package* construct that contains zero or more choreography definitions. A choreography definition is contained in the *Choreography* construct, and is the mechanism to define activities and variables. It is also a unit of reusability within WS-CDL. A *Choreography* definition introduces a new scope with its own variable definitions. Choreographies can be nested by calling other Choreographies. The calling Choreography is referred to as the *Enclosing Choreography* while the called choreography is called the *Enclosed Choreography*. There are two modes for calling a choreography: blocking and non-blocking. In the blocking mode, the enclosing choreography waits for the enclosed choreography to complete before continuing its execution. In the non-blocking mode, the enclosing choreography doesn't wait for the enclosed choreography and might finish before the completion of the enclosed choreography.

2.3 Chor-calculus

In this section, we introduce and define the syntax and semantics for Chor-calculus. The main objectives of the calculus are 1) to capture the activities, control flow and message exchange of WS-CDL choreographies, and 2) to allow formal reasoning about choreographies. The syntax and operational semantics for Chor-calculus are presented in sections 2.3.1 and 2.3.2 respectively.

2.3.1 Syntax

In this section, we provide an informal description of the calculus syntax shown in Table 1. We assume that there exists an infinite set of names ranged over by lower case letters x, y, z, etc… Processes are ranged over by *M, N, P, Q,* etc…

- $P \mid P'$ indicates the normal composition of two processes (P and P' run in parallel).
- $P \parallel P'$ indicates a sequential composition of two processes. We borrow the notation from [5] but without the support for synchronization.
- $!P$ is replication that behaves as $P \mid !P$. A non-guarded replication is used to model a root choreography which is initialized by default [4]. On the other hand, a guarded P is used to model a choreography that can be performed multiple times using a WS-CDL *perform* activity. The guard signifies the name of the choreography and the names passed represent performed choreography free variable bindings.
- $(\upsilon\ \tilde{x})P$ binds the names $\tilde{x}$ to process P.
- If-then-else is the usual choice selection based on the condition matching result.
- $G + G'$ is a guarded choice intended to model a WS-CDL choice activity.
- The input action prefixes are: $x(\tilde{y}).P$ is the usual input and τ is a silent action and represents a non-observable behavior. $\overline{x^t}\langle\tilde{y}\rangle$ where $t \in \{rq, rs, rqrs, pr, fn\}$ is an annotated output [5]. An output with one of the annotations rq, rs or rqrs is used to indicate the type of interaction pattern (through a WS-CDL interaction activity) between two services where rq, rs and rqrs correspond to a request, a response and a request-response respectively. The annotated output pr is used to model a WS-CDL perform activity and fn to model a WS-CDL finalize activity. $\overline{\mathrm{x}}\langle\,\rangle$ is used for signaling.
- $(\upsilon\ \tilde{x})\{P, NR, H\}$ models a choreography in WS-CDL which encapsulates a set of interactions between different participants. We borrow the notation from [5]

where it is used to model scope in WS-BPEL. *P* represents the main activity of the choreography and is provided by the choreography designer while *NR* represents zero or more non-root (nested) choreographies. On the other hand, *H* is the handlers' context for the choreography and is intended to model choreography exception and finalizer blocks. The syntax and semantics for the choreography is further discussed in section 2.3.1.3.

- $W_E(\hat{P})$ and $W_F(\hat{P})$ are intended to model exception and finalizer blocks respectively. The formal definition for both $W_E(\hat{P})$ and $W_F(\hat{P})$ are provided in sections 2.3.1.1 and 2.3.1.2 respectively.

- R and NR model root and non-root choreographies respectively. This distinction is necessary since there are some restrictions on the capabilities and the usage of root choreographies. The annotated input channel z^c in NR, which is semantically equivalent to the name of the choreography, is used to perform a non-root choreography. The tuple $\tilde{y}$ represents the variables expected by the performed choreography.

- A workunit *WU* expressed as $(\upsilon\ \tilde{x})\{P\}$ encapsulates the behavior of the complex workunit WS-CDL activity, where *P* represents the main activity of the workunit. The syntax and semantics for the workunit is further elaborated in section 2.3.1.4.

It is worth noting the design choice of introducing the sequential operator (||) given that its semantics overlaps with that of an action prefix. The sequential operator is mainly used to explicitly describe the sequencing of two processes, which is not possible with an action prefix, and to simplify the translation of Chor-calculus to WS-CDL (specifically the mapping of || to a sequence activity).

Expression			Name
P	::=	$(P \mid P')$	Parallel
	|	$(P \Vert P')$	Sequential
	|	G	Guarded input
	|	$!P$	Replication
	|	$(\upsilon\, \tilde{x})P$	Restriction
	|	$if\ (x = y)\ then\ P\ else\ Q$	Conditional
	|	WU	Workunit progress
G	::=	$0 \mid \pi \mid G + G'$	Guarded choice
π	::=	$\tau.P \mid x(\tilde{y}).P \mid x^t(\tilde{y}).P$	Action prefix
C	::=	$(\upsilon\, \tilde{x})\{P, NR, H\}$	Choreography
H	::=	$0 \mid W_E(\hat{P}) \mid W_F(\hat{P}) \mid \left(W_E(\hat{P}) \mid W_F(\hat{P})\right)$	Choreography handlers
R	::=	$!C$	Root choreography
NR	::=	$0 \mid !z^c(\tilde{y}).C \mid (NR \mid NR')$	Non-Root choreography
WU	::=	$(\upsilon\, \tilde{x})\{P\}$	Workunit
E	::=	$R \mid NR \mid (R \mid NR)$	Choreography system

Table 1: Chor-calculus Syntax

In the following sub-sections, we provide a formal definition for syntactic elements C, $W_E(\hat{P})$, $W_F(\hat{P})$ and WU. The definitions are complimentary to the grammar introduced in Table 1.

2.3.1.1 Exception Block

Given a tuple of processes $\hat{P} = (P_1, \ldots, P_n)$ associated with exception block *workunits* for the tuple of faults $(\tilde{x})$, $W_E(\hat{P})$ is defined as:

$$W_E(\hat{P}) = in_{eb}().\left(\sum_i \left(\left(x_i^{handle}(\tilde{y}).P_i\right) \Vert \bar{y}_{eb}\langle\rangle.0\right) + un_{eb}()\right)$$

The tuple of processes are provided by the choreography designer at design time. The arity of the tuple is equal to the number of exception types being handled. The exception block is enabled through the channel $in_{eb}()$. The exception block is represented as a guarded sum that performs P_i when exception x_i is detected which is semantically equivalent to an exception block with multiple guarded workunits. The guard in the workunit is used to match the triggered fault. After executing P_i, it signals its termination to the enclosing choreography using the channel $\bar{y}_{eb}$. The channel un_{eb} is used to uninstall the exception block.

2.3.1.2 Finalizer Block

Given a tuple of processes $\hat{P} = (P_1, \dots, P_n)$ that represents the actions for the finalizer block activities. Let the tuple $(\tilde{x})$ represent the names for the finalizer blocks, $W_F(\hat{P})$ is defined as:

$$W_F(\hat{P}) = in_{fb}().\left(\sum_i \left(\left(x_i^{finalize}(\tilde{y}).P_i\right) \parallel \bar{y}_{fb}\langle\rangle.0\right) + un_{fb}()\right)$$

The choreography designer is responsible for specifying the process for each finalizer block at design time. The finalizer blocks are enabled using the input channel in_{fb} and wait for one of the finalization blocks to perform P_i through the channel $x_i^{finalize}$. The finalization blocks can also be uninstalled, after installing them, using the channel un_{fb}. This last capability is to accommodate for the case of when the choreography completion condition is met, but the finalizer is not executed yet.

2.3.1.3 Choreography

The choreography is represented as a multi-hole context where the main activity of the choreography (*P*), nested choreographies (*NR*), exception and finalizer blocks (*H*) are provided by the choreography designer. A choreography is defined as follows:

Let $\tilde{x} = (in_{eb},\ un_{eb},\ in_{fb},\ un_{fb},\ y_{eb},\ y_{fb},\ y_{fc},\ x_{com},\ t,\ o,\ q,\ c)$

$$\begin{aligned}(\upsilon\ \tilde{x})\{P, NR, H\} ::= \quad & (\upsilon\ \tilde{x}) \\ & P \parallel \bar{t}\langle\rangle.0 \\ & \mid NR \\ & \mid H \\ & \mid \overline{in_{eb}}\langle\rangle \\ & \mid \left(c().\left(\overline{un_{eb}}\langle\rangle.\overline{in_{fb}}\langle\rangle.0\right) + q().0\right) \\ & \mid y_{fc}().\left(\overline{un_{eb}}\langle\rangle.\bar{o}\langle\rangle.\bar{q}\langle\rangle.0 \mid CC(P_1)\right) \\ & \mid \left(\left(t().\bar{c}\langle\rangle.y_{fb}().0\right) + y_{eb}().0 + x_{com}().\overline{y_{fc}}\langle\rangle.0\right)\end{aligned}$$

where:

- $P \parallel \bar{t}\langle\rangle.0$ represents the normal completion of the choreography's main activity followed by an output on channel t.

- NR represents zero or more local choreographies. NR, itself, is a multi-hole context, since it is defined in terms of a choreography. It is the responsibility of the choreography designer to provide NR in accordance with the *Non-Root choreography* production rule.

- H represents choreography exception and finalizer handlers and is a multi-hole context (refer to 3.1.1 and 3.1.2).

- $\overline{in_{eb}}\langle\,\rangle$ installs the exception blocks.

- In the case of normal completion of the main activity of the choreography, $c(\,).(\overline{un_{eb}}\langle\,\rangle.\overline{in_{fb}}\langle\,\rangle.0)$ uninstalls the exception blocks and installs the finalizer blocks.

- $y_{fc}(\,).\left(\overline{un_{eb}}\langle\,\rangle.\bar{o}\langle\,\rangle.\bar{q}\langle\,\rangle.0 \mid CC(P_1)\right)$ is executed when the completion condition is met while the choreography is in the enabled state. The term uninstalls the exception block, signals the main activity to terminate (using the channel *o)* and disables installing the finalizer block. In parallel, $CC(P_1)$ is defined as:

$$CC(P_1) = \prod_{x'_{com} \in S_n(P_1)} \overline{x'_{com}}\langle\,\rangle$$

and forces all enclosed choreographies to close through the channels in $S_n(P_1)$, where P_1 represents the choreography.

- The summation $\left(\left(t(\,).\bar{c}\langle\,\rangle.y_{fb}(\,).0\right) + y_{eb}(\,).0 + x_{com}(\,).\overline{y_{fc}}\langle\,\rangle.0\right)$ plays a multi-purpose role. $\left(t(\,).\bar{c}\langle\,\rangle.y_{fb}(\,).0\right)$ indicates a *Successfully Completed* [4] state, enables the finalizer block and waits for the completion of the finalizer block execution. Upon receiving the finalizer block signal through channel y_{fb}, the choreography is considered in the *Closed* state [4]. Otherwise if an exception occurred then it waits for the completion of the execution of the exception block using the channel y_{eb}. The choreography moves to the *Closed* state after the exception execution is complete. The last term $x_{com}(\tilde{y}\,).\overline{y_{fc}}\langle\,\rangle.0$ is used to force closing any enclosed choreographies.

2.3.1.4 Workunit

A *workunit* prescribes the constraints that have to be fulfilled for performing some activity within a workflow. A *workunit* has three Boolean properties that control its behavior: guard, repeat and block. The combination of values for these properties and the observed behavior of a *workunit* is summarized in Table 2 (P is the process that represents the behavior of the main activity inside the *workunit*, and $A_p \stackrel{\text{def}}{=} P \mid A_p$).

Block	Guard	Repeat	Behavior
false	false	false	Behaves as a nil process
false	false	true	Behaves as a nil process
false	true	false	Behaves as a process P
false	true	true	Behaves as A_p
true	false	false	Behaves as a process P
true	false	true	Behaves as A_p
true	true	false	Behaves as a process P
true	true	true	Behaves as A_p

Table 2: Workunit Behavior

The guard and repeat attributes' evaluation is both data and event driven (the availability of variables events). We can safely assume that the evaluation of these attributes is purely event driven, since variables are situated and not local. A situated variable indicates that the variable information resides at a participant who is assuming a role type in the choreography. Therefore, the fact that the variable is evaluated to true or false based upon data is considered an event by itself.

Again, we leverage the hole context to model the behavior of a *workunit* where the main activity must be provided by the choreography designer.

Let $\tilde{x} = (y_{repeat}, y_{once}, y_{nil})$ where:

- t_r and f_r are used to signal true and false for the repeat property respectively, and
- t_g and f_g are used to signal true and false for the guard property respectively.

$$
\begin{aligned}
(\upsilon\ \tilde{x})\{P\} ::= \quad & (\upsilon\ \tilde{x}) \\
& \Big(z(\tilde{y}).\Big(\big(y_{repeat}().P \mid (\upsilon\ \tilde{x})\{P\} + y_{once}().P + y_{nil}().0\big) \\
& \mid (if\ (y_1 = false) \\
& \ then \\
& \ \big(f_g().f_r().\overline{y_{nil}}\langle\rangle.0 + t_g().f_r().\overline{y_{once}}\langle\rangle.0 \\
& \ + f_g().t_r().\overline{y_{nil}}\langle\rangle.0 + t_g().t_r().\overline{y_{repeat}}\langle\rangle.0\big) \\
& \ else \\
& \ t_r().\overline{y_{repeat}}\langle\rangle.0 + f_r().\overline{y_{once}}\langle\rangle.0\big)\Big)\Big)
\end{aligned}
$$

Where:

- The channel *z* receives a name to indicate whether the *workunit* blocking behavior is enabled.
- The term $y_{repeat}().P \mid\mid (\upsilon\ \tilde{x})\{P\} + y_{once}().P + y_{nil}().0$ represents the behavior of the *workunit* which can be either: (1) a repeated execution of *P*, (2) a onetime execution of *P* or (3) it behaves as a nil process and does nothing.
- y_1 is used to model the block property.
- $f_g().f_r().\overline{y_{nil}}\langle\rangle.0 + t_g().f_r().\overline{y_{once}}\langle\rangle.0 + f_g().t_r().\overline{y_{nil}}\langle\rangle.0 + t_g().t_r().\overline{y_{repeat}}\langle\rangle.0$ handles the case when the block property is set to false and behaves according to the values of the guard and repeat (refer to Table 2) which are signaled through the channels: t_r, f_r, t_g, and f_g.

The term $t_r().\overline{y_{repeat}}\langle\rangle.0 + f_r().\overline{y_{once}}\langle\rangle.0$ handles the case where the block property is true.

2.3.2 Operational Semantics

The definition of the operational semantics of Chor-calculus is a two-step process 1) define the structural congruence which is the equivalence relationship between two processes [21], and 2) specify the reduction rules which identify how processes evolve by interacting with each other [21].

Reaction Rule	Name
$\tau.P + Q \longrightarrow P$	TAU (1)
$\overline{x^t}\langle\tilde{y}\rangle.P + M \mid x(\tilde{z}).Q + N \longrightarrow P \mid \{\tilde{y}/\tilde{z}\}Q$	REACT (2)
$\dfrac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}$	PAR (3)
$\dfrac{P \longrightarrow P'}{P \parallel Q \longrightarrow P' \parallel Q}$	SEQ(4)
$\dfrac{P \equiv P' \quad P \longrightarrow Q \quad Q \equiv Q'}{P' \longrightarrow Q'}$	STRUCT (5)
$\dfrac{P \longrightarrow P' \text{ and } x = y}{\text{if } (x = y) \text{ then } P \text{ else } Q \longrightarrow P'}$	IFT (6)
$\dfrac{Q \longrightarrow Q' \text{ and } x \neq y}{\text{if } (x = y) \text{ then } P \text{ else } Q \longrightarrow Q'}$	IFF (7)

Table 3: Reduction Rules for Chor-Calculus

Definition 1. *The set of free names in P, fn(P), is defined as:*

1. $fn(P \mid P') = fn(P) \cup fn(P')$
2. $fn(P \parallel P') = fn(P) \cup fn(P')$
3. $fn(0) = \emptyset$
4. $fn(x(\tilde{y}).P) = \{x\} \cup fn(P) \backslash \{\tilde{y}\}$
5. $fn(\bar{x}\langle\tilde{y}\rangle.P) = \{x, \tilde{y}\} \cup fn(P)$
6. $fn(P + P') = fn(P) \cup fn(P')$
7. $fn(!P) = fn(P)$

The names that are not occurring free in P are referred to as bounded names and denoted by *bn(P)*.

Definition 2. *The structural congruence (≡) is the least congruence closed under the following rules:*

1. Changing bounded names using alfa-conversion
2. $P + Q \equiv Q + P$
3. $P \mid 0 \equiv P, P \mid Q \equiv Q \mid P, P \mid (Q \mid R) \equiv (P \mid Q) \mid R$
4. $P \mid (\upsilon\, x)\{Q\} \equiv (\upsilon\, x)(P \mid Q)\; x \notin fn(P)$
5. $R \mid NR \equiv NR \mid R, NR \mid NR' \equiv NR' \mid NR$
6. $0 \parallel P \equiv P$
7. $!P \equiv P \mid !P, !C \equiv C \mid !C, !z^c(\tilde{y}).C \equiv z^c(\tilde{y}).C \mid !z^c(\tilde{y}).C$
8. The rules of the form:

$$\frac{P \equiv Q}{CO[P] \equiv CO[Q]}$$

where $CO[\,.\,]$ represents any context of the form $M \mid [\,.\,]$, $M \mid\mid [\,.\,]$ or $[\,.\,] \mid\mid M$.

Definition 3. *The reduction rule ($\longrightarrow$) is the smallest relationship satisfying the rules in Table 3 and closed with respect to $\equiv$:*

2.4 Mapping Chor-calculus to WS-CDL

In this section, we introduce the mapping from Chor-calculus to WS-CDL. The mapping will enable the generation of WS-CDL code from the specification captured in Chor-calculus. Generating WS-CDL from the verified calculus will result in fewer errors in the WS-CDL artifact. For this purpose, we define the function $F(P_{Chor-calculus}) = E_{ws-cdl}$ which maps a Chor-calculus process to WS-CDL entities. There are several constraints imposed by the translation process:

- A system must have one root choreography which is specified in the form of a non-guarded replication.
- The name of a service operation must be unique across all the ports/interfaces.
- A root choreography must not specify any finalizer blocks [4].

