# Automatic Detection of Business Process Interference

N.R.T.P. van Beest[1], E. Kaldeli[2], P. Bulanov[2], J.C. Wortmann[1], and A. Lazovik[2]

[1] Department of Business & ICT, Faculty of Economics and Business, University of Groningen
Nettelbosje 2, 9747 AE Groningen, The Netherlands
[2] Distributed Systems Group, Johann Bernoulli Institute, University of Groningen, Nijenborgh 9, 9747 AG, The Netherlands

**Abstract.** Today's organizations are characterized by long-running distributed business processes, which involve different stakeholders and share common resources. One of the main challenges posed in such a highly distributed setting comes from the interference between different processes that are running in parallel. During execution of a business process, a data modification caused by some external process may lead to erroneous and undesirable business outcomes. In order to address this problem, we propose to annotate business processes with dependency scopes, which cover critical sections of the process. Erroneous execution can be prevented by executing intervention processes, which are triggered at runtime. However, for complex processes with a large number of activities and many interactions with the environment, the manual specification of the appropriate critical sections can be particularly time-consuming and error-prone. To overcome this limitation, we present an algorithm for automating the discovery of critical sections. The proposed approach is applied on a real case-study of a BP from the Dutch e-Government.

## 1 Introduction

Modern private and public organizations are moving from traditional, proprietary and locally managed Business Process Management Systems (BPMS) to BPMS where more and more tasks are outsourced to third party providers and resources are shared among different stakeholders. Often, this is realized by the emergent paradigms such as Service Oriented Computing (SOC) and cloud computing. As a result, business processes (BPs) can no longer be considered in isolation, since data can be simultaneously accessed and modified by different external processes. Disregarding the interdependencies with external actors and other processes may lead to inconsistent situations, potentially resulting in undesirable business outcomes. The situation where undesirable business outcomes are caused by data modifications of some other concurrently executing process is known as *process interference* [1, 2]. The problem of process interference is particularly relevant for knowledge-intensive BPs, where shared data are accessed and modified by many processes, involving a large number of stakeholders.

E-Government is a typical area characterized by multiple concurrently executing knowledge-intensive processes. These processes access and modify commonly shared resources such as citizen data, information reported by external contracted parties, etc. In such a context, a "think globally, act locally" approach has to be adopted: each BP instance has to take its own action, independently of other processes, based on how its knowledge about the world evolves during runtime, and how this knowledge affects the next tasks in its workflow. For example, important data used by subsequent tasks may become obsolete, and conditions on which the process relies may not hold anymore. Therefore, a BP has to be continuously informed about changes concerning that data, reason about them, and react accordingly in order to be able to ensure its consistency with the new state of the world.

In the Netherlands, a first attempt has been made to provide a Software as a Service (SaaS) solution for the local e-Government (`www.govunited.nl`). One of the processes that is proposed as a candidate for this initiative concerns the process of the Dutch Law for Societal Support, known as the WMO law. This law is intended to offer support for people with a chronic disease or a disability, by providing facilities (usually by external parties) such as domestic care, transportation, a wheelchair or a home modification. Naturally, several different instances of the WMO process can be executed concurrently, together with other governmental processes, which may access and modify the same data. For example, during the execution of the WMO process, the citizen may move to a different address, the medical status of the citizen may alter, the eligibility criteria may change because of some new directive etc. These changes may pass unnoticed by BPs which rely upon them, and consequently result in unexpected behavior and undesirable business outcomes. The consequences are often noticed only by end customers [3], by erroneous orders or invoices, customer requests that are never handled, etc.

Traditional verification techniques for workflow and data-flow (e.g. [4]) are not sufficient for ensuring the correctness of such BPs, as they assume a closed environment where no other process can use a service that affects the data used by that organization. In addition, most work about resolving process interference refers to failing processes or concerns design-time solutions [5, 6]. Consequently, neither of these solutions is suitable for a highly dynamic SaaS environment. In [2], a run-time mechanism is proposed, where vulnerable parts of the process are monitored in order to manage interferences by employing intervention processes. *Dependency scopes (DS)* are used to specify a *critical section* of the BP, whose correct execution relies on the accuracy of a *volatile* process variable, i.e. a variable that can be changed externally during the execution of the process. If a volatile variable is modified by some exogenous factor during execution of the activities in the respective DS, an *intervention process (IP)* is triggered, with the purpose of resolving the potential execution problems stemming from this change event. However, for complex processes with a large number of activities and many interactions with the environment, the task of manually annotating a BP with DSs becomes difficult, time-consuming, and prone to errors. Thus,

critical parts of the BP whose correct execution is dependent on the validity of some volatile variable may be neglected.