- The transformation doesn't generate the timeout property for an interaction because it is not supported by the calculus.

2.4.1 Nil Process

The 0 process is mapped to a *NoAction* activity that does nothing.

$F(0) ::= < noAction />$

2.4.2 Tau

The silent action is mapped to the silentAction activity.

$F(\tau) ::= < silentAction />$

2.4.3 Output

The annotated output is used to signify one of the following operations: request (rq), request-response (rqrs), perform (pr) and finalize (fn).

2.4.3.1 Request

A request output indicates a one-way interaction between two participants. It is mapped to an interaction between two participants that has one exchange whose action property is set to request. To model an exception, we represent it using the exchange of the fault construct instead of using the receive exchange construct (*causeException*) attribute. Such a representation aligns more naturally with the web service messaging pattern where a fault is generated by sending a fault in the response message.

$$
\begin{aligned}
&F(\overline{x^{rq}}\langle\hat{y}\rangle.P) ::= \\
&\quad < interaction\ channelVariable = "y_1" \\
&\quad\ operation = "x"\ initiate = "false" > \\
&\quad\quad < participate\ relationshipType = "y_2" \\
&\quad\quad\ fromRoleTypeRef = "y3"\ toRoleTypeRef = "y_4"/> \\
&\quad\quad < exchange\ action = "request"\ > \\
&\quad\quad\ < send\ variable = "cdl: getVariable('y_5','','','y_3')"/> \\
&\quad\quad\ < receive\ variable = "cdl: getVariable('y_6','','','y_4')"/> \\
&\quad\quad < /exchange > \\
&\quad < /interaction > \\
&\quad F(P)
\end{aligned}
$$

2.4.3.2 Response

An annotated response term is mapped to an interaction with one exchange block that represents a response from a partner. An optional exchange block can be specified to represent the communication of a fault from the receiver to the sender.

$$
\begin{aligned}
&F(\overline{x^{rs}}\langle\hat{y}\rangle.P) ::= \\
&\quad < interaction\ channelVariable = "y_1" \\
&\quad\ operation = "x"\ initiate = "false" > \\
&\quad\quad < participate\ relationshipType = "y_2" \\
&\quad\quad\ fromRoleTypeRef = "y3"\ toRoleTypeRef = "y_4"/> \\
&\quad\quad < exchange\ action = "respond"\ > \\
&\quad\quad\quad < send\ variable = "cdl: getVariable('y_5','','','y_3')"/> \\
&\quad\quad\quad < receive\ variable = "cdl: getVariable('y_6','','','y_4')"/> \\
&\quad\quad </exchange > \\
&\quad </interaction > \\
&\quad F(P)
\end{aligned}
$$

2.4.3.3 Request-Response

An annotated request-response term is mapped to an interaction with two exchange blocks. The first exchange block represents the request interaction, while the second block represents the response interaction. An optional exchange block can be specified to represent the communication of a fault from the receiver to the sender.

$F(\overline{x^{rqrs}}\langle\hat{y}\rangle.P) ::=$

$< interaction\ channelVariable = "y_1"$
$operation = "x"\ \ initiate = "false" >$
$< participate\ relationshipType = "y_2"$
$fromRoleTypeRef = "y3"\ toRoleTypeRef = "y_4"/>$
$< exchange\ action = "request" >$
$< send\ variable = "cdl: getVariable('y_5','','','y_3')"/>$
$< receive\ variable = "cdl: getVariable('y_6','','','y_4')"/>$
$</exchange >$
$< exchange\ action = "respond" >$
$< send\ variable = "cdl: getVariable('y_7','','','y_4')"/>$
$< receive\ variable = "cdl: getVariable('y_8','','','y_3')"/>$
$</exchange >$
$< exchange\ name = "fault" >$
...
$</exchange >$
$</interaction >$
$F(P)$

2.4.3.4 Perform

A perform annotated output is mapped to a WS-CDL *perform* activity. The activity enables a specific choreography by specifying the choreography's name. The channel name is used as the name of the choreography. The performing choreography can specify the variables to bind in the performed choreography. A choreography can be performed in two modes: a blocking mode where the calling activity waits for the performed choreography to complete; and a non-blocking mode where the calling activity continues the execution in parallel to the performed choreography.

1) **Blocking Mode**

$$F(\overline{x^{pr}}\langle\tilde{y}\rangle.P) ::=$$
$$< perform\ choreographyName = "x"$$
$$choreographyInstanceId = "y_1"\ \boldsymbol{block} = "\boldsymbol{true}" >$$
$$< bind\ name = "n_1" >$$
$$< this\ \ variable = "cdl: getVariable('y_2','','')"$$
$$roleType = "y_3" />$$
$$< free\ variable = "cdl: getVariable('y_4','','')"$$
$$roleType = "y_5" />$$
$$</bind >$$
$$</perform >$$
$$F(P)$$

2) Non-Blocking Mode

$$F(\overline{x^{pr}}\langle\tilde{y}\rangle.0 \mid P) ::=$$
$$< perform\ choreographyName = "x"$$
$$choreographyInstanceId = "y_1"\ \boldsymbol{block} = "\boldsymbol{false}" >$$
$$< bind\ name = "n_1" >$$
$$< this\ variable = "cdl: getVariable('y_2','','')"$$
$$roleType = "y_3" />$$
$$< free\ variable = "cdl: getVariable('y_4','','')"$$
$$roleType = "y_5" />$$
$$</bind >$$
$$</perform >$$
$$F(P)$$

2.4.3.5 Finalize

A finalize annotated output signals a choreography to execute one of the finalizer blocks. The name of the channel is used to identify the finalizer block to execute.

$$F\left(\overline{x^{fn}}\langle\tilde{y}\rangle.P\right) ::=$$
$$< finalize\ choreographyName = "y_1"$$
$$choreographyInstaneId = "y_2"\ finalizerName = "x" />$$

2.4.3.6 Parallel

The parallel operator is mapped to the WS-CDL ordering structure activity *Parallel*.

$$F(P \mid Q) ::= < parallel > F(P)F(Q) </parallel >$$

2.4.3.7 Sequence

The sequence operation || is mapped to the WS-CDL ordering structure activity *sequence*.

$F(P \mid\mid Q) ::= < sequence > F(P)F(Q) </sequence >$

2.4.3.8 Guarded Choice

The guarded choice is mapped to the WS-CDL ordering structure activity *choice*.

$F(G + G') ::= < choice > F(G)F(G') </choice >$

2.4.3.9 Workunit

A workunit term is mapped to the WS-CDL *workunit* activity.

$F\big((\upsilon\ \tilde{x})\{P\}\big) ::=$
$< workunit\ name = "NCName"$
$guard = "condition\ expression"$
$repeat = "condition\ expression"\ block = "y_1" >$
$F(P)$
$</workunit >$

2.4.3.10 Exception Block

An exception block is translated to a set of WS-CDL workunits with each *workunit* handling a specific type of exception. The workunit must have the block property set to false, and the repeat attribute must not be specified. A *workunit* with a missing guard is intended to handle all exceptions.

$F\left(W_{EB}(\hat{A})\right) ::=$
$< exceptionBlock >$
$< workunit\ guard =$ "cdl:hasExceptionOccurred('x_1')"
$block = "false" >$
$F(A_1)$
$</workunit >$
$< workunit\ guard = "\ldots"\ block = "false" > \cdots </workunit >$
$< workunit\ block = "false" >$
$F(A_n)$
$</workunit >$
$</exceptionBlock >$

2.4.3.11 Finalizer Blocks

Each occurrence of a finalizer block in the summation is translated to a WS-CDL finalizerBlock entity.

$$F\left(W_{FB}(\hat{A})\right) ::=$$
$$< finalizerBlock\ name = \text{"x}_1\text{"} >$$
$$F(A_1)$$
$$</finalizerBlock >$$
$$< finalizerBlock\ name = "\ldots" > \cdots </finalizerBlock >$$
$$< finalizerBlock\ name = "x_n" >$$
$$F(A_n)$$
$$</finalizerBlock >$$

2.4.3.12 Choreography

The translation differentiates the occurrences of root and non-root choreographies. Mainly, a root choreography is mapped to a WS-CDL package with one choreography definition marked as a root. On the other hand, a non-root choreography is mapped to a choreography definition whose root property is set to false.

1) **Root-Choreography**

$$F(R) = \left(!\left((\upsilon\ \tilde{x})\{P, NR, H\}\right)\right) ::=$$
$$< package\ name = "NCName"$$
$$targetNamespace = "uri"$$
$$xmlns = "http://www.w3.org/2005/10/cdl" >$$
$$< informationType/>$$
$$< token\ />$$
$$< tokenLocator\ />$$
$$< roleType\ />$$
$$< relationshipType\ />$$
$$< participantType\ />$$
$$< channelType\ />$$
$$< choreography\ root = "true" >$$
$$F(NR)$$
$$F(P)$$
$$F\left(W_{EB}(\hat{A})\right)$$
$$</choreography >$$
$$</package >$$

2) **Non-Root Choreography**

$$F(NR) = F\left(!\left(z(\tilde{y}).\left((\upsilon\, \tilde{x})\{P, NR, H\}\right)\right)\right) ::=$$

$$< choreography\ name = "z"\ \ root = "false" >$$

$$F(NR)$$

$$F(P)$$

$$F\left(W_{EB}(\hat{A})\right)$$

$$F\left(W_{FB}(\hat{A})\right)$$

$$</choreography >$$

2.5 Evaluation

In [22], the authors investigate WS-CDL expressiveness to model two categories of patterns:

1. Workflow patterns which capture recurrent control-flow dependencies in business processes.
2. Service interaction patterns which capture recurrent compositions of interaction patterns.

We leverage the work from [22] to evaluate the expressiveness and comprehensiveness of Chor-calculus. Principally, we assess the language in term of its capabilities to model the two collections of patterns supported by WS-CDL. Table 4 and Table 5 taken from [22] list the patterns supported by both WS-CDL and Chor-calculus.

Workflow Pattern	WS-CDL	Chor-calculus
1. Sequence	+	+
2. Parallel Split	+	+
3. Synchronization	+	+
4. Exclusive Choice	+	+
5. Simple Merge	+	+
6. Multiple Choice	+	+
7. Synchronizing Merge	+	+
8. Multiple Merge	+/-	+/-
9. Discriminator	-	-
10. Arbitrary cycles	-	-
11. Implicit Termination	+	+
12. MI without synchronization	+	+
13. MI with a priori design time knowledge	+	+
14. MI with a priori runtime knowledge	-	-
15. MI with no a priori runtime knowledge	-	-
16. Deferred Choice	+	+
17. Interleaved Parallel Routing	-	-
18. Milestone	+	+
19. Cancel Activity	+	+
20. Cancel Case	+	+

Table 4: Workflow Patterns Supportability where (+) indicates a support of, (-) no support of and (+/-) indicates a partial support of the pattern.

We have selected some of the more advanced patterns; and describe how they can be modeled using Chor-calculus to illustrate the analysis. The full list of Chor-calculus pattern implementations are shown in Table 5 & Table 6.

Service Interaction Patterns	WS-CDL	Chor-calculus
1. Send	+	+
2. Receive	+	+
3. Send/receive	+	+
4. Racing incoming messages	+	+
5. One-to-many send	+/-	+/-
6. One-from-many receive	+	+
7. One-to-many send/receive	+/-	+/-
8. Multi-responses	+	+
9. Contingent requests	+/-	-
10. Atomic multicast notification	-	-
11. Request with referral	+	+
12. Relayed request	+	+

Table 5: Service Interaction Patterns Supportability

WP	Chor-calculus Expression
1	$(v\ x)\{P\} \parallel (v\ y)\{P'\}$
2	$(\overline{x^{pr}}\langle\tilde{y}\rangle.0 \mid P) \mid !\,x(\tilde{z}).C$
3	$\big((\bar{w}(true).0 \parallel u(m).(v\ \tilde{x})\{P\}) \mid w(n).(v\ \tilde{x})\{Q\} \parallel \bar{u}(true).0\big)$
4	$x_1(\tilde{i}).P_1 + x_2(\tilde{j}).P_2$
5	$(x_1(\tilde{i}).P_1 + x_2(\tilde{j}).P_2) \parallel P_3$
6	$(v\ \tilde{x})\{P\} \mid (v\ \tilde{y})\{Q\} \mid (v\ \tilde{z})\{R\}$
7	$\big((v\ \tilde{x})\{P\} \mid (v\ \tilde{y})\{Q\} \mid (v\ \tilde{z})\{R\}\big) \parallel S$
8	$(\overline{x^{pr}}\langle\tilde{y}\rangle.0 \mid P) \mid !\,\big(x(\tilde{z}).\big((v\ \tilde{x})\{(Q \mid R \mid S), X\}\big)\big)$
9	N/A
10	N/A
11	Any arbitrary choreography system
	$(!\,C_1) \mid \big((\overline{m^{pr}}\langle\tilde{y}\rangle.0 \mid P) \mid !\,m(\tilde{z}).C_2\big) \mid \big((\overline{n^{pr}}\langle\tilde{y}\rangle.0 \mid P) \mid !\,n(\tilde{z}).C_3\big)$
12	$\bar{z}\langle true\rangle.\overline{t_r}\langle\rangle \mid (v\ \tilde{x})\{(\overline{o^{pr}}\langle\tilde{y}\rangle.0 \mid P)\} \mid !\,o(\tilde{z}).C$
13	$\bar{z}\langle false\rangle.\overline{t_r}\langle\rangle.\bar{z}\langle false\rangle.\overline{t_r}\langle\rangle.\bar{z}\langle false\rangle.\overline{t_r}\langle\rangle \mid (v\ x)\{P\}$
14	N/A
15	N/A
16	$(v\ \tilde{x})\{P\} + (v\ \tilde{y})\{Q\}$
17	N/A
18	$(\bar{z}\langle false\rangle) \mid \big((v\ \tilde{x})(\overline{o^{rqrs}}\langle\rangle.\bar{q}\langle true\rangle)\big) \mid \big((v\ \tilde{y})(Q)\big)$
19	Supported through the forceful termination of enclosed choreographies when an exception occurs
20	Supported through the forceful termination of a root choreography when an exception occurs

Table 6: Workflow Patterns in Chor-calculus

2.5.1 Workflow Patterns

2.5.1.1 Synchronization

The authors in [22] propose using arbitrary workunits with variables to implement the synchronization pattern in WS-CDL. The Chor-calculus for the synchronization example in [22] is:

Let $\tilde{z} = \{w, u\}$

$(v\ \tilde{z})\big((\overline{interactionAB^{rqrs}}\langle\rangle \parallel \bar{w}(true) \parallel \overline{interactionBC^{rqrs}}\langle\rangle$
$\parallel u(m).(v\ \tilde{x})\{\overline{interactionBA^{rqrs}}\langle\rangle\}\big)$
$\mid (\overline{interactionDB^{rqrs}}\langle\rangle \parallel w(n).(v\ \tilde{x})\{\overline{interactionBD^{rqrs}}\langle\rangle\}$
$\parallel \bar{u}(true)\big)$

2.5.1.2 MI without synchronization

The MI without synchronization pattern can be realized by having a non-blocking perform activity inside a workunit with a repeat condition. The perform activity will perform multiple instances of the same choreography concurrently:

$\bar{z}\langle true\rangle.\,\overline{t_r}\langle\,\rangle.\,0$
$|(v\ \tilde{x})\{(\overline{buyerSellerChor^{pr}}\langle\tilde{y}\rangle.\,0\mid P)\}$
$|\,!\,buyerSellerChor(\tilde{z}).\,C$

where $(v\ \tilde{x})\{(\overline{buyerSellerChor^{pr}}\langle\tilde{y}\rangle.\,0\mid P)\}$ represents the *workunit* that performs the choreography *buyerSellerChor*.

2.5.1.3 MI with a priori design time knowledge

The pattern executes a specific activity n times where n is a constant specified at design time. The pattern can be encoded in Chor-calculus using a work unit that is invoked n times. In the example below, the workunit represented by $(v\ x)\{P\}$ is invoked three times (3 consecutive occurrences of $\bar{z}\langle false\rangle.\,\overline{t_r}\langle\,\rangle$):

$\bar{z}\langle false\rangle.\,\overline{t_r}\langle\,\rangle.\,\bar{z}\langle false\rangle.\,\overline{t_r}\langle\,\rangle.\,\bar{z}\langle false\rangle.\,\overline{t_r}\langle\,\rangle.\,0\mid(v\ x)\{P\}$

SIP	Chor-calculus Support
1	$\overline{x^{rq}}\langle\tilde{y}\rangle$
2	$\overline{x^{rs}}\langle\tilde{y}\rangle$
3	$\overline{x^{rqrs}}\langle\tilde{y}\rangle$
4	$\overline{x_1{}^{rq}}\langle\tilde{u}\rangle+\ \overline{x_2{}^{rq}}\langle\tilde{v}\rangle+\ \overline{x_3{}^{rq}}\langle\tilde{w}\rangle$ *where* u_1, v_1 *and* w_1 *are channels to the same recepient*
5	$\overline{x_1{}^{rq}}\langle\tilde{u}\rangle\mid\overline{x_2{}^{rq}}\langle\tilde{v}\rangle\mid\overline{x_3{}^{rq}}\langle\tilde{w}\rangle$ *where the number of recipients is known at design time*
6	$\bar{z}\langle false\rangle.\,\overline{t_r}\langle\,\rangle.\,\bar{z}\langle false\rangle.\,\overline{t_r}\langle\,\rangle.\,\bar{z}\langle false\rangle.\,\overline{t_r}\langle\,\rangle\mid(v\ x)\{\overline{x^{rs}}\langle\tilde{y}\rangle\}$
7	$\overline{x_1{}^{rqrs}}\langle\tilde{u}\rangle\mid\overline{x_2{}^{rqrs}}\langle\tilde{v}\rangle\mid\overline{x_3{}^{rq}}rs\langle\tilde{w}\rangle$ *where the number of recipients is known at design time*
8	$\overline{x^{rq}}\langle\tilde{y}\rangle$ $\parallel(\bar{z}\langle false\rangle.\,\overline{t_r}\langle\,\rangle.\,\bar{z}\langle false\rangle.\,\overline{t_r}\langle\,\rangle.\,\bar{z}\langle false\rangle.\,\overline{t_r}\langle\,\rangle\mid(v\ x)\{\overline{x^{rs}}\langle\tilde{y}\rangle\})$
9	Not Supported
10	N/A
11	$\overline{x^{rq}}\langle\tilde{y}\rangle.\,P$ where the channel (referral is passed as variable y_1)
12	$\overline{x^{rq}}\langle\tilde{y}\rangle$ *where* $\tilde{y}$ *contains the channel passed*

Table 7: Interaction Patterns in Chor-calculus

2.5.2 Service Interaction Patterns

2.5.2.1 Racing Incoming Message

According to [22], a party is expected to receive one among a set of messages which can be expressed in WS-CDL using a choice with multiple interactions operation with the same recipient. The Chor-calculus for this interaction pattern is:

$\overline{x_1{}^{rq}}\langle\tilde{u}\rangle+\ \overline{x_2{}^{rq}}\langle\tilde{v}\rangle+\ \overline{x_3{}^{rq}}\langle\tilde{w}\rangle$

where u_1, v_1 and w_1 are channels to the same recipient.

2.5.2.2 Request with Referral

According to [22], participant A communicates with participant B and requests that any follow up responses be sent to one or more parties other than A. This is supported in WS-CDL using channel passing and is expressed as follows:

$\overline{x^{rq}}\langle\tilde{y}\rangle$ where $\tilde{y}$ is the channel passed during the interaction.

2.5.2.3 Contingent Request

According to [22], in a contingent request pattern, a party X makes a request to party Y, and if a response is not received in a timely fashion, a new request is issued to another party Z and so on. This pattern is not supported by Chor-calculus because it depends on timeouts which is not covered by this version of our calculus.

2.6 Chor-calculus Demonstration

In this section, we illustrate how to model a choreography using Chor-calculus. We first introduce a purchase order scenario and describe the steps in the business process to realize the choreography. Next, we present the calculus for the choreography and provide the translation to WS-CDL code. Finally we demonstrate how to use the HAL toolkit to verify the correctness of the calculus.

2.6.1 Purchase Order Scenario

The purchase order scenario involves three parties that collaborate together to realize the purchase and procurement of certain merchandise. The parties are: buyer, seller and shipper. The following sequence diagram shows the ordering of interactions among the partners:

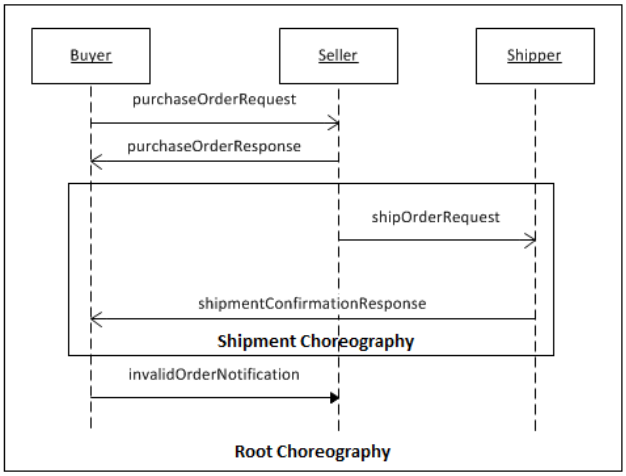

Figure 1: Purchase Order Interaction Sequence

The choreography is initiated when the buyer submits a purchase order request (*purchaseOrderRequest* message). The seller processes the request and responds with an acknowledgement (*purchaseOrderResponse* message). Following that, the seller initiates the "Shipment Choreography" which takes care of shipping the order to the buyer. The choreography is performed when a *shipOrderRequest* message is sent from the seller to the buyer. Along with that request, the seller sends additional information on how to further communicate with the buyer (Request with Referral pattern). Once the shipment order is processed, the shipper responds to the buyer with shipment details (*shipmentConfirmationResponse* message) which concludes the choreography. The buyer throws an exception in the case of the shipper sending invalid shipment details (e.g. invalid reference number or a missing item in the delivered order). The exception is handled in the root choreography which sends a notification from the buyer to the seller.

In the context of WS-CDL, the purchase order is made up of two choreographies 1) the root choreography which takes care of placing the purchase order with the seller, and 2) a shipment choreography which takes care of shipping the order. The shipment choreography is an enclosed choreography and is performed by the root choreography.

2.6.2 Chor-calculus Model

Referring to the purchase order example introduced in the previous section, the model using Chor-calculus is:

$$PurchaseOrder = \ !PC$$

$$PC = \left(v\ shipmentChoreography^{c}, in_{eb}, un_{eb}, in_{fb}, un_{fb}, y_{fb}, y_{fc}, x_{com}\right)\{P, NR, H\}$$

$$P = \overline{purchaseOrder^{rqrs}}\langle\tilde{x}\rangle.0 \parallel \overline{shipmentChoreography^{pr}}\langle\tilde{y}\rangle.0$$

$$\tilde{x} = \{buyer2SellerChannel, buyer2SellerRelationship, BuyerRole, SellerRole, purchaseOrderRequest, purchaseOrderRequest, purchaseOrderResponse, purcahseOrderResponse,\ tobuyerChannel\}$$

$$\tilde{y} = \{buyer_1, BuyerRole, toBuyerChannel, SellerRole, toBuyerChannel\}$$

$$NR = \ !shipmentChoreography^{c}(\tilde{z}).QC$$

$$QC = \left(\left(v\ in_{eb},\ un_{eb}, in_{fb}, un_{fb}, y_{fb}, y_{fc}, x_{com}\right)\{Q, NR_1, H_1\}\right)$$

$$\tilde{z} = \{toBuyerChannel\}$$

$$H = W_E'\left(\widehat{A_{eb}}\right) \mid W_F'\left(\widehat{A_{fb}}\right)$$

$$W_E'\left(\widehat{A_{eb}}\right) = in_{eb}(\,).\left((invalidOrder^{handle}(orderDetails).R \parallel \bar{y}_{eb}\langle\,\rangle.0) + un_{eb}(\,)\right)$$

$$R = \overline{invalidOrder^{rq}}\langle\tilde{w}\rangle.0$$

$$\tilde{w} = \{buyer2SellerChannel, buyer2SellerRelationship, BuyerRole, SellerRole, invalidOrderNotification, invalidOrderNotification\}$$

$$W_F'\left(\widehat{A_{fb}}\right) = in_{fb}(\,).\overline{un}_{fb}\langle\,\rangle.0$$

$$Q = \overline{shipOrder^{rq}}\langle\tilde{u}\rangle.0 \parallel \overline{shipmentConfirmation^{rs}}\langle\tilde{v}\rangle.0$$

$$\tilde{u} = \{seller2ShipperChannel, seller2ShipperRelationship, SellerRole, ShipperRole, shipOrderRequest, shipOrderRequest\}$$

$$\tilde{v} = \{toBuyerChannel, buyerToShipperRelationship, ShipperRole, BuyerRole, shipmentConfirmationResponse, shipmentConfirmationResponse\}$$

$$NR_1 = 0$$

$$H_1 = W_E''(\widehat{A_{eb}}) \mid W_F''(\widehat{A_{fb}})$$

$$W_E''(\widehat{A_{eb}}) = in_{eb}().un_{eb}().0$$

$$W_F''(\widehat{A_{fb}}) = in_{fb}().\overline{un}_{fb}\langle\rangle.0$$

2.6.3 Verification Process

In this section, we demonstrate the use of the HAL toolkit to reason about the correctness of the behavior of choreographies specified in Chor-calculus. First, the choreography is written using Chor-calculus and then translated to HAL compatible syntax. The transformation is syntactic and does not interfere in the verification process. Further program are listed under Appendix A – Chor-calculus section. The resultant pi-calculus for the purchase order is as follows:

 # Purchase Order Choreography

define **PC**(porqrs,scpr,sorq,scrs,x,y,iorh,iorq,io) =

(z)(ineb)(uneb)(infb)(unfb)(c)(yfc)(yeb)(yfb)(o)(q)(xcom)(t)(

P(porqrs,scpr,sorq,scrs,x,y,t,iorh,iorq,io) | **H**(iorh,iorq,io,yeb,ineb,uneb,infb,unfb) |

NR(scpr,sorq,scrs,x,y) | ineb!z.nil | (c?(z).uneb!z.infb!z.nil + q?(z).nil) |

yfc?(z).(uneb!z.o!z.q!z.nil | xcom!z.nil) | (t?(z).c!z.yfb?(z).nil + yeb?(z).nil + xcom?(z).yfc!z.nil))

define **P**(porqrs,scpr,sorq,scrs,x,y,t,iorh,iorq,io) =

(z)(porqrs!x.scpr!y.t!z.PC(porqrs,scpr,sorq,scrs,x,y,iorh,iorq,io))

define **WEB_PC**(iorh,iorq,io,yeb,ineb,uneb) = ineb?(z). iorh?(z).iorq!io.yeb!z.nil + uneb?(z).nil

define **WFB_PC**(infb,unfb) = infb?(z).unfb!z.nil

define **H**(iorh,iorq,io,yeb,ineb,uneb,infb,unfb) =

WEB_PC(iorh,iorq,io,yeb,ineb,uneb) | **WFB_PC**(infb,unfb)

 # Shipment Choreography

define **QC**(scpr,sorq,scrs,x,y) =
(z)(ineb)(uneb)(infb)(unfb)(c)(yfc)(yeb)(yfb)(o)(q)(xcom)(t)(

Q(scpr,sorq,scrs,x,y,t) | **H1**(ineb,uneb,infb,unfb,yeb) | ineb!z.nil |

(c?(z).uneb!z.infb!z.nil + q?(z).nil) | yfc?(z).(uneb!z.o!z.q!z.nil | xcom!z.nil) |

(t?(z).c!z.yfb?(z).nil + yeb?(z).nil + xcom?(z).yfc!z.nil))

define **Q**(scpr,sorq,scrs,x,y,t) = (z)(sorq!x.scrs!y.t!z.NR(scpr,sorq,scrs,x,y))

define **WEB_QC**(ineb,uneb,yeb) = ineb?(z).yeb!z.nil + uneb?(z).nil

define **WFB_QC**(infb,unfb) = infb?(z).unfb!z.nil

define **H1**(ineb,uneb,infb,unfb,yeb) = **WEB_QC**(ineb,uneb,yeb) | **WFB_QC**(infb,unfb)

define **NR**(scpr,sorq,scrs,x,y) = scpr?(z).QC(scpr,sorq,scrs,x,y)

build **PC**

Table 8 lists the mappings between pi-calculus and Chor-calculus symbols.

Name	Chor-calculus
PC	PC
P	P
NR	NR
H	H
WEB_PC	$W'_{EB}(\widetilde{A_{eb}})$
WFB_PC	$W'_{FB}(\widetilde{A_{fb}})$
QC	QC
Q	Q
H1	H_1
WEB_QC	$W''_{EB}(\widetilde{A_{eb}})$
WFB_QC	$W''_{FB}(\widetilde{A_{fb}})$
E	$PurchaseOrder$
porqrs	$purchaseOrder^{rqrs}$
scpr	$shipmentChoreography^{pr}$, $shipmentChoreography^{c}$
sorq	$shipOrder^{rq}$
scrs	$shipmentConfirmation^{rs}$
iorq	$invalidOrder^{rq}$
iorh	$invalidOrder^{handle}$

Table 8: Symbols Mapping To Chor-calculus

The desired properties for the calculus are expressed as pi-logic formulas. Finally, the calculus and formulas are loaded into the HAL toolkit to validate that the calculus satisfies the formulas. Issues detected by the toolkit are then fixed by the designer and the verification process is restarted. Referring to the purchase order example, the capability to always perform the nested shipment choreography is expressed in pi-logic as:

P1 = *AG(EF<scpr?z>true)*

The property of always receiving a shipment confirmation when performing the purchase order root choreography is expressed as:

P2 = *AG([porqrs!x]EF<scrs!y>true)*

The HAL toolkit asserts that the purchase order Chor-calculus[1] indeed exhibits the two properties (refer to po.pi and formulas.pl). Furthermore, mutations are introduced to the purchase order to demonstrate the effectiveness of the verification process in detecting bugs such as:

***M1*:** the capability *scrs!y* which signifies sending a shipment confirmation response is deleted. In the case the property P2 is not met and HAL validation fails (refer to po-m1.pi).

***M2*:** the capability *scpr!y* which signifies the performance of the nested shipment choreography is deleted. Again the property P2 is not met, since the shipment confirmation interaction will never happen (refer to po-m2.pi).

***M3*:** the deliberate reordering of capabilities porqrs!x and scpr!y is detected through HAL validation since P2 property is not met (refer to po-m2.pi).

There are a few considerations regarding the transformation from Chor-calculus to pi-calculus. Since HAL does not support replication, recursion is used instead. The sequential operator is implemented using action prefixing and synchronization between agents.

A known limitation to HAL is its ability to model check pi-calculus processes up to a certain level of complexity. Such processes are characterized by having finite numbers of states, and thus referred to as finitary processes. The infinite state problem or state explosion is more prominent in complex processes [23], [24]. State explosion is detected when verifying Chor-calculus processes with two or more levels of choreography nesting. While HAL is able to model check Chor-calculus processes with one level of nesting, this is not the case for deeper nesting. We argue that most choreographies exhibit one level of choreography nesting; and thus, the viability of using HAL for automated verification of the majority of Chor-calculus processes.

2.6.4 Translation to WS-CDL

We have developed a Java application that transforms[1] Chor-calculus to WS-CDL code. The parser for the tool is generated using Antlr v4 [25], [26], a popular language recognition tool. Antlr generates the parser from an existing BNF grammar. The grammar definition is a slightly modified version of the Chor-calculus grammar to make it editor friendly (e.g. editors do not support subscripts, superscripts, etc…). The modification to

[1] Available from http://www.steam.ualberta.ca/main/research_areas/chor_calculus.rar

the grammar does not interfere with the semantics of the Chor-calculus grammar and is purely syntactic. The application leverages the visitor pattern that traverses the parse tree generated by the parser and maps Chor-calculus tokens to WS-CDL constructs.

2.7 Related Work

Several formal methods in the literature are proposed to reason about both web service orchestrations and choreographies. The authors in [27] use pi-calculus to model the behavior of services to determine the degree of service compatibility in a choreography. The calculus in [27] is limited and is designed to model only primitive web service message exchange patterns and does not accommodate more advanced choreography scenarios. In [28], the authors provide a calculus for the Web service choreography interface (WSCI) and use it to reason about the compatibility of two or more services. Adaptors are generated if incompatibility is detected which are used to mediate between mismatched services. However WS-CDL supersedes WSCI which is a much less popular standard. Event-B is proposed in [14] to model the WS-CDL package through a set of transformation rules that generates Event-B models from an existing WS-CDL package. Compared to Chor-calculus, [14] does not handle nested choreographies and variable scoping, and thus does not demonstrate the feasibility of generating WS-CDL from a B-Event model. A specialized formal language called CDL is introduced in [29]. Unlike Chor-calculus, CDL doesn't cover many of the core features of WS-CDL such as channels, exceptions, finalizers and nested choreographies. A notable contribution is presented in [30] which introduces two paradigms for describing communication behaviors based on a formal calculi based on session-types. One of the paradigms is based on the idea of a global message flow which seems to be inspired by WS-CDL. While it originated from WS-CDL, no explicit mapping to WS-CDL is provided. In the area of error handling, the authors in [31] introduce a formal semantics for choreography exceptions as an extension to global calculus. In the context of global calculus, exceptions are treated as a form of transferring the execution to a different choreography. Related to choreographies, the authors in [32] describe interaction-oriented (IOC) and process-oriented (POC) as two approaches for describing choreographies and provide a formalization of equivalence between the two using a process algebra. In the same direction, the authors [33] introduce a new approach called the bipolar approach to

address design issues in service oriented systems by exploiting the synergy between chereopgraphy and orchestration languages. Such synergy is explored in terms of mathematical relations by using SOCK, a process algebra inspired by WS-BPEL.

2.8 Conclusions

In this paper, we present Chor-calculus, a new formal language based on pi-calculus and the semantics of WS-CDL. A mapping between Chor-Calculus and WS-CDL is provided which allows the generation of WS-CDL programs from a Chor-calculus specification. We evaluate the expressiveness of the language by modeling both the workflow, and interaction service, patterns, supported by WS-CDL. A tool is developed that generates WS-CDL programs from a Chor-calculus specification. Furthermore, we demonstrate the use of the HAL toolkit to verify the correctness properties of Chor-calculus choreographies.

One of the main objectives of WS-CDL is to be used along with other orchestrated systems such as WS-BPEL applications. We envisage using Chor-calculus with other calculi-based formal methods to investigate the compatibility of interacting services. For example, Chor-calculus can be used along with BP-calculus [5] and [27] to determine the behavioral service compatibility of existing systems.

3 A Workflow Approach to Identifying and Detecting WS-BPEL Temporal Fault Models

While there is considerable research on the monitoring of BPEL systems, limited effort has been dedicated to identifying faults manifested in such applications. In this paper, we identify temporal faults that might manifest themselves in BPEL programs. The fault models can be employed in mutation testing to assess the effectiveness of test suites or measure the accuracy of runtime monitors that are capable of verifying the correctness. We introduce a language to capture the trace behavior of BPEL programs and describe the fault models. We also propose a runtime verification system that consumes the specification language and detects temporal faults.

3.1 Introduction

The Business Process Execution Language (BPEL) is a standardized language for describing workflows composed of web services. BPEL leverages several web standards such as WSDL and SOAP. Increasing the reliability and stability of BPEL systems through testing and monitoring continues to be an active research area [34], [35], [36], [37], [38], [39]. Most of the literature in the area of monitoring of BPEL systems focuses either on the correctness of BPEL programs (e.g. pre-conditions and post-conditions are satisfied) or the compliance to predefined Service Level Agreements (SLAs). However, limited research exists on practical engineering approaches to the production of such systems especially in the area of classifying and identifying faults that might exist in BPEL systems.