In this paper, we extend the initial idea presented in [2], by systematizing the main methodology, and providing an algorithm which automates the task of identifying the critical parts of a BP. To this end, we concretize the proposed approach by describing the semantic extensions to the BP modelling that allow the specification of DSs for resolving runtime process errors. Given a block-style BP specification and some basic information about the services it uses (i.e. the input-output parameters and internal state variables), we show how the parts of the process that are covered by DSs can be automatically inferred. This way, the task of the BP designer can be highly facilitated.

The remainder of this paper is organized as follows. Section 2 describes a possible interference scenario on a real case-study taken from Dutch e-Government, which plays the role of our running example. In Section 3 the basic definitions required for the proposed approach are presented. The algorithm for the automatic identification of critical sections is described in Section 4. Section 5 provides an overview of related work, and the overall conclusions are drawn in Section 6.

## 2 A Process Interference Case-study

In order to illustrate the effects of process interference and the potential ways to overcome them, let us consider a real case-study from the Dutch e-Government regarding the WMO law, as described in [2]. The BP under investigation (referred to as WMO process) concerns the handling of the requests from citizens at one of the 430 municipalities in the Netherlands. In this section, the WMO process is described as used by one of the municipalities. Furthermore, an example is provided, showing the required DSs along with the required IPs.

### 2.1 WMO Process Description

The WMO process (shown in Figure 1) starts with the submission of an application for a provision by a citizen. After receiving the application at the municipality office, a home visit is executed by an officer, in order to gather a detailed understanding of the situation. After the home visit, additional information on the citizen's health may still be required, which can be obtained via a medical advice provided by e.g. a general practitioner. Based on this information, a decision is made by the municipality to determine whether the citizen is eligible to receive the requested provision or not. In case of a negative decision, the citizen has the possibility for appeal. In case of a positive decision, the process continues and the requested provision will be provided. For domestic help, the citizen has the choice between "Personal Budget" and "Care in Kind". In case of a "Personal Budget", the citizen periodically receives a certain amount of money for the granted provision, and in case of "Care In Kind" suppliers who can take care of the provision are contacted. For obtaining a wheelchair, first the detailed requirements are acquired before sending the order to the supplier. The home

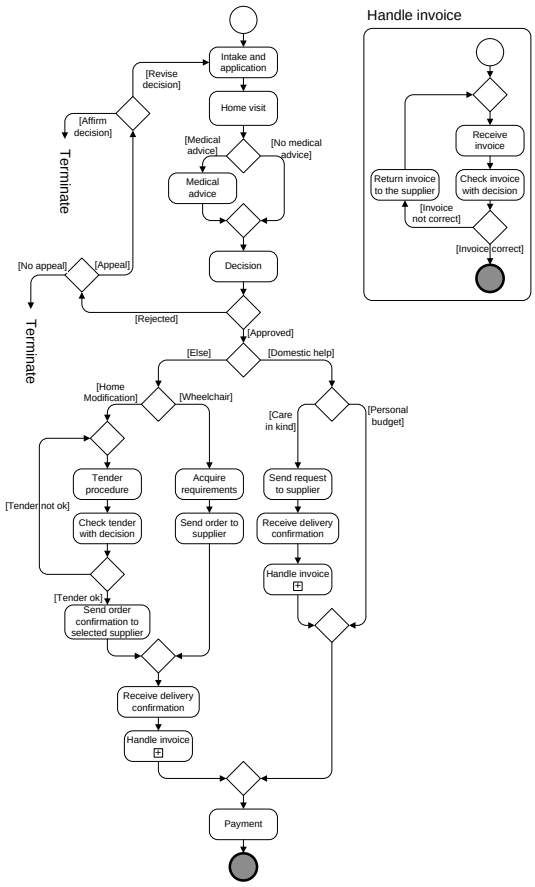


Fig. 1: The WMO process

modification involves a tender procedure to select a supplier that provides the best offer. If the selected tender is approved by the municipality, the order is sent to the selected supplier. After delivery of the provision, an invoice is sent by the supplier to the municipality. Finally, the invoice is checked and paid.