In this paper, we identify temporal faults as one type of fault affecting BPEL programs. In the context of this paper, a temporal fault is any fault that leads to the out-of-order execution of activities. By identifying the temporal fault models in BPEL programs, we are effectively identifying the temporal mutants that can be inadvertently introduced to BPEL programs. Hence, based upon this identification, mutation testing can be utilized to assess the effectiveness of potential test suites. Moreover by utilizing the fault models, we can assess the accuracy and effectiveness of BPEL runtime monitors in detecting temporal faults. To identify the fault models, we follow a disciplined approach by analyzing well-known workflow patterns to derive the fault models. We introduce CSP_{BPEL} to describe the fault models, a notation derived from Communicating Sequential Processes (CSP) [40], [41], [42], as a formal language for describing patterns of interaction in concurrent systems. We use CSP_{BPEL} to capture the specifications of the derived fault models and the trace specification of BPEL programs. Furthermore, we introduce a new platform for the runtime verification of BPEL programs (RV-BPEL). Specifically, RV-BPEL undertakes runtime monitoring for the purpose of temporal fault detection. RV-BPEL main responsibility is to verify that the actual execution order of activities satisfies the expected behavior specified via the trace specification.

The contribution of this paper is two-fold. Firstly, it seeks to identify and provide a formal definition of a class of fault models, temporal faults, which can arise in BPEL

systems; and secondly, it produces a platform for the runtime verification of BPEL programs, specifically with regard to the detection of temporal faults.

The remainder of the paper is structured as follows: In the next section, we provide an overview about mutation testing and WS-BPEL. In section 3.3, we describe how we utilize a restricted set of CSP to describe BPEL processes. Moreover, the section describes the set of fault models inferred from workflow patterns. In Section 3.4, we propose a platform for runtime verification of BPEL programs to detect temporal faults. Section 3.5 describes research related to this paper; and finally, Section 3.6 presents the conclusions from the work.

3.2 Overview

In this section we provide a brief overview of WS-BPEL and mutation testing.

3.2.1 WS-BPEL

Service Oriented Architecture (SOA) is an architectural concept that promotes building and delivering software applications from smaller granular software units manifested as services [42], [44]. Web services are the best realization of building applications as services. Composing applications from autonomous web services involves the action of invoking web services in a certain temporal sequence. To describe this composition, BEA, IBM, Microsoft and other major software vendors have introduced the Web Services Business Process Execution Language (WS-BPEL). The most recent public release of its specification can be found in [45].

BPEL is an XML based language. It defines a model and grammar for describing the behavior of business processes based upon interactions between Web services [46]. BPEL defines several types of basic activities including but not limited to invoking Web service operations, receiving and replying to requests, and assigning data to messages. These basic activities can be combined into structured activities using sequencing, concurrency, conditional and repetition constructs, and selective communication mechanisms. In many ways, a BPEL system can be thought of as a set of distributed and interacting processes.

3.2.2 Mutation Testing

Mutation testing is a fault-based testing technique [47] and is used to measure the effectiveness of a test set in terms of its ability to detect faults [48]. A fault is small perturbation which is introduced into a program [49]. The concept of introducing a fault into a process is referred to fault injection. The perturbations are created by applying mutation operators which are syntactic changes to the program. In general, in order to avoid interaction effects between faults, one fault at a time is injected into the process. Test sets are then applied to the program to detect the variation in test results. A fault is killed when the runtime monitoring detects the fault [59]; a "dead" fault implies that the defect has been detected by the runtime monitoring system. In the context of this research, we use mutation testing to measure the effectiveness of our proposed monitoring engine in terms of its ability to detect temporal faults.

3.3 Production the Fault Models

The steps for producing the fault models are: (1) a subset of CSP is used to describe formally the specifications of fault models. And, (2) we identify fault models, which are defined as unexpected deviations from an expected execution flow. To rationalize this description, we utilize an existing description of the types of execution flow-types that are present in BPEL systems [51]. Hence, the task of describing fault models is simplified to defining fault patterns applicable to each execution flow pattern.

3.3.1 Specification Language

A formal specification language for describing temporal faults is necessary for several reasons. BPEL inherits its constructs from XLANG and WSFL which introduces construct redundancy. Under BPEL, for example the same concurrency behavior can be expressed using two different constructs. Moreover, a higher level specification language is required to abstract the underlying details of BPEL, generalize and formalize our temporal faults description to expand their applicability to other workflow technologies. Finally, a specification language is needed to describe the trace specification of a BPEL program which is consumed by the runtime monitoring system which is introduced later in this paper. We utilize CSP [41] and [42], a process algebra for describing patterns of interaction in concurrent systems, to describe our fault models. Process algebras

including CSP have been utilized to formally model BPEL and Web Service Choreography Description Language (WS-CDL) programs [52], [37] and [34]. To describe temporal aspects of BPEL systems, a subset of CSP notations is sufficient to produce a minimal specification. For this purpose, we introduce CSP_{BPEL}, an adaptation of CSP to make it fully compatible with describing BPEL systems. CSP_{BPEL} focuses on the trace behavior of BPEL process and abstracts the details of BPEL implementations.

CSP	CSP_{BPEL}
Process	Execution Trace
Event	Event.Start → Event.End
STOP	STOP
SKIP	SKIP
Sequence Operator (→)	Sequence Operator (→)
Interleaving (|||)	Parallel (||)
Choice	IF <exp> THEN Process ELSE Process
Recursion	WHILE <exp> Process

Table 9 is a shortlist of the language constructs taken from CSP and mapped to CSP_{BPEL} notation. Refer to [40] for the complete specifications of CSP.

CSP	CSP_{BPEL}
Process	Execution Trace
Event	Event.Start → Event.End
STOP	STOP
SKIP	SKIP
Sequence Operator (→)	Sequence Operator (→)
Interleaving (|||)	Parallel (||)
Choice	IF <exp> THEN Process ELSE Process
Recursion	WHILE <exp> Process

Table 9: CSP_{BPEL} Grammar

While some mappings between CSP and CSP_{BPEL} constructs are one-to-one however few are not and require further clarification:

- Event: a CSP Event is translated to a start and end event pair. Breaking down an event to start and end allows us to track the start and end of the activity execution (e.g. invoke activity) in a BPEL program; and therefore, enabling the comparison of sequencing of events in a BPEL process from a temporal perspective.

- Choice: choice in CSP is driven by events; that is the selection of branch is based on the occurrence of some event. In general this is not the case in BPEL where

branch selection is based on logical expressions. The result of a logical expression evaluation mostly depends on process data state.

- Recursion: formally, recursion in CSP is defined as $X = F(X)$ where $F(X)$ is a guarded expression containing the process X. Given that repetitive behavior in BPEL is further guarded by a logical expression, we refine the recursion definition to become $X = IF\ \langle exp \rangle\ THEN\ F(X)\ ELSE\ SKIP$. As short, we rewrite the expression as $WHILE \langle exp \rangle\ Process$ where $Process$ represents the prefix part of $F(X)$. Both expressions are semantically equivalent and lead to the same repetitive behavior under same conditions that is the same process data state.

3.3.2 Workflow Patterns

A workflow pattern is an abstract definition and representation of recurring business processes having similar traits and characteristics. A workflow pattern is abstract in the sense that it does not provide actual implementation of the pattern through predefined notations but rather provides a guideline. The realization of a workflow pattern varies from one modeling language to another given that the language provides the proper constructs to support it. [51] presents an analysis of workflow patterns supported by BPEL. We use the patterns supported by BPEL as a starting point to investigate how faults can manifest themselves within each workflow pattern.

Pattern	Standard		
	BPEL	XLANG	WSFL
Sequence	+	+	+
Parallel Split	+	+	+
Synchronization	+	+	+
Exclusive Choice	+	+	+
Simple Merge	+	+	+
Multi Choice	+	-	+
Synchronization Merge	+	-	+
Multi Merge	-	-	-
Discriminator	-	-	-
Arbitrary Cycles	-	-	-
Implicit Termination	+	-	+
MI without Synchronization	+	+	+
MI with Priori Design Time	+	+	+

Knowledge			
MI with Priori Runtime Knowledge	-	-	-
MI without a Priori Runtime Knowledge	-	-	-
Deferred Choice	+	+	-
Interleaved Parallel Routing	+/-	-	-
Milestone	-	-	-
Cancel Activity	+	+	+
Cancel Case	+	+	+

Table 10 summarizes the list of patterns analyzed by Wohed (2002). A '+' refers to direct support (i.e. there is a construct in the language which directly supports the pattern) while '-' refers to no direct support. A partial support of a pattern is indicated by '+/-'.

Pattern	Standard		
	BPEL	XLANG	WSFL
Sequence	+	+	+
Parallel Split	+	+	+
Synchronization	+	+	+
Exclusive Choice	+	+	+
Simple Merge	+	+	+
Multi Choice	+	-	+
Synchronization Merge	+	-	+
Multi Merge	-	-	-
Discriminator	-	-	-
Arbitrary Cycles	-	-	-
Implicit Termination	+	-	+
MI without Synchronization	+	+	+
MI with Priori Design Time Knowledge	+	+	+
MI with Priori Runtime Knowledge	-	-	-
MI without a Priori Runtime Knowledge	-	-	-
Deferred Choice	+	+	-
Interleaved Parallel Routing	+/-	-	-
Milestone	-	-	-
Cancel Activity	+	+	+
Cancel Case	+	+	+

Table 10: Standards Support for Workflow Patterns

We only consider workflow patterns that are supported by BPEL. Therefore, we eliminate patterns that are either not supported or indirectly supported by BPEL. A closer look to table II shows that BPEL supports a pattern if it is supported by either XLANG or WSFL standard. This is due to the fact that BPEL constructs are a hybrid of XLANG and

WSFL constructs. A clear implication of this is that although our work specifically addresses BPEL, adapting the work to cover XLANG and WSFL is straightforward.

3.3.3 Temporal Fault Models

Table 11 lists the temporal fault models for workflow patterns supported by BPEL. Both the workflow patterns and fault models are described in CSP_{BPEL}.

Pattern\Fault Model		**CSP_{BPEL}**
Sequence		activityA→activityB→SKIP
FM1	Sequence to Parallel	(activityA→SKIP) ‖ (activityB→SKIP)
FM2	Unexpected Termination	activityA→STOP or activityA→activityB→STOP or STOP
FM3	Switch Activities	activityB→activityA→SKIP
FM4	Deadlock Mutant	activityA→activityB→SKIP ‖ activityB→activityA→SKIP
Parallel Split		(activityA→SKIP) ‖ (activityB→SKIP)
FM5	Parallel to Sequence	activityA→activityB→SKIP or activityB→activityA→SKIP
FM6	Unexpected Termination	activityA→STOP or activityB→STOP or STOP
Synchronization		(activityA1→SKIP) ‖ (activityA2→SKIP) →activityB
FM7	Synchronization to Sequence	activityA1→activityA2→activityB→SKIP
FM8	Synchronization to Parallel	(activityA1→SKIP) ‖ (activityA2→SKIP) ‖ (activityB→SKIP)
FM9	Extra Single Condition	(activityA1→SKIP) ‖ (activityA2→SKIP) → IF(C1){activityB→SKIP}
FM10	Extra OR Condition	(activityA1→SKIP) ‖ (activityA2→SKIP) →IF(C1 | C2) {activityB→SKIP}
FM11	Extra AND Condition	(activityA1→SKIP) ‖ (activityA2→SKIP) →IF(C1 & C2) {activityB→SKIP}
Exclusive Choice		activityA1→ IF (C1) {activityA2→SKIP} ELSE {activityA3→SKIP}
FM12	Conditional Branches Switch	activityA1→ IF (C1) {activityA3→SKIP} ELSE {activityA2→SKIP}
FM13	Exclusive Choice to Parallel	activityA1→(activityA2→SKIP) ‖ (activityA3→SKIP)
Simple Merge		IF (C1) {activityA1→SKIP} ELSE {IF (C2) {activityA2→SKIP}}
FM14	Switch Condition	IF (C1) {activityA2→SKIP} ELSE {IF (C2) {activityA1→SKIP}}

FM15	Invalid Condition	IF (C1*) {activityA1→SKIP} ELSE {IF (C2*) {activityA2→SKIP}}
Multi-Choice		activityA1→(IF (C1){activityA2→SKIP} ‖ IF (C2){activityA3→SKIP})
FM16	Switch Condition	activityA1→ (IF (C2){activityA2→SKIP} ‖ IF (C1){activityA3→SKIP})
FM17	Multi-Choice to Simple Merge	activityA1→ (IF (C1){activityA2→SKIP} ‖ IF (C2*){activityA3→SKIP})
FM18	Multi-Choice to Parallel	activityA1→ ((activityA2→SKIP) ‖ (activityA3→SKIP))
Synchronization Merge		(IF (C1){activityA1→SKIP} ‖ IF (C2){activityA2→SKIP})→ IF(C1 | C2) {activityC→SKIP}ELSE{SKIP}
FM19	Synchronization with Extra AND Condition	(IF (C1){activityA1→SKIP} ‖ IF (C2){activityA2→SKIP})→ IF(C1 & C2) activityC→SKIP}ELSE{SKIP}
Deferred Choice		WHILE(BPEL_WAIT_Name < P3DT10H) {IF(BPEL_Message == M1){activityA→ SKIP} ELSE { IF(BPEL_Message == M2){acitivytB→SKIP}} → IF(BPEL_WAIT_Name >= P3DT10H) {activityC→ SKIP}
FM20	Missing Alarm	WHILE(true) {IF(BPEL_Message == M1){activityA→ SKIP } ELSE { IF(BPEL_Message= M2){acitivytB→ SKIP }}
Cancel Activity		activityB.Start→IF(BPEL_Fault == true){ IF((BPEL_FaultName == ”Fault1”) & (BPEL_FaultVar == ”Var1”))}{activityA→STOP}ELSE{activityB.End→SKIP}}
FM21	Incorrect Fault Matching	IF(BPEL_Fault == true){IF((BPEL_FaultName == ”Fault2”) & (BPEL_FaultVar == ”Var2”)) {activityA→STOP}ELSE{activityB.End→SKIP}}
Cancel Case		Process→STOP
FM22	Unexpected Termination	Process*→STOP

Table 11: Fault Models

Following is a brief description for each of the fault models:

Sequence to Parallel (FM1): Two activities that are required to execute in sequence are instead executed in parallel. This type of fault is caused by incorrectly using a *flow* activity instead of a *sequence* activity.

Unexpected Termination (FM2): At most one activity is executed. This type of fault is caused by the failure of an activity or inadvertently introducing a *terminate* activity.

Switch Activities (FM3): Activities are executed out-of-order. This fault is generated by misplacing activityA after activityB.

Deadlock Mutant (FM4): Two executing activities are waiting for each other to complete. E.g. a *sequence* activity with a link construct used incorrectly introducing a deadlock.

Parallel to Sequence (FM5): Two activities that are supposed to execute concurrently actually execute in sequence. This type of fault is caused by misusing a *sequence* activity instead of a *flow* activity.

Unexpected Termination (FM6): At most one activity is executed. This fault model is similar to FM2.

Synchronization to Sequence (FM7): Two activities required to execute in parallel execute in sequence. This fault is caused by omitting the *Parallel* activity between activityA and activityB.

Synchronization to Parallel (FM8): This fault model indicates that all three activities execute in parallel. This failure may be caused by placing all the activities in the *flow* activity.

Extra Single Condition (FM9): In this fault model, an extra condition is applied to the execution of activityB after the process reaches the synchronization point. ActivityB is supposed to execute unconditionally.

Extra OR Condition (FM10): In this fault model, two extra conditions are applied to the execution of activityB after the process reaches the synchronization point. The two conditions have an OR relationship. The cause for this fault model is an extra transition condition is applied to each activity and the relationship between the conditions is set to OR.

Extra AND Condition (FM11): This fault model is quite similar to FM10. The only difference is that in this fault model the relationship between conditions is AND rather than OR.

Conditional Branches Switch (FM12): In this fault mode, activityA3 executes when condition C1 is true, while it is expected to run when condition C1 is false. Analogously,

activityA2 executes when condition C1 is false, while it is expected to run when condition C1 is true. This fault model is caused by incorrectly reversing the link names or transition conditions between two source elements of activityA1.

Exclusive Choice to Parallel (FM13): In this fault model, the conditions used to determine the selection of branch activities are missed. After activityA1 executes, the other two activities are executed in parallel. The cause for this fault model is that the transition conditions in both source elements are not provided.

Switch Condition (FM14): Reversing activities activityA1 and activityA2 causes this fault model. In this fault mode, activityA2 executes instead of activityA1 when condition C1 is met.

Invalid Condition (FM15): This fault occurs when the conditions for the activities, activityA1 and activityA2, are modified incorrectly. Under certain circumstances, the execution trace of the activities for a faulty process doesn't change and therefore the fault is not detected.

Switch Condition (FM16): This fault model is quite similar to FM12. Both are concerned with switching the conditions between two activities.

Multi-Choice to Simple Merge (FM17): In this fault model, the condition applied to activityA3 is modified so that conditions C1 and C3 cannot be true at the same time. Since the conditions cannot be true at the same time, activityA2 and activityA3 won't execute at the same time.

Multi-Choice to Parallel (FM18): This fault model is quite similar to FM13. Both concurrent activities are missing their conditions for conditional activities.

Synchronization Merge with Extra AND Condition (FM19): In this fault model, if the relationship between the conditions C1 and C2 for activityC changes from OR to AND, activityC will not execute until both conditions C1 and C2 are satisfied.

Missing Alarm (FM20): The BPEL process is missing an alarm and will be idle forever waiting for an expected message that is dropped.

Incorrect Fault Matching (FM21): The fault model occurs when specifying either an invalid fault type or invalid variable name in a fault handler.

Unexpected Termination (FM22): A termination happens unexpectedly. An instance of a BPEL process execution stops when a termination activity is encountered.

3.4 BPEL Runtime Verification Platform (RV-BPEL)

A runtime monitoring system is a software system that observes the behavior of another software system and determines its consistency with a given specification. Monitoring is concerned with actual transitions between states and not the possible transitions. It takes an executing software system and a specification of the software properties and validates that the execution meets the properties. Generally speaking, there are two types of properties: safety and temporal. Some examples of safety properties but not limited to: invariants, sequence of events definitions, and resource allocation. On the other hand, temporal properties include timing, progress and bounded liveness [53]. Sometimes the distinction between the two categories of properties is not clear. For example, a property of a system "Event A occurs within 10 seconds" can be classified as both safety and temporal properties.

In this section we introduce a platform for runtime monitoring and verification of BPEL programs (RV-BPEL) for the purpose of detecting temporal faults. We further analyze the efficiency of the platform in discovering temporal faults by putting it under test based on the classification of temporal fault models and their representation using the notational set.

3.4.1 Specification Language Properties

The runtime monitoring system checks various safety and temporal properties. Hence, we investigate and identify the properties that are to be covered by our specification language [50]. The safety properties we consider are: deadlock free, invariants, properties that define a sequence of events, properties that check values of variables and properties that deal with resource allocation.

3.4.1.1 Property of Deadlock Free

The very basic safety property of a BPEL process is the property of deadlock free mainly because of its impact on other properties. If for any reason a deadlock occurs, the process is blocked and no further properties in the process are exercised. Most existing model-checking methods for verifying BPEL processes check the property of deadlock free. Although all BPEL processes require this property, we do not need to specify it through the specification language explicitly. The reason for this decision is that deadlock free is a common property for all instances of a process rather than a property for a specific instance and it is not valuable to repeatedly specify this property for every process. Therefore, we treat the property as an implicit one within a specification. It means every specification of properties specifies a deadlock free property, although it is not declared explicitly. If the execution of a BPEL process satisfies the property specification, then it has the property of deadlock free.

3.4.1.2 Invariants and Check Values of Variables Properties

An invariant is an expression whose value does not change during the program execution. In general, in object oriented programming there are two types of invariants: class invariants and loop invariants. A class invariant is an invariant used to constrain objects of a class. Methods of the class should preserve the invariant. The class invariant constrains the state stored in the object. Although BPEL does not have the concept of class, however we can easily map this concept to process invariant. Therefore, we think of class invariants as process invariants, which are used to constrain states in an instance of business process. Again in object oriented programming, there is the concept of a loop invariant to constrain the properties of loops. BPEL has a loop structure that is subject to loop invariant properties. When we take a close look at both invariants, process and loop, we find that they both are Boolean expressions built from other variables in the BPEL program. Therefore, we combine both invariants and check values of variables properties into one property.

3.4.1.3 Defining a Sequence of Events Properties

In the context of BPEL, the properties for defining a sequence of events refers to the properties for defining a sequence of activity events. We use activity events to represent

all the events in BPEL since it is a fundamental construct. We believe that this property is very critical for BPEL processes. Other domains, such as Java, have emphasized this importance [54], [55]. More importantly, one of the major goals of BPEL is to describe the order of execution of activities, which eventually defines the business process logic [56]. That being said, we conclude that the execution order of activities is crucial in BPEL programs. Therefore, the concept of execution order of activities can be translated into an ordered set of activity events, and the order or sequence of the activity events is critical to the runtime monitoring of BPEL.

Based on the analysis of properties in the context of BPEL, we conclude that our runtime monitoring system should provide the facilities to explicitly define properties that check the value of variables and that define a sequence of events. When we take a closer look at these properties, we find out that in many cases the sequencing of events is determined by the value of variables. Therefore when verifying the sequencing of events it is also important to account for the values of variables. For example, the *transitionCondtion* attribute of the *source* element for a *link* construct is a logical expression constructed from other BPEL process variables. The evaluation of the expression eventually controls whether the target element of the link is executed. Similarly, the condition attribute for a <while> activity, which is also a logical expression composed of process variables, determines whether the <while> activity is exited.

3.4.2 Verification Process

RV-BPEL's main responsibility is to verify that the actual execution order of activities satisfies the expected behavior specified via the trace specification. The overall verification process performed by the monitoring system is depicted in Figure 2.

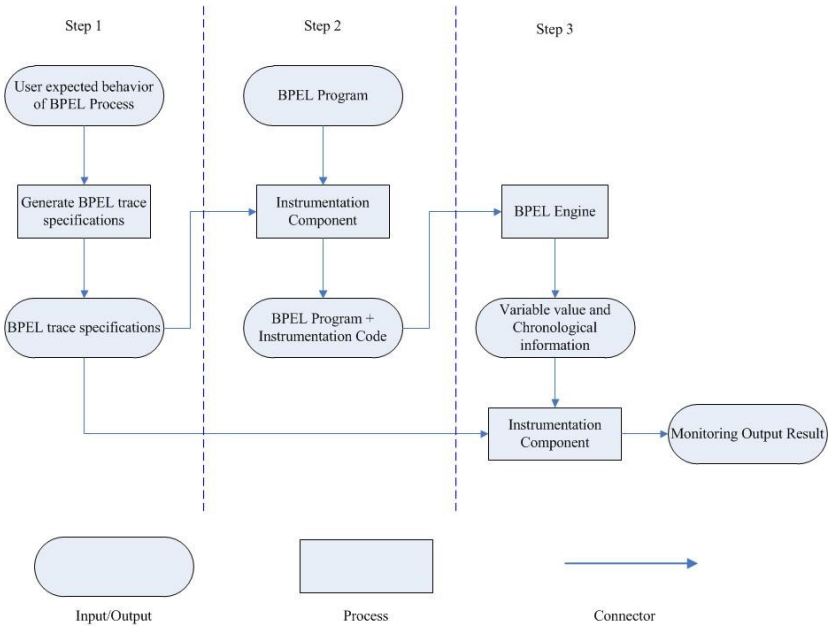

Figure 2: RV-BPEL Validation Platform.

The verification process is comprised of three main steps:

1. A BPEL trace specification definition: the dynamic behavior of a BPEL program is defined in CSP_{BPEL}. Given that the trace specification must describe the expected behavior of the system, it is built upon the requirements themselves.

2. Program Instrumentation: the monitoring system injects instrumentation code into the BPEL program using the trace specification generated in step 1. The purpose of instrumentation code is to collect process related runtime data.

3. Trace verification: performed by a specialized subsystem referred to as the *Trace Verification Subsystem*, the subsystem parses the BPEL trace specification to identify the expected temporal order of activities. The expected temporal order is then compared to the actual temporal order to identify abnormalities.

3.4.3 Platform Implementation

The RV-BPEL implementation implements the process defined in the previous section. We leverage JavaCC [57], a compiler-compiler, to automatically generate a compiler from the CSP_{BPEL} grammar. JavaCC is a powerful tool that provides a set of capabilities such as tree building via a tool called JJTree, actions, debugging and others. JavaCC

enables us to embed actions in the grammar to enable insertion of the monitoring code while parsing.

JavaCC supports bnf_production that has some discrepancies with the classical BNF notation. Nevertheless, bnf_production is very similar to BNF. The main difference between bnf_production and standard BNF is that you can embed actions in bnf_production. For example, a BNF specification is:

```
ProcessParallelExpression::=  ProcessPrefixExpression (
<TRACE_PARALLEL_OPERATOR> ProcessPrefixExpression )*
```

is translated into the following bnf_production code:

```
void ProcessParallelExpression():
{}
{(ProcessChoiceExpression() ( <TRACE_PARALLEL_OPERATOR>
ProcessChoiceExpression() )*) }
```

The system is fed the original BPEL file injecting the monitoring code from which a DOM object is constructed. Once the code is augmented with instrumentation symbols, the modified code is executed using ActiveBPEL, our choice for BPEL engine. The BPEL engine is extensible by providing mechanisms to add custom functionalities to the engine and therefore extending the behavior of the engine. These custom functionalities are in the form of methods that are plugged into the engine and are callable by the monitoring system from within the BPEL program.

Time recording is a determining factor in identifying the temporal order of activity execution within a BPEL program. BPEL lacks mechanisms to record the execution time for activities hindering the retrieval of certain information such as the start and the end of an activity execution. To accommodate for this limitation, part of the instrumentation code is responsible for recording start and end timestamps for each activity execution. This is realized by injecting an assign activity right before the activity construct which captures the value of a system timestamp. Similarly, we inject an assign activity right after the activity construct.

3.4.4 RV-BPEL in Action

In this section, we provide two examples to demonstrate the operation of the monitoring system. In an attempt to illustrate the definitions in Section 3, the examples utilize several workflow patterns. The BPEL processes were executed under the supervision of the runtime monitoring system to demonstrate its fault detection effectiveness. The next flowchart (Figure 3) describes the different steps in the experiment.

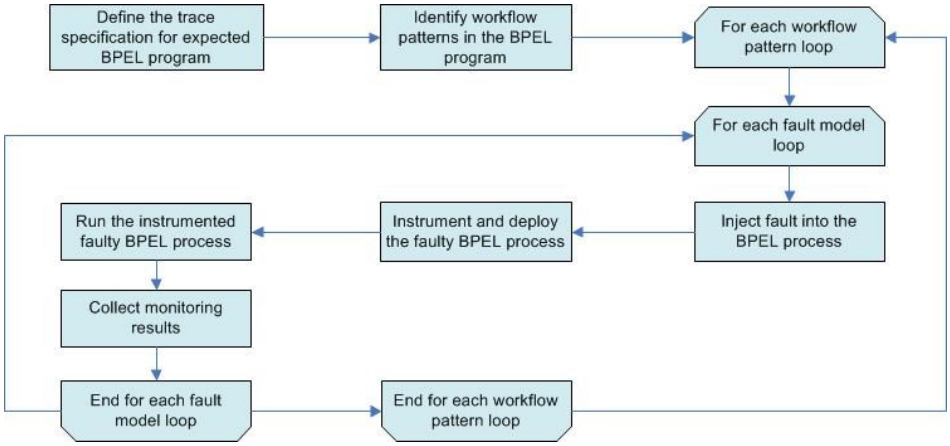

Figure 3: Experiment Steps

Following is a description of each step:

1. Define the trace specification of the experimental process: The expected trace behavior of the experimental process is generated as a trace specification in the form of CSP_{BPEL}. The trace specification is used to control the instrumentation of the BPEL program. Furthermore, it is translated into a series of events; later these events are compared with the actual events recorded during execution to determine if there is an error in the execution.

2. Analyze the trace specification and identify the workflow patterns: The trace specification is parsed to identify the different workflow patterns. The purpose of this step is to determine the applicable fault models that can be injected into the BPEL program.

3. Inject a temporal fault into the BPEL program: Applicable faults are derived from the specifications listed in Section 3. Each fault is injected into the process one at

a time producing a fault program. Since each workflow pattern has one or more fault models; this step is repeated for each fault model.

4. Add instrumentation code to the faulty process: The trace specification is written to a text file which is used as an input to the instrumentation system. Using the trace specification and faulty processes, the instrumentation system injects the instrumentation test code into each faulty process. After completing the instrumentation phase, the filename of the faulty process is not changed.

5. Deploy and execute faulty process: After completing the instrumentation, the faulty process is deployed into the BPEL engine. A client program is used to invoke the execution of the instrumented faulty process.

6. Analyze and evaluate the result of execution: During the execution of the faulty process, the test results are written into log files by the trace verification system. The test results are then analyzed to evaluate the "effectiveness" of the proposed BPEL runtime monitoring system. The effectiveness of the runtime monitoring system is proportional to the number of detected faults.

Figure 4 illustrates the steps executed by RV-BPEL to identify temporal faults of type **FM1**. This fault is caused by mistakenly changing the relationship between the two activities, *Activity A* and *Activity B* from sequential to a concurrent relationship.

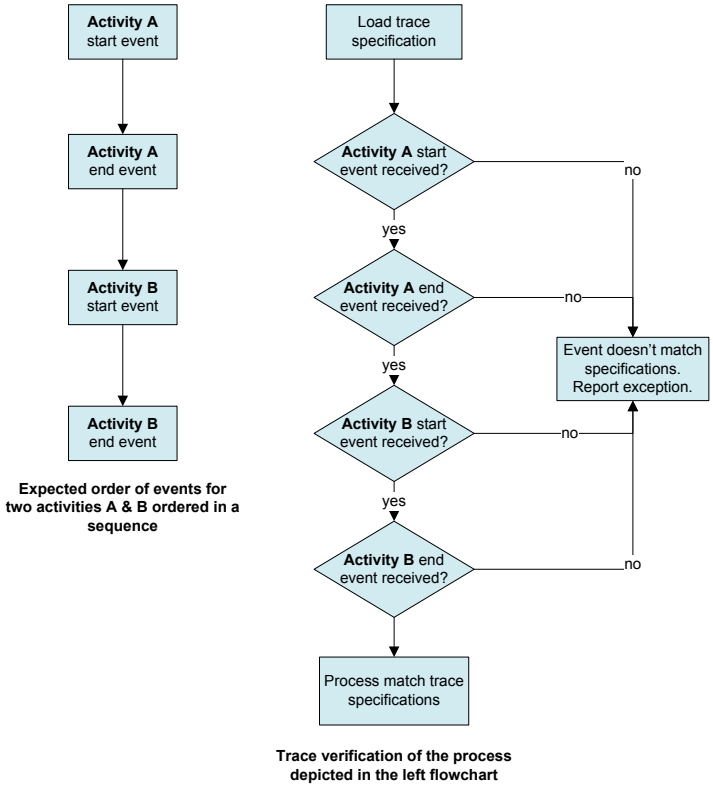

Figure 4: RV-BPEL Process to Capture FM1

The figure depicts two flowcharts: the one to the left is the expected order of events for a sequence pattern while the one to the right describes the steps taken by RV-BPEL to verify the runtime trace match the actual trace specification. Any violation to the trace specification is captured and logged by the monitoring system.

3.4.4.1 Example I

For this experiment, we utilize the Purchase Order Process presented in [45]. Upon receiving the purchase order from a customer, the process initiates three tasks concurrently: calculating the final price for the order, selecting a shipper, and scheduling the production and shipment for the order (refer to Figure 5). While some of the processing can proceed concurrently, there are control and data dependencies between the three tasks (depicted as solid lines). In particular, the shipping price is required to finalize the price calculation, and the shipping date is required for the complete fulfillment schedule. When the three tasks are completed, invoice processing can proceed and the invoice is sent to the customer.

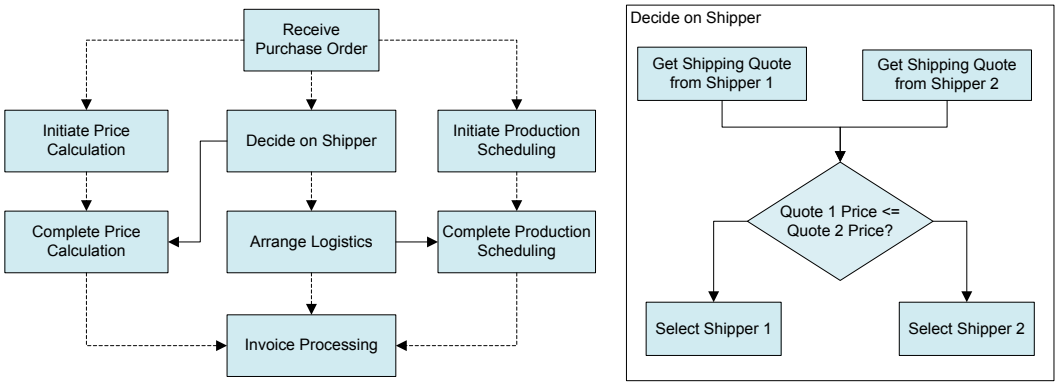

Figure 5: Purchase Order Process

LISTING 1. Trace Specification

receivePurchaseOrder → (((quoteShipper1→ SKIP) ‖ (quoteShipper2 → SKIP)) → IF(shipperInfo1.price <= shipperInfo2.price) {useShipper1} ELSE {useShipeer2} → ((checkBalance → SKIP) ‖ (checkCredit → SKIP)) → IF(totalCharge.number < totalCredit.number) {payWithCredit } ELSE IF(totalCharge.number<totalBalance.number) {payWithBalance } ELSE {errorPay } → ArrangeLogistics → SKIP) ‖ (InitiatePirceCalculation → CompletePriceCalculation → SKIP) ‖ (InitiateProducationScheduling→ CompleleteProductionScheduling → SKIP)) → (IF(po.needPaperInvoce == ”yes”) {sendPaperInvoice → SKIP}) ‖ (IF(po.needElectronicInvoice == ”yes”) {sendElectronicInvoice → SKIP}) → replyPurchaseOrder → STOP

The first step in the experiment is defining the trace specification for the process (refer to Listing 1). The next step is identifying the workflow patterns for the purpose of designing and introducing the faults. The workflow patterns implemented in the process are: Sequence, Parallel Split, Synchronization, Simple Merge and Multi-Choice and Synchronizing Merge (Figure 5). Accordingly, the following faults are introduced as mutants to the process: FM1, FM2, FM3, FM4, FM5, FM6, FM8, FM9, FM10, FM11, FM14, FM15 and FM16. For each fault model, we generate one fault according to the mutation process defined in Section 3. Moreover, we further generated five independent faults using frequently used mutant operators for general programming languages. The mutant operators are “Mathematics operators exchanged”, “Variable by variable replacement”, “Increment/decrement variables/constants” and “Output missing” [58]. The details of the mutants are shown in Table 1 - Appendix C – BPEL Mutants. The main purpose of the second set of mutants is to crosscheck that the system does not returns

false positives or inappropriately terminate in the presence of defects that are not of a temporal nature. In total eighteen mutants are generated. Four test cases are created manually in the form of .NET console applications that interact with the BPEL process. The test cases provide 100% path coverage and exercise all possible scenarios as dictated by the requirements. Subsequently, we executed the test cases against the faulty BPEL processes. Table IV summarizes the results - the number of faults killed by each test case.

As a result of executing the test cases, all of the temporal faults were killed and one general mutant was killed by serendipity. Since 100% of the temporal faults were killed we conclude that RV-BPEL performs well in uncovering defects of temporal nature.

Test Case	# Kills of Temporal Mutants	# Kills of Generic Mutants
#1	10	0
#2	10	1
#3	8	1
#4	10	0

Table 12: Mutants per Test Killing Rate

3.4.4.2 Example II

Experiment II demonstrates the fault models for workflow patterns that are not covered by experiment I. Further, wherever possible, experiment II uses different variations of implementations of the workflow patterns available in experiment I. In this scenario, we developed an auto insurance claim processing application (Figure 6). Upon receiving the claim for a car accident, the process calculates the total expense of the accident, including the auto repairing expense and the victim's medical expense. Depending on the total expense and the police report of the accident, the process invokes the appropriate external Web service to calculate the new insurance rate. Next, the payment is made through direct deposit or a mailing check. Finally, a report is sent back to the client. The trace specification is shown in Listing 2.