## 2.2 Interference Examples

The request for a wheelchair or a home modification may take up to 6 weeks until the delivery of the provision. These processes depend on the correctness of a number of process variables, like the address of the citizen and the content of the decision. However, these process variables may be changed by another process running in parallel, independently from the WMO process, and are, therefore, volatile. A change in either of these process variables (e.g. address) may have potentially negative consequences for the WMO process, due to its dependencies

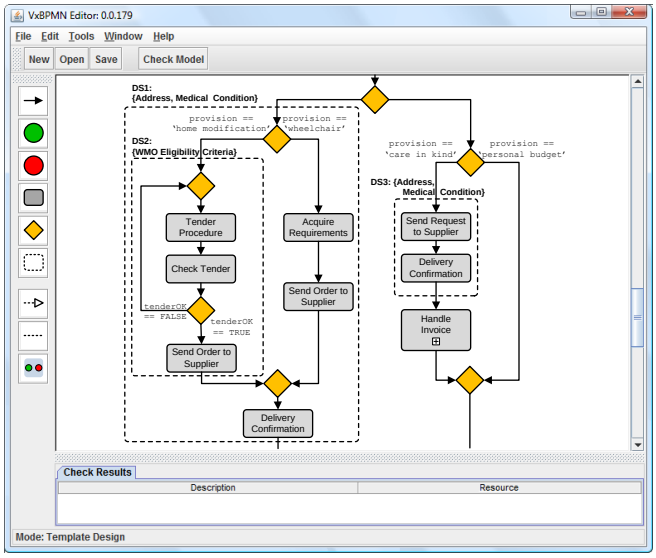


Fig. 2: WMO dependency scopes

on those variables, and lead to erronous outcomes. Such situations are typical examples of process interference.

For example, the requirements of a wheelchair may depend on certain characteristics of the citizen's home. Consequently, an address change after "Acquire requirements" might result in a wheelchair that does not fit the actual requirements. Similarly, if the citizen moves to a nursing home after "Check tender with decision", the home modification is not necessary anymore. However, the supplier is not notified of this address change and the municipality is notified through a different process, which is external to the WMO process. As a result, unless some action is taken to cancel or update the order, the WMO process will proceed with the home modification. In order to guard for changes to the volatile process variables, DSs can be defined, covering those activities for which such a change poses a potential risk of interference. In Figure 2, a part of the process is annotated with DSs using a Process Modeller tool developed for the graphical modeling of BPs. The tool provides a selection of standard control blocks like flow, switch etc., with the extra support of design tools for modeling DSs. For the implementation details see [7].

The activities in DS1 rely on the accuracy of the address. If the address changes, the DS should be triggered, and potentially some recovery activities need to be executed, depending on the state of the BP at that point. For example, if the address change is detected before the order for a wheelchair is sent to the supplier, it is sufficient to execute the IP as shown in Figure 3a. However, if the order is already sent to the supplier, some additional activities are required (Figure 3b). First of all, the current order should be put on hold. After acquiring

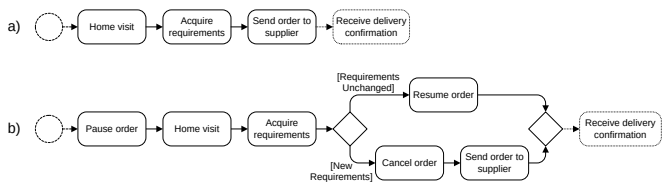


Fig. 3: WMO intervention examples

the requirements again, it is evaluated whether there is a change. If not, the order can be resumed, otherwise the old order should be cancelled and a new order should be sent. The specification of IPs is outside the scope of this paper (for a detailed discussion about the specification of IPs see [2] and [7]).

## 3 Basic Definitions

In this section, we provide the basic definitions regarding the BP representation extended with the support of DSs. First, we define the *Service Repository (SR)*, which is a registry that keeps semantic information about a set of services that are accessible to the client who is executing a specific BP. The SR plays the role of a pool of service descriptions and instances, which are used as the building elements of different process specifications. Service descriptions specify the basic functionalities provided by a service. Service instances refer to specific providers, which offer a service whose functionality conforms to some service description.

The service descriptions specify the operations offered by the respective service type and are represented in terms of simple semantics. Service instances refer to specific providers of a certain service description. The service descriptions can be extracted from standard semantic languages for representing Web Services, such as WSDL-S (`www.w3.org/Submission/WSDL-S`) and OWL-S (`www.w3.org/Submission/OWL-S`). The service descriptions capture the Input-Output behavior of the operations, i.e. the type of the input parameters inputs and of the expected outputs, as well as some information about its internal variables (similar to Locals in OWL-S). No extra semantic information is required to automatically identify the critical sections of a BP.

**Definition 1 (Service Repository (SR)).** *A Service Repository* $SR=(SD, SI)$ *is a registry, which keeps a set of* Service Descriptions $SD$*, and a set of* Service Instances $SI$*. A Service Description* $sd \in SD$ *is a tuple* $sd = (sdid, O, SV)$*, where sdid is a unique identifier,* $O$ *is a set of service operations, and* $SV$ *is a list of variables, each ranging over a finite domain. These variables correspond to state variables internal to the service, whose value can be changed by the service operations. Each service operation* $o \in O$ *is a tuple* $o = (id(o), in(o), out(o))$ *where:*

- $id(o)$ *is the identifier of the operation*
- $in(o)$ *is a list of variables that play the role of input parameters to o, ranging over finite domains*

– *$out(o)$ is a list of variables that play the role of output parameters to $o$, ranging over finite domains*

*A Service Instance $si \in SI$ is a tuple $si = (iid(si), st(si))$:*
– *$st(si)$ is the unique identifier (service type) of the service description $sd \in SD$ this instance complies with*
– *$iid(si)$ is an instance identifier. For each pair of service instances $si_1, si_2 \in SI$ that have the same service type $st(si_1) = st(si_2)$, $iid(si_1) \neq iid(si_2)$.*

The set of state variables involved in the SR may be used by different running process instances, and their value may be changed by any process that has access to the respective setting service operation.

In the followings, the working definition of a Business Process (BP) is provided. Although the WMO process (Figure 1) is represented in BPMN-notation for readability, the core BP representation used in this paper is block-structured [8], and uses the basic BPEL constructs of BPEL, enriched with DSs. As such, the syntax of the BP is block-structured and unambiguously defined, so that the BP can be directly executed by an orchestrator [9], and automatically parsed to identify the parts of the BP that should be covered by a DS. The representation is ultimately a tree structure where a block can have other blocks as children, and for each block its parent can be obtained. All activities included in the BP are references to service instances that exist in the Service Repository.

**Definition 2 (Business Process (BP)).** *Given a Service Repository $SR = (SD, SI)$, a Business Process is a tuple $BP = (PV, E)$, with $E$ being a process element $E = (ACT \mid SEQUENCE \mid FLOW \mid SWITCH \mid REPEAT \mid WHILE \mid DS)$, where:*

– *$PV = PV_i \cup PV_e$ is a set of variables ranging over finite domains.*
  - *$PV_i$ is a set of internal variables, which are declared at the BP level (BP-specific). A subset of $PV_i$ are passed as input parameters to the entire BP, in which case we write $BP(pv_1, \ldots, pv_n)$, where $pv_i \in PV_i$ and $pv_i$ can be initialized with specific values at execution time.*
  - *$PV_e$ is a set of external variables, which refer to state variables declared in the SR. An external variable $v \in PV_e$ is a reference $sdid.iid.vid$, where $sdid$ is the identifier of a service description $sd = (sdid, O, SV) \in SD$, $iid$ is the identifier of a service instance $si = (iid, sdid) \in SI$, and $vid$ is the identifier of some state variable $v \in SV$.*
– *$ACT$ is a process activity, which represents the invocation of a service operation. For instance, in BPEL it may correspond to an invoke, receive, reply, etc. Every $ACT$ refers to an operation that exists in $SI$. It is a tuple $act = (id(act), in(act), out(act))$, where $id(act)$ is a reference $sdid.iid.oid$, with $sdid$ being an identifier of a service description $sd = (sdid, O, SV) \in SD$, $iid$ the identifier of a service instance $si = (iid, sdid) \in SI$, and $oid$ is the identifier of some operation $o \in O$. The input and output parameters of $act$ refer to the inputs and outputs of the respective $oid$, i.e. $in(act) = in(oid)$ and $out(act) = out(oid)$. The input (output) parameters of all activities in the BP*

*form the sets IP (OP). Input variables can be assigned with constant values or other process variables:* $id(act)(ip_1 := v_1, \ldots, ip_n := v_n)$*, where* $ip_i \in in(act)$*,* $v_i \in (PV \cup OP)$*, or* $v_i$ *is a value compliant with* $ip_i$*'s domain. There are also two special types of activities: no-op, which represents an idle activity, and exit, whose execution causes the entire BP to halt.*