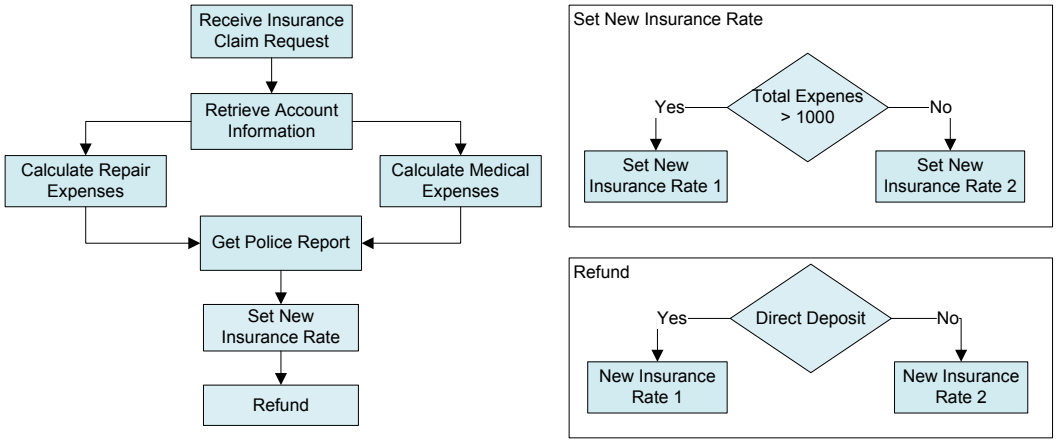

Figure 6: Claim Insurance Process

LISTING 2. Auto Insurance Claim Process Trace Specification

ReceiveInsuranceClaim → retrieveAccountInfo → ((hospitalExpense → SKIP) || (repairExpense → SKIP)) → getPoliceReport → readPoliceReport → ((IF(claim.totalExpense>1000){ newInsuranceRate1 → SKIP }) || (IF((claim.totalExpense <= 1000) & (claim.totalExpense > 200)) {newInsuranceRate2 → SKIP }))
→ ((IF(claim.directDeposite == "yes") {directDeposit → SKIP }) || (IF(claim.directDeposite != "yes") {mailCheck → SKIP })) → STOP

The workflow patterns covered by this experiment are Sequence, Synchronization, Simple Merge, Exclusive Choice, Deferred Choice and Cancel Activity (depicted in Figure 5). In total, we generated 10 faults based on the fault models for the identified workflow patterns. The fault models are: FM1, FM2, FM4, FM6, FM7, FM11, FM12, FM13, FM17 and FM18. In addition, we generated 5 further "generic" faults. The details of all the faults are given in Table 2 - Appendix C – BPEL Mutants.

Table 13: Mutants per Test Killing RateTable 13 lists the results - the number of faults detected by each test case. Again, all of the workflow faults were killed and this time 2 of the 5 generic mutants were also killed by serendipity. Again RV-BPEL was successful in achieving 100% coverage of temporal faults. Similar to experiment I, the system failed to catch most of the common programming faults (only 40%).

Test Case	# Kills of Temporal	# Kills of Generic

	Mutants	Mutants
#1	9	1
#2	12	1
#3	8	1
#4	7	1

Table 13: Mutants per Test Killing Rate

3.5 Related Work

The testing and verification of BPEL systems is an ongoing research area. Several approaches have been proposed to test and increase the stability of such systems. [59] introduce BP-calculus, a process algebra based on pi-calculus, for the formal modeling and generation of WS-BPEL programs. The calculus is used to perform formal verification using formulae expressed in pi-logic and the HAL (HD-Automata Laboratory) Toolkit. A mapping from BP-calculus to WS-BPEL is provided for the automated generation of WS-BPEL code from its calculus. Compared to CSP_{BPEL}, BP-Calculus is more complex grammar and its verification is less practical for larger BPEL systems due to the state space explosion problem. [60] introduces a method to verify business processes with SPIN. In this method, the BPEL specification is firstly translated into an intermediate representation Guarded Automata. The automata are then translated into PROMELA. Because the guards in the guarded automata are expressed as XPath expressions, this enables the verification of the XML data manipulation properties. Furthermore, it proposes the concept of synchronization to tackle one inherent limitation of SPIN which can only achieve partial verification by fixing the size of communication channels in PROMELA. While the work addresses the verification of temporal properties, however it does not investigate the nature of temporal faults contrary to our work. [61] explains how to use activity diagrams to capture the execution flow of a BPEL program. The resultant UML activity model is transformed to PROMELA based on a set of mapping rules. The paper does not provide any details about the type of faults that can be investigated. Test case generation based on CP-net is proposed by [62] to test concurrency aspects of BPEL programs. The approach relies on translating BPEL program into CP-net model followed by the generation and execution of test cases. The authors claim that the resultant CP-net model tackles state space explosion and results in the generation of fewer test cases. While the paper emphasizes the reduction in number of

generated test cases, however it fails to demonstrate the effectiveness of the generated test cases and the nature of targeted faults.

So far, all approaches discussed are based on model checking. Because model checking is based on formal methods, they all share one characteristic, they are not easy to understand and use by the average practitioner. Moreover, the large size of the explicit representation of the state space of most systems severely limits the size of systems that can be model checked. Although state reduction techniques have been proposed to reduce the state space explosion, fully describing and verifying a system is still extremely difficult. In addition, model checking conducts static checks of a business process, which may not fully considered the value and timing characteristics of the process. The massive number of runtime interactions that connect various components is what makes web application reliability a challenging task. Similarly, for business processes composed of various Web services, the runtime interactions between web services are critical to the reliability of the business process. Hence we cannot ignore those significant factors and it is clearly of valuable to investigate verifying these business processes from the perspective of run time monitoring.

Closely related to our work, [63], [36] introduce a set of mutation operators for WS-BPEL programs and the means to minimize the computational cost of mutation testing. The former uses genetic algorithm (GA) and describes a framework for generating and reducing the mutants to a minimal set. The latter introduce a set of metrics for evaluating objectively the quality of a set of mutants with respect to a test suite. The metrics are used to reduce the number of mutants and thus reducing the overall cost associated with mutation testing. While their work is similar to ours with respect to using mutation testing to test WS-BPEL, however we believe the set of mutation operators of temporal nature is not comprehensive. For example, the work ignores faults related to synchronization and presents only one fault related to sequence out of the four faults presented in our work. Moreover, their approach to identify mutants is not methodical and the steps to apply mutants on BPEL programs are not specified. We follow a more rigorous and comprehensive approach to identifying temporal faults by analyzing known workflow patterns. In addition, the authors neither provide a formal description of the

mutants nor describe the production rules to introduce mutants into BPEL systems. Whereas this paper proposes a formal specification to describe the faults, and the production rules required to inject mutants into BPEL systems. This enables the automation of detecting workflow patterns and fault injection. Another work in the area of mutation testing and BPEL is that of [64]. The work explores the applicability of the mutant operators for BPEL programs introduced by [63]. Mainly the authors investigate the operators in the context of weak mutation testing. They further evaluate the effectiveness of the operators in a complementary work where they introduce WeMuTe, a weak mutation testing tool for WS-BPEL. The work fails to describe the programs and the operators used in the experiments to test each program. Therefore, no conclusions can be drawn regarding the effectiveness of the mutants in evaluation the comprehensiveness of the test suite. Our empirical analysis involves providing a formal description of the programs used in the experiments and analyzes the applicable faults based on the detected workflow patterns. To alleviate the complexity of applying formal methods to reason about BPEL programs, [35] apply dynamic invariant generation to extract properties from actual executions of BPEL programs. The authors indicate that one possible usage of this approach is to detect bugs in compositions and improve test suites and hence can be used along to our approach.

In the area of runtime monitoring of BPEL systems, [65] presents an approach to support dynamic monitoring of WS-BPEL processes as a mechanism to assess and ensure the quality of running processes. The work proposes using monitoring rules that are weaved dynamically into the process to control the execution of WS-BPEL processes. The monitoring system adopts a proxy based approach to support dynamic selection of monitoring rules at runtime. A user-oriented language is defined to integrate data acquisition and analysis into monitoring rules. We believe that our fault models can be used in mutation testing to verify the effectiveness of the proposed monitoring system. Similarly, [66] introduce VieDAME a system which allows monitoring of BPEL processes and binding to web services at runtime to ensure certain Quality of Service (QoS) attributes. The system uses various replacement strategies to identify substitute services that are either syntactically or semantically equivalent. Aspect Oriented is employed to intercept SOAP messages and allow services to be exchanged during

runtime. By using VieDAME and RV-BPEL side-by-side, we can address both functional and non-functional aspects of BPEL program and thus providing a holistic monitoring solution. Another contribution in the area of BPEL reliability is [67], which proposes self-supervising BPEL processes that are capable of assessing their behavior and reacting accordingly by taking recovery actions if necessary. For this purpose, the authors suggest two XML based language, WSCoL (Web Service Constraint Language) to express assertions and WSRel (Web Service Recovery Language) to express recovery actions. The proposed approach does not address issues of temporal nature and focuses on assessing preconditions and expectations based on process state.

3.6 Conclusion

The testing and verification of BPEL systems is given considerable attention in the literature. Contrary to that, very limited research exists in the area of classification and identification of BPEL faults. To-date, no methodical approach is proposed for the identification of BPEL faults and more specifically temporal faults. The only work related to identifying BPEL mutation operators is that of [30]. However the list of mutation operators of temporal nature is far from complete. Moreover, their approach to identifying mutation operators is less methodical and a formal description of mutation operators is not provided. The lack of formal description hinders the automation of fault injection.

In this paper, we identify and categorize temporal faults in BPEL programs. We follow a methodical and disciplined approach in identifying temporal faults by analyzing well-known workflow patterns to derive the fault models. In total, we analyzed eleven workflow models and derived all major temporal defect types that are derivable from these fundamental definitions. We utilize a minimal set of CSP notations, $CSP_{BPEL,}$ to build the vocabulary needed to capture the specifications of BPEL programs. This minimal vocabulary simplifies the formal description of trace specifications. Furthermore, we have built a runtime verification platform for BPEL environments and demonstrated its effectiveness to detect and identify temporal faults. Specifically, our system is able to non-intrusively instrument and monitor the execution trace of BPEL systems and detect faults of temporal nature.

This research is a major step towards producing a comprehensive list of temporal fault models. Understanding the type of faults in BPEL programs advances the field of BPEL runtime monitoring and therefore contributes towards stabilizing such family of systems. Furthermore, the fault models can be used in mutation testing, a fault-testing technique to assess the effectiveness of test suites for the purpose of test optimization and cost reduction. Although our work specifically addresses BPEL, however the fault models are equally applicable to other workflow languages. The source code for RV-BPEL and the examples used in the experiments is available for download[2]. Future work includes identifying and classifying other fault models.

[2] http://www.steam.ualberta.ca/main/research_areas/temporalfaults.rar

4 Conclusion

In this research we advance the field of testing and verification of composite systems that are built on top of WS-CDL and WS-BPLE.

As for WS-CDL, we present Chor-calculus, a new formal language based on pi-calculus and the semantics of WS-CDL. A mapping between Chor-Calculus and WS-CDL is provided which allows the generation of WS-CDL programs from a Chor-calculus specification. A tool is developed that generates WS-CDL programs from a Chor-calculus specification. Furthermore, we demonstrate the use of the HAL toolkit to verify the correctness properties of Chor-calculus choreographies. One of the main objectives of WS-CDL is to be used along with other orchestrated systems such as WS-BPEL applications. We envisage using Chor-calculus with other calculi-based formal methods to investigate the compatibility of interacting services. For example, Chor-calculus can be used along with BP-calculus [5] and [27] to determine the behavioral service compatibility of existing systems.

As for WS-BPEL, we identify and categorize temporal faults in BPEL programs. We follow a methodical and disciplined approach in identifying temporal faults by analyzing well-known workflow patterns to derive the fault models. In total, we analyzed eleven workflow models and derived all major temporal defect types that are derivable from these fundamental definitions. We utilize a minimal set of CSP notations, $CSP_{BPEL,}$ to build the vocabulary needed to capture the specifications of BPEL programs. This minimal vocabulary simplifies the formal description of trace specifications. Furthermore, we have built a runtime verification platform for BPEL environments and demonstrated its effectiveness to detect and identify temporal faults. Specifically, our system is able to non-intrusively instrument and monitor the execution trace of BPEL systems and detect faults of temporal nature.

This research is a major step towards enhancing testing techniques of service compositions. The contribution helps in understanding and identifying faults that manifest in service composition programs by offering new means for writing effective test suites against WS-CDL and WS-BPEL programs. Moreover, the work for WS-BPEL

advances the field of BPEL runtime monitoring and therefore contributes towards stabilizing such family of systems.

References

[1] W. Tsai, "Service-oriented system engineering: a new paradigm", *In IEEE International Workshop on Service-Oriented Systems Engineering (SOSE) 2005*, pp. 3-6, Oct. 2005.

[2] T. Erl, *SOA Design Patterns*. Prentice Hall, Upper Saddle River, NJ, pp. 35-36, 2009.

[3] Oasis, "Web Services Business Process Execution Language Version 2.0", http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, 2007.

[4] N. Kavantzas, D. Burdett, G. Ritzinger, Y. Lafon, "Web Services Choreography Description Language Version 1.0", http://www.w3.org/TR/ws-cdl-10. 2005.

[5] F. Abouzaid, J. Mullins, "A Calculus for Generation, Verification and Refinement of BPEL Specifications", Electronic Notes in Theoratical Computer Science, pp. 43-65, 2008.

[6] M. Mazzara, R. Lucchi, "A Pi-calculus Based Semantics for WS-BPEL. Journal of Logic and Algebraic Programming", Journal of Logic and Algebraic Programming 70(1), pp. 96–118, 2006.

[7] M. ter. Beek, A. Bucchiarone, S. Gnesi, Web Service Composition Approaches: From Industrial Standards to Formal Methods. *Second International Conference on Internet and Web Applications and Services (ICIW '07)*, Mauritius, May 2007.

[8] F. Abouzaid, M. Mazzara, J. Mullins, N. Qamar, "Towards a Formal Analysis of Dynamic Reconfiguration in WS-BPEL", Intelligent Decision Technologies, vol. 7, pp. 213-224, 2013.

[9] N. Lohmannn, "A Feature-Complete Petri Net Semantics for WS-BPEL 2.0", Web Services and Formal Methods, Lecture Notes in Computer Science, vol. 4937, pp. 77-91, 2008.

[10] M. Bravetti, M. Núñez, and G. Zavattaro, "Web Services and Formal Methods", *Proceedngs of Third International Workshop, WS-FM 2006*, vol. 4184, Vienna, Austria, Sept. 2006.

[11] N. Dragoni and M. Mazzara, "A Formal Semantics for the WS-BPEL Recovery Framework: The pi-Calculus Way". In WS-FM, vol. 6194, pp. 92-109, Springer, 2010.

[12] M. Mazzara, "Towards Abstractions for Web Services Composition", PhD dissertation, Dept. of Computer Science, University of Bologna, Bologna, Italy, 2006.

[13] A. Barros, M. Dumas, P. Oaks, "A Critical Overview of the Web Services Choreography Description Language", BPTrends Newsletter, http://news.bptrends.com/publicationfiles/03-05 WP WS-CDL Barros et al.pdf. 2005.

[14] H. Ahn Le, N. Thuan Truong, "Modeling and Verifying WS-CDL Using Event-B", Context-Aware Systems and Applications, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 109, pp. 290-299, 2013.

[15] Wil M. P. van der Aalst, A.H. M. ter Hofstede, B. Kiepuszewski, A.P. Barros, "Workflow Patterns. Distributed and Parallel Databases", 14(1):5–51, 2003.

[16] A. Barros, M. Dumas, A. ter Hofstede, "Service Interaction Patterns", *In Proceedings 3rd International Conference on Business Process Management (BPM 2005)*, pp. 302–318, Nancy, France, 2005.

[17] G. Ferrari, G. Ferro, S. Gnesi, U. Montanari, M. Pistore and G. Ristori, "An Automata Based Verification Environment for Mobile Processes", *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97),* vol. 1217, pp. 275–289, 1997.

[18] G. Ferrari, S. Gnesi, U. Montanari, M. Pistore, "A Model-Checking Verification Environment for Mobile Processes", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 12, no. 4, pp. 440-473, Oct 2003, doi:10.1145/990010.990013.

[19] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, "Web Services Description Language (WSDL) 1.1", http://www.w3.org/TR/wsdl, March 2001.

[20] R. Chinnici, J. Moreau, A. Ryman, S. Weerawarana, “Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language”, http://www.w3.org/TR/wsdl20/, 2007.

[21] R. Milner, Communicating and Mobile Systems: The Pi-Calculus, Cambridge, UK:Cambridge University Press , 1999.

[22] G. Decker, H. Overdick, J. M. Zana, “On the Suitability of WS-CDL for Choreography Modeling”, Proceedings of Methoden, Konzepte und Technologien für die Entwicklung von dienstebasierten Informationssystemen, Hamburg, Germany, Oct 2006.

[23] S. Rajamani and J. Rehof, “A Behavioral Module System for the Pi-Calculus”, Lecture Notes in Computer Science vo. 2126, pp. 375-394, 2001.

[24] P. Wong and J. Gibbons, "A Process-Algebraic Approach to Workflow Specification and Refinement", Lectures Notes in Computer Science vo. 4829, pp. 51-65, 2007.

[25] T. Parr, “ANother Tool for Language Recognition”, http://www.antlr.org/, 2013.

[26] T. Parr, The Definitive Antlr 4 Reference, Dalas-Texas Raleigh, North Carolina:Pragmatic Bookshelf, 2nd edition, Jan. 2013.

[27] S. Deng, Z. Wu, M. Zhou, Y. Li, J. Wu, “Modeling Service Compatibility with Pi-calculus for Choreography”, Lecture Notes in Computer Science vo. 4215, pp. 26-39, 2006.

[28] A. Brogi, C. Canal, E. Pimentel, A. Vallecillo, “Formalizing Web Services Choreographies”, Electronic Notes in Theoretical Computer Science, v. 105, pp. 73–94, 2004.

[29] H. Yang, X. Zhao, Z. Qiu, G. Pu, S. Wang, “A Formal Model for Web Service Choreography Description Language (WS-CDL)”, *IEEE International Conference on Web Services (ICWS’06)*, pp. 893–894, 2006.

[30] M. Carbone, K. Honda, and N. Yoshida, “Structured Communication-Centred Programming for Web Services”, *Proceedings of ESOP’07*, vol. 4421, pp. 2–17, Springer-Verlag, 2007.

[31] M. Carbone, “Session-based Choreography with Exceptions”, *Electronic Notes Theoratical Computer Science*, vol. 241, pp. 35-55, 2009.

[32] I. Lanese, C. Guidi, F Montesi, G. Zavattaro, "Bridging the Gap between Interaction-and Process-Oriented Choreographies", *IEEE International Conference on Software Engineering and Formal Methods (SEFM'08)*, IEEE Computer Society, pp. 323-332, 2008.

[33] C. Guidi, "Formalizing languages for Service Oriented Computing", PhD dissertation, Dept. of Computer Science, University of Bologna, Bologna, Italy, 2007.

[34] G. Campos, N. Rosa, and L. Ferreira Pires. A Survey of Formalization Approaches to Service Composition. In *Proceedings of the 2014 IEEE International Conference on Services Computing*, Anchorage, AK, 179 – 186, 2014.

[35] M. Palomo-Duarte, A. García-Domínguez and I. Medina-Bulo. Automatic dynamic generation of likely invariants for WS-BPEL compositions. *Expert Systems with Applications*, v. 41, pp. 5041-5055, Sept. 2014.

[36] A. Estero-Botaro, F. Palomo-Lozano, I. Medina-Bulo, J. Domínguez-Jiménez and A. García-Domínguez. Quality metrics for mutation testing with applications to WS-BPEL compositions. *Software Testing, Verification and Reliability*, Wiley Online Library, 2014.

[37] M. Khaxar and S. Jalili. WSCMon: runtime monitoring of web service orchestration based on refinement checking. Service Oriented Computing Applications, Vol. 6. Springer-Verlag, 33-49, 2012.

[38] F. Belli, A. Endo, M. Linschulte, and A. Simao. A holistic approach to model-based testing of Web service compositions. Software: Practice and Experience, Vol. 44. Wiley Online Library, 201-234, 2014.

[39] A. Kocbek and M. Juric. Towards a Reusable Fault Handling in WS-BPEL, Int. J. Soft. Eng. Knowl. Eng, Vol. 24. World Scientific, 243-267, 2104.

[40] C.A.R Hoare. Communicating Sequential Processes, Retrieved February, 7, 2009, from http://www.usingcsp.com/cspbook.pdf.

[41] C.A.R Hoare. Communicating Sequential Processes, *Communications of ACM*, Vol. 21, 666-677, 1978.

[42] S.D. Brookes, C.A.R Hoare, and B. Roscoe. A Theory of Communicating Sequential Processes, *Journal of the ACM*, Vol. 31, 560-599, 1984.

[43] T. Earl. SOA Design Patterns, Prentice Hall, Boston, MA, USA, 2009.

[44] E. Newcomer and G. Lomow. *Understanding SOA with Web Services*, Addison Wesley, Boston, Massachusetts, USA, 2004.

[45] OASIS. Retrieved January 4, 2014, from http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[46] P. Sarang and M. Juric. *Business Process Execution Language for Web Services*, Pack Publishing Ltd., 32 Lincoln Road Olton Birmingham, B27 6PA, UK, 2006.

[47] R. Lipton and R. DeMillo Hints on Test Data Selection: Help for the Practicing Programmer. IEEE Computer, Vol. 11, 34-41, 1978.

[48] Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing, *IEEE Transactions on Software Engineering*, Vol. 37, 649-678, 2001.

[49] M. Roper. Software Testing. McGraw Hill International Software Quality Assurance, 87-88, 1994.

[50] J. Offutt. A Practical System for Mutation Testing: Help for the Common Programmer, *Proceedings of 12th International Conference on Testing Computer Software*, Washington, DC, 99-109, 1995.

[51] P. Wohed, W. M.P. van der Aalst, M. Dumas, and A. H.M. ter Hofstede. Pattern Based Analysis of BPEL4WS, *QUT Technical Report*, FIT-TR-2002-04, Queensland University of Technology, Brisbane, Australia, 2002.

[52] W.L. Yeung. Mapping WS-CDL and BPEL into CSP for Behavioral Specification and Verification of Web Services, *Proceedings of the 4th European Conference on Web Services (ECOWS'06)*, IEEE Computer Society, 297-305, 2006.

[53] N. Delgado, A. Gates, and S. Roach. A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools, *IEEE Transactions on Software Engineering*, Vol. 30, 859-872, 2004.

[54] M. Brorkens and M. Moller. Runtime Checking the Dynamic of Java Programs, *Proceedings of the IFIP 14th International Conference on Testing of Communicating Systems*, Berlin, Germany, 39-48, 2002.

[55] M. Moller. Specifying and Checking Java Using CSP, *In Workshop on Formal Techniques for Java-like Programs-FTfJP*, Technical Report NIII-R0204, Computing Science Department, University of Nijmegen, 1-9, 2002.

[56] C. Peltz. Retrieved May 24, 2014, from http://xml.coverpages.org/HP-WSOrchestration.pdf.

[57] JavaCC. Java Compiler Compiler. Retrieved April, 12, 2014 from http://javacc.java.net.

[58] S. do Rocio, S. de souza, J. Maldonado, S. Pinto, F. Fabbri, and W. de Souza. Mutation Testing Applied to Estelle Specifications, *Software Quality Journal*, Vol. 8, 285-301, 1999.

[59] F. Abouzaid and J Mullins. A Calculus for Generation, Verification and Refinement of BPEL Specifications, Electronic Notes in Theoretical Computer Science, 43-65, 2008.

[60] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services, *Proceedings of the 13th International Conference on World Wide Web*, New York, NY, USA, 2004, 621-630, 2004.

[61] H. Cao, S. Ying, and D. Du. Towards Model-Based Verification of BPEL with Model Checking, *Proceedings of the 6th IEEE International Conference on Computer and Information Technology*, Seoul, Korea, Sept. 190-194, 2006.

[62] Y. Wang and N. Yang. Test Case Generation of Web Service Composition based on CP-nets, Journal of Software, March 2014, v.9, pp. 589-595, 2014.

[63] J. Domınguez-Jimenez, A. Estero-Botaro, and I. Medina-Bulo. A Framework for Mutant Genetic Generation for WS-BPEL, *Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science*, Spindleruv Mlyn, Czech Republic, Vol. 5404, 229-240, 2009.

[64] P. Boonyakulsrirung and T. Suwannasart. A Weak Mutation Testing framework for WS-BPEL, *Proceedings of the 8th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, Thailand, 313-318, 2011.

[65] L. Baresi and S. Guinea. Towards Dynamic Monitoring of WS-BPEL Processes. *Proceedings of the 3rd International Conference of Service-Oriented Computing (ICSOC'05)*, Amsterdam, 269-282, 2005.

[66] O. Moser, F. Rosenberg and S. Dustdar. Non-Intrusive Monitoring and Service Adaptation for WS-BPEL, *Proceedings of the 17th International Conference on World Wide Web*, New York, USA, 815-824, 2008.

[67] L. Baresi and S. Guinea. Self-Supervising BPEL Processes. *IEEE Transactions on Software Engineering*, 247-263, 2011.

[68] M.P. Papazoglou, D. Georgakopoulos. Service-Oriented Computing, *Communications of the ACM*, Vol. 46, no. 10, 25-2, 2003

Appendix A – Chor-calculus

Agent	Chor-calculus
PC	$PC = ((v\ shipmentChoreography^{c}, in_{eb},\ un_{eb}, in_{fb}, un_{fb}, y_{fb}, y_{fc}, x_{com})\{P, NR, H\})$
P	$P = \overline{purchaseOrder^{rqrs}}\langle \tilde{x} \rangle.0 \parallel \overline{shipmentChoreography^{pr}}\langle \tilde{y} \rangle.0$
NR	$NR = \left(!\, shipmentChoreography^{c}(\tilde{z}).\left((v\ in_{eb},\ un_{eb}, in_{fb}, un_{fb}, y_{fb}, y_{fc}, x_{com})\{Q, NR_1, H_1\}\right)\right)$
H	$H = W'_{EB}(\widehat{A_{eb}}) \mid W'_{FB}(\widehat{A_{fb}})$
WEB_PC	$W'_{EB}(\widehat{A_{eb}}) = in_{eb}().\left((invalidOrder^{handle}(orderDetails).R \parallel \bar{y}_{eb}\langle\rangle.0) + un_{eb}()\right)$
WFB_PC	$W'_{FB}(\widehat{A_{fb}}) = in_{fb}().\overline{un}_{fb}\langle\rangle.0$
QC	$(v\ in_{eb},\ un_{eb}, in_{fb}, un_{fb}, y_{fb}, y_{fc}, x_{com})\{Q, NR_1, H_1\}$
Q	$Q = \overline{shipOrder^{rq}}\langle \tilde{u} \rangle.0 \parallel \overline{shipmentConfirmation^{rs}}\langle \tilde{v} \rangle.0$
H1	$H_1 = W''_{EB}(\widehat{A_{eb}}) \mid W''_{FB}(\widehat{A_{fb}})$
WEB_QC	$W''_{EB}(\widehat{A_{eb}}) = in_{eb}().un_{eb}().0$
WFB_QC	$W''_{FB}(\widehat{A_{fb}}) = in_{fb}().\overline{un}_{fb}\langle\rangle.0$
E	**PurchaseOrder** $= !((\mathbf{v\ shipmentChoreography^{c}, in_{eb},\ un_{eb}, in_{fb}, un_{fb}, y_{fb}, y_{fc}, x_{com}})\{\mathbf{P, NR, H}\})$

Table 14: Agents mapping to Chor-calculus

Name	Chor-calculus
porqrs	$purchaseOrder^{rqrs}$
scpr	$shipmentChoreography^{pr},\ shipmentChoreography^{c}$
sorq	$shipOrder^{rq}$
scrs	$shipmentConfirmation^{rs}$
iorq	$invalidOrder^{rq}$
iorh	$invalidOrder^{handle}$

Table 15: Names mapping to Chor-calculus

<po.pi>

```
# Purchase Order Choreography

define PC(porqrs,scpr,sorq,scrs,x,y,iorh,iorq,io) =
(z)(ineb)(uneb)(infb)(unfb)(c)(yfc)(yeb)(yfb)(o)(q)(xcom)(t)(
P(porqrs,scpr,sorq,scrs,x,y,t,iorh,iorq,io) |
H(iorh,iorq,io,yeb,ineb,uneb,infb,unfb) |
NR(scpr,sorq,scrs,x,y) |
ineb!z.nil |
(c?(z).uneb!z.infb!z.nil + q?(z).nil) |
yfc?(z).(uneb!z.o!z.q!z.nil | xcom!z.nil) |
(t?(z).c!z.yfb?(z).nil + yeb?(z).nil + xcom?(z).yfc!z.nil))

define P(porqrs,scpr,sorq,scrs,x,y,t,iorh,iorq,io) =
(z)(porqrs!x.scpr!y.t!z.PC(porqrs,scpr,sorq,scrs,x,y,iorh,iorq,io))

define WEB_PC(iorh,iorq,io,yeb,ineb,uneb) = ineb?(z).iorh?(z).iorq!io.yeb!z.nil +
uneb?(z).nil

define WFB_PC(infb,unfb) = infb?(z).unfb!z.nil

define H(iorh,iorq,io,yeb,ineb,uneb,infb,unfb) = WEB_PC(iorh,iorq,io,yeb,ineb,uneb) |
WFB_PC(infb,unfb)

# Shipment Choreography

define QC(scpr,sorq,scrs,x,y) =
(z)(ineb)(uneb)(infb)(unfb)(c)(yfc)(yeb)(yfb)(o)(q)(xcom)(t)(
Q(scpr,sorq,scrs,x,y,t) |
H1(ineb,uneb,infb,unfb,yeb) |
ineb!z.nil |
(c?(z).uneb!z.infb!z.nil + q?(z).nil) |
yfc?(z).(uneb!z.o!z.q!z.nil | xcom!z.nil) |
(t?(z).c!z.yfb?(z).nil + yeb?(z).nil + xcom?(z).yfc!z.nil))
```

define **Q**(scpr,sorq,scrs,x,y,t) = (z)(sorq!x.scrs!y.t!z.**NR**(scpr,sorq,scrs,x,y))

define **WEB_QC**(ineb,uneb,yeb) = ineb?(z).yeb!z.nil + uneb?(z).nil

define **WFB_QC**(infb,unfb) = infb?(z).unfb!z.nil

define **H1**(ineb,uneb,infb,unfb,yeb) = **WEB_QC**(ineb,uneb,yeb) | **WFB_QC**(infb,unfb)

define **NR**(scpr,sorq,scrs,x,y) = scpr?(z).**QC**(scpr,sorq,scrs,x,y)

build **PC**

<formulas.pl>

define **P1** = *AG*(*EF*<scpr?z>true)

define **P2** = *AG*([porqrs!x]*EF*<scrs!y>true)

define **P3** = *EF*<scpr!y>true

define **P4** = *EF*<scrs!y>true

<po-m1.pi>

define **PC**(porqrs,scpr,sorq,scrs,x,y,iorh,iorq,io) =
(z)(ineb)(uneb)(infb)(unfb)(c)(yfc)(yeb)(yfb)(o)(q)(xcom)(t)(
P(porqrs,scpr,sorq,scrs,x,y,t,iorh,iorq,io) |
H(iorh,iorq,io,yeb,ineb,uneb,infb,unfb) |
NR(scpr,sorq,scrs,x,y) |
ineb!z.nil |
(c?(z).uneb!z.infb!z.nil + q?(z).nil) |
yfc?(z).(uneb!z.o!z.q!z.nil | xcom!z.nil) |
(t?(z).c!z.yfb?(z).nil + yeb?(z).nil + xcom?(z).yfc!z.nil))

define **P**(porqrs,scpr,sorq,scrs,x,y,t,iorh,iorq,io) =
(z)(porqrs!x.scpr!y.t!z.**PC**(porqrs,scpr,sorq,scrs,x,y,iorh,iorq,io))

define **WEB_PC**(iorh,iorq,io,yeb,ineb,uneb) = ineb?(z).iorh?(z).iorq!io.yeb!z.nil + uneb?(z).nil

define **WFB_PC**(infb,unfb) = infb?(z).unfb!z.nil

define **H**(iorh,iorq,io,yeb,ineb,uneb,infb,unfb) = **WEB_PC**(iorh,iorq,io,yeb,ineb,uneb) | **WFB_PC**(infb,unfb)

Shipment Choreography

define **QC**(scpr,sorq,scrs,x,y) =
(z)(ineb)(uneb)(infb)(unfb)(c)(yfc)(yeb)(yfb)(o)(q)(xcom)(t)(
Q(scpr,sorq,scrs,x,y,t) |
H1(ineb,uneb,infb,unfb,yeb) |
ineb!z.nil |
(c?(z).uneb!z.infb!z.nil + q?(z).nil) |
yfc?(z).(uneb!z.o!z.q!z.nil | xcom!z.nil) |
(t?(z).c!z.yfb?(z).nil + yeb?(z).nil + xcom?(z).yfc!z.nil))

define **Q**(scpr,sorq,scrs,x,y,t) = (z)(sorq!x.t!z.**NR**(scpr,sorq,scrs,x,y))

define **WEB_QC**(ineb,uneb,yeb) = ineb?(z).yeb!z.nil + uneb?(z).nil

define **WFB_QC**(infb,unfb) = infb?(z).unfb!z.nil

define **H1**(ineb,uneb,infb,unfb,yeb) = **WEB_QC**(ineb,uneb,yeb) | **WFB_QC**(infb,unfb)

define **NR**(scpr,sorq,scrs,x,y) = scpr?(z).**QC**(scpr,sorq,scrs,x,y)

build **PC**

<po-m2.pi>

Purchase Order Choreography

define **PC**(porqrs,scpr,sorq,scrs,x,y,iorh,iorq,io) =
(z)(ineb)(uneb)(infb)(unfb)(c)(yfc)(yeb)(yfb)(o)(q)(xcom)(t)(
P(porqrs,scpr,sorq,scrs,x,y,t,iorh,iorq,io) |

H(iorh,iorq,io,yeb,ineb,uneb,infb,unfb) |
NR(scpr,sorq,scrs,x,y) |
ineb!z.nil |
(c?(z).uneb!z.infb!z.nil + q?(z).nil) |
yfc?(z).(uneb!z.o!z.q!z.nil | xcom!z.nil) |
(t?(z).c!z.yfb?(z).nil + yeb?(z).nil + xcom?(z).yfc!z.nil))

define **P**(porqrs,scpr,sorq,scrs,x,y,t,iorh,iorq,io) =
(z)(porqrs!x.t!z.**PC**(porqrs,scpr,sorq,scrs,x,y,iorh,iorq,io))

define **WEB_PC**(iorh,iorq,io,yeb,ineb,uneb) = ineb?(z).iorh?(z).iorq!io.yeb!z.nil +
uneb?(z).nil

define **WFB_PC**(infb,unfb) = infb?(z).unfb!z.nil

define **H**(iorh,iorq,io,yeb,ineb,uneb,infb,unfb) = **WEB_PC**(iorh,iorq,io,yeb,ineb,uneb) |
WFB_PC(infb,unfb)

Shipment Choreography

define **QC**(scpr,sorq,scrs,x,y) =
(z)(ineb)(uneb)(infb)(unfb)(c)(yfc)(yeb)(yfb)(o)(q)(xcom)(t)(
Q(scpr,sorq,scrs,x,y,t) |
H1(ineb,uneb,infb,unfb,yeb) |
ineb!z.nil |
(c?(z).uneb!z.infb!z.nil + q?(z).nil) |
yfc?(z).(uneb!z.o!z.q!z.nil | xcom!z.nil) |
(t?(z).c!z.yfb?(z).nil + yeb?(z).nil + xcom?(z).yfc!z.nil))

define **Q**(scpr,sorq,scrs,x,y,t) = (z)(sorq!x.scrs!y.t!z.**NR**(scpr,sorq,scrs,x,y))

define **WEB_QC**(ineb,uneb,yeb) = ineb?(z).yeb!z.nil + uneb?(z).nil

define **WFB_QC**(infb,unfb) = infb?(z).unfb!z.nil

define **H1**(ineb,uneb,infb,unfb,yeb) = **WEB_QC**(ineb,uneb,yeb) | **WFB_QC**(infb,unfb)

define **NR**(scpr,sorq,scrs,x,y) = scpr?(z).**QC**(scpr,sorq,scrs,x,y)

build **PC**

<po-m3.pi>

Purchase Order Choreography

define **PC**(porqrs,scpr,sorq,scrs,x,y,iorh,iorq,io) =
(z)(ineb)(uneb)(infb)(unfb)(c)(yfc)(yeb)(yfb)(o)(q)(xcom)(t)(
P(porqrs,scpr,sorq,scrs,x,y,t,iorh,iorq,io) |
H(iorh,iorq,io,yeb,ineb,uneb,infb,unfb) |
NR(scpr,sorq,scrs,x,y) |
ineb!z.nil |
(c?(z).uneb!z.infb!z.nil + q?(z).nil) |
yfc?(z).(uneb!z.o!z.q!z.nil | xcom!z.nil) |
(t?(z).c!z.yfb?(z).nil + yeb?(z).nil + xcom?(z).yfc!z.nil))

define **P**(porqrs,scpr,sorq,scrs,x,y,t,iorh,iorq,io) = (z)(
scpr!y.porqrs!x.t!z.**PC**(porqrs,scpr,sorq,scrs,x,y,iorh,iorq,io))

define **WEB_PC**(iorh,iorq,io,yeb,ineb,uneb) = ineb?(z).iorh?(z).iorq!io.yeb!z.nil +
uneb?(z).nil

define **WFB_PC**(infb,unfb) = infb?(z).unfb!z.nil

define **H**(iorh,iorq,io,yeb,ineb,uneb,infb,unfb) = **WEB_PC**(iorh,iorq,io,yeb,ineb,uneb) |
WFB_PC(infb,unfb)