- *SEQUENCE refers to a totally ordered set of process elements, which are executed in sequence:* $SEQUENCE\{e_1 \ldots e_n\}$*, where* $e_i$ *is a process element.*
- *FLOW represents a set of process elements, which are executed in parallel:* $FLOW\{e_1 \ldots e_n\}$*, where* $e_i$ *is a process element.*
- *SWITCH is a set of tuples* $\{(c_1, e_1), \ldots, (c_n, e_n)\}$*, where* $e_i$ *is a process element and* $c_i$ *is a logical condition* $C ::= var \circ v$*, where* $var \in (PV \cup OP)$*,* $v$ *is some constant belonging to var's domain, and* $\circ$ *is a relational operator* ($\circ \in \{=, <, >, \neq, \leq, \geq\}$)*. All* $c_i$ *participating in a SWITCH refer to the same variable var and are mutually exclusive.*
- *REPEAT represents a loop structure, and is defined as a tuple* $(pe, c\{pe_i\})$*, where c is a logical condition as already defined, and* $pe, pe_i$ *are process elements. c is evaluated just after the end of pe, and if it holds then pe is repeated, after the execution of the optional* $pe_i$*.*
- *DS is a dependency scope as defined in Definition 3.*

### 3.1 Dependency scopes

The DS is based on a **guard-verify** structure to deal with modification events due to factors exogenous to the BP, e.g. due to some other process execution which affects some data on which the BP relies. The critical part of the BP is included in the *guard* block, while the *verify* block specifies the types of events that require intervention. The mechanism of event recording and handling are out of scope of this paper (for a system dealing with process-generated events see e.g. [10]). Whenever such an event occurs, the control flow is transferred to the *verify* block, and the respective goal is activated. Once the resulting IP finishes execution in the updated environment, the control flow of the BP continues from the point following the *guard-verify* structure, unless it is explicitly forced to terminate.

**Definition 3 (Dependency Scope (DS)).** *Given a* $SR = (SD, SI)$ *and a* $BP = (PV_i \cup PV_e, E)$*, a dependency scope is a tuple* $DS = \langle guard(VV)\{CS\}, verify(\{(c_i, IP_i \mid terminate(IP_i))\})\rangle$*, where:*

- $guard(VV)$ *indicates the set of volatile variables* $VV \subset PV_e$ *whose modification triggers the verification of the DS, and CS a process element of BP which is called the* Critical Section*. Whenever during the execution of CS a modification event regarding the value of a* $vv \in VV$ *is received, the verify part of the DS is triggered, and BP's execution is interrupted.*
- $verify(\{(c_i, IP_i)\})$ *comprises a set of tuples consisting of a logical condition* $c_i$ *and an intervention process* $IP_i$ *in compliance with Definition 2 to be pursued if* $c_i$ *holds. Providing a case condition is optional, with the default interpretation*

*being $c_i = TRUE$. $IP_i$ specifies a BP which ensures the satisfaction of the properties that reflect the state right after the final activity of CS. After the interruption of the BP, some $IP_i$ is executed, and then BP is resumed just after CS (and from any other parallel branches that were interrupted).*
- *terminate(IP) forces the rest of BP's execution to be aborted after completing IP's execution.*

Following Definition 3, the DS specification representing $DS_1$ of Figure 2 is as follows, where *IPa*, *IPb* and *IPc* refer to the respective intervention processes, which take care of repairing the erroneous execution in each of the cases.

```
<ds>
  <guard>
    <variables>
      <variable name="address" dataType="dt:address"/>
      <variable name="medCond" dataType="dt:medInfo"/>
    </variables>
    <criticalSection>
      <!-- Subprocess covered by DS1 as in Figure 2 -->
    </criticalSection>
  </guard>
  <verify>
    <case condition="address.county!='Groningen'">
      <terminate>
        <invoke name="IPa"/>
      </terminate>
    </case>
    <case condition="address.county='Groningen'&AND;medCond!='deceased'">
      <invoke name="IPb"/>
    </case>
    <case condition="medCond='deceased'">
      <terminate>
        <invoke name="IPc"/>
      </terminate>
    </case>
  </verify>
</ds>
```

According to $DS_1$, if a modification event regarding the address or the medical condition is received within the scope of the guarded subprocess, different IPs are executed, depending on the state of execution and the kind of modification that has occurred. For example, if the address change indicates that the citizen has moved to another municipality, then *IPa* includes canceling the order (either for a wheelchair or home modification) if one has already been issued, and sending a notification to the city hall. Similarly, IPb takes care of the situation where the customer has moved within the range of the municipality, and IPc in case his medical condition has changed to 'deceased'. In the following section we describe how the $guard(VV)\{CS\}$ part of a DS description can be derived automatically, by parsing the BP specification.