Shipment Choreography

define **QC**(scpr,sorq,scrs,x,y) =
(z)(ineb)(uneb)(infb)(unfb)(c)(yfc)(yeb)(yfb)(o)(q)(xcom)(t)(
Q(scpr,sorq,scrs,x,y,t) |
H1(ineb,uneb,infb,unfb,yeb) |

```
ineb!z.nil |
(c?(z).uneb!z.infb!z.nil + q?(z).nil) |
yfc?(z).(uneb!z.o!z.q!z.nil | xcom!z.nil) |
(t?(z).c!z.yfb?(z).nil + yeb?(z).nil + xcom?(z).yfc!z.nil))

define Q(scpr,sorq,scrs,x,y,t) = (z)(sorq!x.scrs!y.t!z.NR(scpr,sorq,scrs,x,y))

define WEB_QC(ineb,uneb,yeb) = ineb?(z).yeb!z.nil + uneb?(z).nil

define WFB_QC(infb,unfb) = infb?(z).unfb!z.nil

define H1(ineb,uneb,infb,unfb,yeb) = WEB_QC(ineb,uneb,yeb) | WFB_QC(infb,unfb)

define NR(scpr,sorq,scrs,x,y) = scpr?(z).QC(scpr,sorq,scrs,x,y)

build PC
```

Appendix B – BNF of CSP_{BPEL}

```
<DEFAULT>
TOKEN:
{
     < TRACE_PARALLEL_OPERATOR: "||" >
   | < TRACE_CHOICE_OPERATOR: "[]" >
   | < TRACE_DEADLOCK_OPERATOR: "STOP" >
   | < TRACE_PREFIX_OPERATOR: "->" >
   | < TRACE_TERMINATION_OPERATOR: "TERM" >
   | < TRACE_CALL_OPERATOR:"CALL" >
   | < TRACE_IF_OPERATOR: "IF">
   | < TRACE_ELSE_OPERATOR: "ELSE">
   | < TRACE_WHILE_OPERATOR: "WHILE">
   | < BPELTRACE: "trace" >
}

/* SEPARATORS */
<DEFAULT>
TOKEN:
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < LBRACE: "{" >
| < RBRACE: "}" >
| < LBRACKET: "[" >
```

```
| < RBRACKET: "]" >
| < COMMA: "," >
| < DOT: "." >
}

/* OPERATORS */
<DEFAULT>
TOKEN:
{
  < ASSIGN: "=" >
| < GT: ">" >
| < LT: "<" >
| < BANG: "!" >
| < HOOK: "?" >
| < COLON: ":" >
| < PLUS: "+" >
| < TILDE: "~" >
| < EQ: "==" >
| < LE: "<=" >
| < GE: ">=" >
| < NE: "!=" >
| < MINUS: "-" >
| < STAR: "*" >
| < SLASH: "/" >
| < SC_OR: "|" >
| < SC_AND: "&" >
| < XOR: "^" >
| < REM: "%" >
| < AT: "@">
}

TOKEN:
{
  <INT:"int">
| <CHAR: "char">
| <DOUBLE:"double">
| <BOOLEAN: "boolean">
| <NULL: "null">
}

/* LITERALS */
TOKEN:
{
  < INTEGER_LITERAL:
        ["1"-"9"] (["0"-"9"])* (["l","L"])? // DECIMAL LITERAL
      | "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+ (["l","L"])? //HEX LITERAL  >
```

```
|  < STRING_LITERAL:
     "\""
     (   (~["\"","\\","\n","\r"])
       | ("\\" ["n","t","b","r","f","\\","'","\""] )
     )*
     "\""  >
|<DURATION: "DU_”<STRING_LITERAL> >
|<DEADLINE:  “DE_”<STRING_LITERAL> >
}

TOKEN:
{
  < IDENTIFIER: (["a"-"z"] | ["A"-"Z"]) ( ["a"-"z"] | ["A"-"Z"] | ["0"-"9"] )* >
| <EVENT_IDENT: <IDENTIFIER>"."<IDENTIFIER>>
| <TRACE_DEADLOACK_OPERATOR:   "BPEL_Dead">
| <BPEL_WAIT_NAME>     :   “BPEL_Wait_Name”>
| <BPEL_WAITING_TIME> :   <DURATION>| <DEADLINE>
| <BPEL_MESSAGE_NAME:    “BPEL_Message_Name”>
| <BPEL_FAULT_NAME    :    “BPEL_Fault_Name”>
| <BPEL_VAR                  :     <IDENTIFIER> >
| <BPEL_FAULT_VAR>     :    “BPEL_Falut_Var”>
}

TraceAssertion          ::=      ( AssertionLabel )? TraceAssertionDeclaration
AssertionLabel          ::=      "[" <IDENTIFIER> "]"
TraceAssertionDeclaration    ::=      <BPELTRACE> <LPAREN> ( TraceConstant )*
( ProcessDeclaration )* <RPAREN>
TraceConstant           ::=      FieldDeclaration
FieldDeclaration        ::=      (Type) VariableDeclarator ( "," VariableDeclarator )* ";"
Type    ::=      ( PrimitiveType ) ( "[" "]" )*
PrimitiveType           ::=      <INT>| <CHAR> | <BOOLEAN>  | <DOUBLE>
VariableDeclarator      ::=      VariableDeclaratorId ( "=" VariableInitializer )?
VariableDeclaratorId    ::=     <IDENTIFIER> ( "[" "]" )*
VariableInitializer     ::=      ( ArrayInitializer | Expression )
ArrayInitializer        ::=     "{" ( VariableInitializer ( "," VariableInitializer )* )? ( "," )?
"}"
Expression      ::=      ConditionalExpression ( AssignmentOperator Expression )?
AssignmentOperator   ::=      ( "=" | "+=" | "-=" )
ConditionalExpression   ::=    ConditionalAndExpression ( "||"
ConditionalAndExpression )*
ConditionalAndExpression    ::=      EqualityExpression ( "&&" EqualityExpression )*
EqualityExpression    ::=      InstanceOfExpression ( ( "= =" | "!=" )
InstanceOfExpression )*
InstanceOfExpression          ::=      AdditiveExpression ( ( "<" | ">" | "<=" | ">=" )
AdditiveExpression )*
```

AdditiveExpression ::= MultiplicativeExpression (("+" | "-") MultiplicativeExpression)*
MultiplicativeExpression ::= UnaryExpression (("*" | "/" | "%") UnaryExpression)*
UnaryExpression ::= (PreIncrementExpression | PreDecrementExpression | PrimaryExpression)
PreIncrementExpression ::= "++" PrimaryExpression
PreDecrementExpression ::= "--" PrimaryExpression
PrimaryExpression :: = (Literal | <IDENTIFIER> | "(" Expression ")")
Literal ::= <INTEGER_LITERAL> | <STRING_LITERAL> | <BOOLEAN> | <NULL>
ProcessDeclaration ::= ((ProcessDeclarator <LBRACE> (FieldDeclaration)* ProcessExpression <RBRACE>) | (FieldDeclaration)* ProcessExpression)
ProcessDeclarator ::= <IDENTIFIER> FormalParameters
FormalParameters ::= "(" (FormalParameter("," FormalParameter)*)? ")"
FormalParameter ::= Type(VariableDeclaratorId)
ProcessExpression ::= ProcessParallelExpression
ProcessParallelExpression ::= ProcessPrefixExpression (<TRACE_PARALLEL_OPERATOR> ProcessPrefixExpression)*
ProcessPrefixExpression ::= (ProcessPrimaryExpression) (<TRACE_PREFIX_OPERATOR> (ProcessPrimaryExpression))*
ProcessPrimaryExpression ::= (<EVENT_IDENT> |BasicProcess | ProcessIfElseExpression | ProcessWhileExpression | (<LPAREN> ProcessExpression <RPAREN>))
BasicProcess ::= (<TRACE_DEADLOCK_OPERATOR> | <TRACE_TERMINATION_OPERATOR>)
ProcessIfElseExpression ::= <TRACE_IF_OPERATOR> <LPAREN> Expression <RPAREN> <LBRACE> ProcessExpression <RBRACE> <TRACE_ELSE_OPERATOR> <LBRACE> ProcessExpression <RBRACE>
ProcessWhileExpression ::= <TRACE_WHILE_OPERATOR> <LPAREN> Expression <RPAREN> <LBRACE> ProcessExpression <RBRACE>
Arguments ::= "(" (ArgumentList)? ")"
ArgumentList ::= Expression ("," Expression)

Appendix C – BPEL Mutants

The lines of code preceded with "//" represents the original code fragments that is replaced with new code to generate one or more mutants. The new code is the one missing the "//".

No.	Mutants Description	Fault Model
1	//<sequence> <flow>	**Sequential to Parallel (FM1)**
	//</sequence> </flow>	

2	//<invoke partnerLink="scheduling" name ="InitiateProductionScheduling"
	// portType="sch:scheduling" operation="requestProductionScheduling"
	// inputVariable="PO" outputVariable="productionSchedule"/>
	<invoke partnerLink="scheduling" name ="CompleteProductionScheduling"
	portType="sch:scheduling" operation="sendShippingSchedule"
	inputVariable="shippingSchedule" outputVariable ="finalSchedule"> <target linkName="ship-to-scheduling"/>
		Switch Activity (FM3)

	//<invoke partnerLink="scheduling" name ="CompleteProductionScheduling"
	// portType="sch:scheduling" operation="sendShippingSchedule"
	// inputVariable="shippingSchedule" outputVariable ="finalSchedule">
	// <target linkName="ship-to-scheduling"/>
	
	<invoke partnerLink="scheduling" name	

	="InitiateProductionScheduling" portType="sch:scheduling" operation="requestProductionScheduling" inputVariable="PO" outputVariable="productionSchedule"/>	
3	//</links> <link name="shipper1First"/></links>	**Parallel to Sequential (FM5)**
	//<source linkName="quoteShipper1"/> <source linkName="quoteShipper1"/><source linkName="shipper1First"/>	
	//<source linkName="quoteShipper2"/> <source linkName="quoteShipper2"/><target linkName="shipper1First"/>	
4	//</invoke> </invoke><terminate/>	**Unexpected Terminate (FM2)**
5	//</links> <link name="start-scheduling"/></links>	**Deadlock (FM4)**
	//</invoke> <target linkName=”start-scheduling”/></invoke>	
	//</invoke> <source linkName=”start-scheduling”/></invoke>	
6	//<source linkName="quoteShipper2"/> <target linkName=”quoteShipper1”/> <source linkName=”quoteShipper2”/>	**Synchronization to Sequence (FM6)**
	//<target linkName="quoteShipper1"/>	
	//joinCondition="quoteShipper1 AND quoteShipper2"	

7	//<source linkName="quoteShipper1"/> <source linkName="quoteShipper1" transitionCondition= "bpws:getVariableData(‘shippingInfo1’, 'price')<200"/>	**Extra Single Condition (FM8)**
8	//<source linkName="quoteShipper1"/> <source linkName="quoteShipper1" transitionCondition= "bpws:getVariableData(‘shippingInfo1’, 'price')<200"/>	**Extra And Conditions (FM9)**
	//<source linkName="quoteShipper2"/> <source linkName="quoteShipper2" transitionCondition= "bpws:getVariableData(‘shippingInfo2’, 'price')<200"/>	
9	//<source linkName="quoteShipper1"/> <source linkName="quoteShipper1" transitionCondition= "bpws:getVariableData(‘shippingInfo1’, 'price')<200"/>	**Extra OR Conditions (FM10)**
	//<source linkName="quoteShipper2"/> <source linkName="quoteShipper2" transitionCondition= "bpws:getVariableData(‘shippingInfo2’, 'price')<200"/>	

	// joinCondition="quoteShipper1 AND quoteShipper2"
	joinCondition="quoteShipper1 OR quoteShipper2"	

10	// <case condition="bpws:getVariableData('shippingInfo1','price') <= // bpws:getVariableData('shippingInfo2','price')> <case condition="bpws:getVariableData('shippingInfo1','price') > bpws:getVariableData('shippingInfo2','price')>	**Switch Condition (FM11)**
11	//transitionCondition="bpws:getVariableData('PO',"needElectronicInvoice") = 'yes'" transitionCondition=" bpws:getVariableData('PO',’needElectronicInvoice’) = 'yes'” AND bpws:getVariableData('PO',’needPaperInvoice’) = 'no'"	**Multi-Choice to Simple Merge (FM14)**

12	//transitionCondition="bpws:getVariableData('PO',"needPaperInvoice")= 'Yes'"
	
	//transitionCondition="bpws:getVariableData('PO',"needElectronicInvoice") = 'Yes'"
		Multi-Choice to Parallel (FM15)

13	// joinCondition="sendPaper-invoice OR sendEmail-invoice"
	joinCondition="sendPaper-invoice AND sendEmail-invoice"
		Synchronizing Merge to Synchronization with AND conditions (FM16)

14	// <from expression="bpws:getVariableData('po','price') + // bpws:getVariableData('shippingInfo','price')"/> <from expression="bpws:getVariableData('po','price') - bpws:getVariableData('shippingInfo','price')"/>	**Mathematics operator exchanged**

15	// outputVariable="shippingInfo2"
	outputVariable="shippingInfo1"
		Variable by Variable replacement

16	// <from expression="bpws:getVariableData('po','price') + // bpws:getVariableData('shippingInfo','price')"/> <from expression="bpws:getVariableData('po','price') + bpws:getVariableData('shippingInfo','price')+ 100"/>	**Increment Variables**
17	// <from expression="bpws:getVariableData('po','price') + // bpws:getVariableData('shippingInfo','price')"/> <from expression="bpws:getVariableData('po','price') + bpws:getVariableData('shippingInfo','price')- 100"/>	**Decrement Variables**
18	//<case condition="bpws:getVariableData('totalCharge','number') <= //bpws:getVariableData('credit','number')> <case condition="bpws:getVariableData('totalCharge','number') <= 100	**Variable by constant replacement**

Table 16: Mutants generated in Illustration 1

No.	Mutants Description	Fault Model
19	//<source linkName="policeReport"/>	**Sequential to Parallel(FM1)**
	//<target linkName="policeReport"/>	
20	//</invoke> </invoke><terminate/>	**Unexpected Terminate (FM2)**
21	//</links> <link name="policeReport"/> </links>	**Deadlock (FM4)**
	//</invoke> <target linkName=”extra-link”/></invoke>	
	//</assign> <source linkName=”extra-link”/></assign>	
22	//</flow>	**Synchronization to Parallel (FM7)**
	//</assign> </assign></flow>	
23	//<flow> <sequence>	**Synchronization to Sequence (FM6)**
	//</flow> </sequence>	

24	//transitionCondition="bpws:getVariableData('claim','directDeposit")= 'yes'
	transitionCondition="bpws:getVariableData('claim','directDeposit")!= 'yes'
		Switch Condition (FM11)

	//transitionCondition="bpws:getVariableData('claim','directDeposit")=! 'yes'
	transitionCondition="bpws:getVariableData('claim','di	

	rectDeposit")= 'yes'	
25	//transitionCondition="bpws:getVariableData('claim','directDeposit")= 'yes'	**Exclusive Choice to Parallel (FM12)**
	//transitionCondition="bpws:getVariableData('claim','directDeposit")!= 'yes'	
26	// trnaiditionCondition="bpws:getVariableData('claim','totalExpense')<1000 AND // bpws:getVariableData('claim','totalExpense')>200"/> bpws:getVariableData('policeReport','liability')!='Full'	**Simple Merge to Multi-Choice (FM13)**
27	// <onAlarm for="'PT02M'"> // <throw faultName="lns:Timeout" faultVariable="Fault" /> // </onAlarm>	**Missing Alarm(FM17)**
28	//<catch faultName="lns:InfoNotAvaliable" faultVariable="Fault"> <catch faultName="lns:Timeout" faultVariable="Fault">	**Incorrect Fault Matching(FM18)**
29	//<from expression="bpws:getVariableData('medExpense','count')+ //bpws:getVariableData('repairExpense','count')"/> <from expression="bpws:getVariableData('medExpense','count') - bpws:getVariableData('repairExpense','count')"/>	**Mathematics operator exchanged**
30	//<from expression="bpws:getVariableData('medExpense','count')+	**Variable by Variable replacement**

	//bpws:getVariableData('repairExpense','count')"/> <from expression="bpws:getVariableData('repairExpense','count') + bpws:getVariableData('repairExpense','count')"/>	
31	//<source linkName="majorAccident" //transitionCondition="bpws:getVariableData('claim','totalExpense')>1000"/> <source linkName="majorAccident" transitionCondition="bpws:getVariableData('claim','totalExpense')>10000"/>	**Increment constants**
32	//<source linkName="majorAccident" //transitionCondition="bpws:getVariableData('claim','totalExpense')>1000"/> <source linkName="majorAccident" transitionCondition="bpws:getVariableData('claim','totalExpense')>100"/>	**Decrement constants**
33	//<from expression="bpws:getVariableData('medExpense','count')+ //bpws:getVariableData('repairExpense','count')"/> <from expression="bpws:getVariableData('medExpense','count') +500”/>	**Variable by constant replacement**

Table 17: Mutants generated in Illustration 2