## 4 Automatic Identification of Critical Sections

The algorithm of automated generation of the parts of a BP covered by a DS is presented in Algorithm 1 below. The algorithm guarantees that the computed

CSs are elements of the BP in compliance with Definition 2. CSs cover all activities that are directly or indirectly dependent on the same set of volatile variables $VV$. That is, they either use a $vv \in VV$ as input or use the output of another activity, which is dependent on $vv$. These activities are referred to as *Dependent Activities* ($DA$). In order to ensure that important change events will not pass untreated, any part of the process in a potential execution path between two activities dependent on the same $VV$ should also be covered by the respective CS. This is necessary to take care of any modification of $vv$ that occurs during the execution of this intermediate part, since the modification may require the cancelation or repetition of some preceding part of the BP which relied on some $vv \in VV$ (e.g. performing a new visit to the new house if the address has changed), and which is used by a succeeding element (e.g. to calculate the characteristics of the requested wheelchair). However, branches in switch or flow constructs that are not on a potential path between two activities dependent on some $vv$, should not be unnecessarily included in the respective CS, in order to avoid unnecessary invocation of intervention processes.

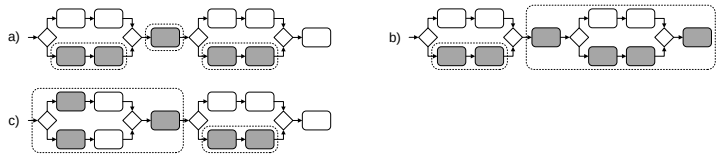


Fig. 4: CS creation examples

In Figure 4, some examples of CSs are provided to illustrate the properties described above. The shaded activities are dependent on $VV$ and should be covered by a CS. The CSs are indicated by a dashed line. In case (a), only the specific branches of the switch-constructs that comprise dependent activities are included in the CS. In situation (b), however, the second switch has to be covered entirely by a CS, because the last activity is dependent on $VV$ as well. Any modification event regarding a $vv \in VV$ that occurs during the upper branch (which is not dependent on $VV$) has still to be dealt with, since the last activity may use a a variable that is a result of some dependent activities before the switch, which produced this result based on the obsolete $vv$. In situation (c), both branches of the first switch contain activities that are not dependent on $VV$. However, as they both are on a path between activities that are dependent on $VV$, the entire switch is covered by a CS.

The main function of Algorithm 1 is *extractScopes*, which takes as an input a BP specification in accordance with Definition 2 and the list of volatile variables $VV$. *extractScopes* returns a list of tuples $\langle VV_i, CS_i \rangle$, which correspond to the *guard* parts of all DSs in the BP. Given a $BP = (PV_i \cup PV_e, E)$, $VV = PV_e$. That is, all state variables that are declared in the SR and used in the BP should be guarded, since their modification may be a source of erroneous results. The

BP is treated as a tree (represented in XML), where the root is the outermost element in the specification, and the leaves are the activities.

The outermost loop in the function *extractScopes* iterates over the list of volatile variables $VV$. For each $vv \in VV$, critical sections are extracted separately. Identical CSs for different variables are merged into a united CS at the end by *mergeScopes*. The first step (line 4) is to find all activities and switch–blocks that depend directly or indirectly on the volatile variable $vv$, by calling the function *getDependentElems*. First (line 18), all activities for which $vv$ is assigned to some of their input parameters directly or by transitivity are added to the dependent elements $DE$. Then (line 24), $DE$ is augmented by adding all switch–blocks whose condition is either on $vv$, or some variable produced by the already considered activities. All elements in $DE$ are arranged in a breadth-first order as they appear in the BP. The next step in *extractScopes* is to iterate through the list $DE$. In the inner loop, for each pair of elements $e_i, e_j$, it is checked whether their minimal common ancestor is of type sequence. If so, then the function *getTempCS* is called, which returns a set of elements that are candidates for being CSs with respect to the variable $vv$, and lie between $e_i$ and $e_j$. Then, $e_j$ can be removed from $DE$, since subsequent inspections on it are redundant, as the appropriate CSs covering it have already been computed.

Function *getTempCS*($e_i, e_j, BP$) first calls *getPathBtw* to compute the path between $e_i$ and $e_j$ (line 31), which comprises all elements that are part of the sequence between $e_i$ and $e_j$, including the special markers *StartBranchEl* and *EndBranchEl*. These markers indicate the start (splits) and end points (joins) of branching elements. Consequently, a *path* is a list with members of type *Item* (line 44), where an item is either a process element or a *BranchElMarker*. Markers are added in the path only if they concern joins (splits) for which the respective split (join) is not encountered during the traversal of the BP from $e_i$ to $e_j$. This way, the markers divide the path into the appropriate sequences of elements (lines 33 to 39), each of which is a candidate for being a CS.

Function *getPathBtw* uses the auxiliary function *nextItems* (not explained in the algorithm for space reasons), which returns a list consisting of the next element in the sequence path, and some possible *EndBranchEl*, if any are encountered before the next element is fetched. These are added to the path, and the process proceeds by fetching the next items (line 45), until the element in the sequence that contains $e_j$ is reached. In the latter case, *pathInElem* is called, which traverses the path within this last element until $e_j$ is reached. If the element containing $e_j$ is an activity or sequence, this activity ($e_j$) or the subsequence till $e_j$ (line 52) are returned respectively. If the element is a switch or flow, then a *StartBranchEl* marker is added in the list of results, and the branch containing $e_j$ is inspected. *pathInElem* is called recursively on this branch, and all items in the path leading to $e_j$ are collected in $path_j$. Consequently, the computation of the entire path is completed, and returned to *getTempCS*. The path is traversed (line 33), and divided into the appropriate CSs: *currCS* is constructed as a sequence of the elements in *path*, until a marker is met, at which point *currCS* is added to the list of candidate CSs.

**Algorithm 1** Automatic computation of the set of the pairs Guarded=$\{\langle VV_i, CS_i\rangle\}$, consisting of volatile variables and respective elements that constitute the Critical Sections

```
1: function extractScopes(BP, VV): List[(List[V], E)]
2:   for each vv ∈ VV do
3:     guardList = ∅
4:     DE = getDependentElems(vv, BP)
5:     for each e_i ∈ DE do
6:       tmpCS = ∅
7:       DE = DE.remove(e_i)
8:       for each e_j ∈ DE do
9:         if type(minCommonAncestor(e_i, e_j))=sequence then
10:          tmpCS = tmpCS ∪ getTempCS(e_i, e_j, BP)
11:          DE = DE.remove(e_j)
12:      for tmpCS_i ∈ tmpCS do
13:        guardList.add(⟨{vv}, tmpCS_i⟩)
14:  mergeScopes (guardList)

15: function getDependentElems(vv, BP): List[Element]
16:   varList = {vv}
17:   DE = ∅
18:   for each a_i ∈ BP.getActivities do
19:     for each ip_i := v ∈ a_i.parseInputAssignments do
20:       if v ∈ varList then
21:         for each op_i ∈ out(a_i) do
22:           varList.add(op_i)
23:         DE.add(a_i); break;
24:   for each SWITCH_i ∈ BP.getSWITCHelements do
25:     c_i = SWITCH_i.getFirstCondition
26:     if c_i.getLeftVariable ∈ varList then
27:       miDE.add(SWITCH_i);
28:   return DE

29: function getTempCS(e_i, e_j, BP): List[Elem]
30:   tmpCSList = ∅
31:   path = getPathBtw(e_i, e_j, BP)
32:   currCS = ∅
33:   for each item ∈ path do
34:     match type(item)
35:       case Element:
36:         currCS.attachInSeq(item)
37:       case BranchElMarker:
38:         tmpCSList.add(currCS)
39:         currCS = ∅
40:   return tmpCSList
```

```
41: function getPathBtw(e_i, e_j, BP): List[Item]
42:   currElem = e_i
43:   while ¬ currElem.contains(e_j) do
44:     path.append(currItems)
45:     currItems = nextItem(currElem, e_i, BP)
46:     currElem = currItems.getElement
47:     if currItems = ∅ then return ∅
48:   path.append(pathInElem(currElem, e_j, BP))
49:   return path

50: function pathInElem(el, endEl, BP): List[Item]
51:   match type(el)
52:     case activity:
53:       return {el}
54:     case sequence:
55:       return el.subsequenceTill(endEl)
56:     case SWITCH ∨ flow:
57:       path_j = {StartBrEl}
58:       branch_j = el.getBranchWith(endEl)
59:       return path_j.append(pathInElem(branch_j, endEl, BP)
60:   return ∅
```

Once the list of temporary CSs *tmpCS* regarding a volatile variable *vv* is computed as described above, *extractScopes* proceeds with constructing the respective *guardList* consisting of tuples $\langle\{vv\}, tmpCS_i\rangle$ (line 12). After repeating the process described above for each $vv \in VV$, *mergeScopes* is called, in order to clean up the candidate CSs. The following steps are performed in that order:

- If there are two tuples $\langle\{v_1\}, CS_1\rangle$ and $\langle\{v_2\}, CS_2\rangle$, where $CS_1$ and $CS_2$ are identical, then they are replaced by a single tuple $\langle\{v_1, v_2\}, CS_1\rangle$.
- If there are two tuples $\langle\{v_1\}, CS_1\rangle$ and $\langle\{v_2\}, CS_2\rangle$, where $v_1 = v_2$ and $CS_1.descendantOf(CS_2)$, then the former tuple is removed as redundant.
- If a list of tuples on the same volatile variable set $\langle VV, CS_1\rangle, \ldots, \langle VV, CS_n\rangle$ correspond to the branches of a switch, i.e. there is an $e_{switch} = switch\{(CS_1, e_1), \ldots, (CS_n, e_n)\}$, then these are replaced with a single CS, which covers the entire switch–element. A similar process is performed for flow branches.
- If a list of tuples on the same volatile variable set $\langle VV, CS_1\rangle, \ldots, \langle VV, CS_n\rangle$ are interrelated through a sequence relation, i.e. there is a $seq\{CS_1, \ldots, CS_n\}$, then these are replaced with a single CS, which covers the entire sequence.

Algorithm 1 has been applied to the BP specification of the WMO process represented in Figure 1. The algorithm identified three volatile variables, and all five critical sections related to them. The total time for parsing the WMO process specification and computing all CSs is below 100 msec. The discovered CSs can then be projected on the Process Modeller, as presented in Figure 2.

## 5 Related work

Process interference between concurrent BPs occurs frequently in organizations, and some solutions have been provided in literature, e.g. [2, 5, 6]. Although the use of temporal logic for data-flow analysis in business processes can ensure soundness of both the control-flow and the data-flow [4], runtime disruptions due to external data changes are not accounted for. As a result, process interference can not be prevented or resolved by such methods.

However, most existing mechanisms to resolve process interference are either providing a design-time solution, thus requiring that the designer anticipates all potential problems and ways to overcome them in advance, or are based on failing processes [5]. A more elaborate solution for process interference in Service-Oriented Computing is provided by [6], where in addition to failing processes, events like exceptional conditions or unavailable activities are covered. More specifically to cloud computing, an approach for handling faults due to failing processes or services is presented by [11]. In practice, however, process interference does not necessarily cause processes to fail. Often, processes may end up with providing erroneous outcomes as a result of wrong data values, a problem that is acknowledged in [2].

Interference causes processes to provide erroneous outcomes as a result of wrong data values. In most cases, however, wrong data values are interpreted a data integrity problem. Much work has been done with respect to ensuring data integrity in distributed and concurrent systems. Some techniques for checking the integrity of distributed and dynamic data stored on the cloud are discussed in [12, 13], while [14] focus on run-time failures that affect cloud short-lived data. Although the interference problem is related to concurrent data usage, the cause of the problem is beyond data integrity issues. Therefore, we focus on problems that arise at the level of process execution due to the use of outdated data.

## 6 Concluding Remarks

One of the main challenges posed by the emergent distributed setting of modern BP Management Systems comes from the interference between different processes that access common resources. During execution of a business process, a data modification caused by some external factor may lead to erronous results, and should, therefore, be guarded and dealt with. To address this issue, the correct identification of the sections of a business process, whose correct execution depends on some volatile variable, is very important. These sections shoul be guarded upon, so that whenever a modification event is received during their execution, an appropriate intervention process is executed, in order to restore the process to a consistent state. However, the task of manual specification of these critical sections can become cumbersome and prone to errors, especially for processes with a complex structure, using many shared resources. To facilitate this task, we have developed an algorithm, which automatically computes the appropriate critical sections, given a BP specification and some semantics

regarding the input-output and the internal state variables of the service operations used by the process. We have shown how this can be applied in a real case-study taken from the Dutch e-government. The results can be presented on a process modelling tool in a graphical way, so as to assist the process designer in the specification of the necessary dependency scopes in order to ensure the delivery of correct results by the process.

## References


1. Xiao, Y., Urban, S.: Process dependencies and process interference rules for analyzing the impact of failure in a service composition environment. In: Business Inf. Systems. Volume 4439 of LNCS. (2007) 67–81
2. van Beest, N.R.T.P., Bulanov, P., Wortmann, J., Lazovik, A.: Resolving business process interference via dynamic reconfiguration. In: Proc. of 8th Int. Conf. on Service Oriented Computing (ICSOC). (2010) 47–60
3. van Beest, N.R.T.P., Szirbik, N.B., Wortmann, J.C.: Assessing the interference in concurrent business processes. In: Proc. of 12th Int. Conf. on Enterprise Information Systems (ICEIS). (2010) 261–270
4. Trčka, N., van der Aalst, W., Sidorova, N.: Data-flow anti-patterns: Discovering data-flow errors in workflows. In: Adv. Inf. Systems Eng. Volume 5565 of LNCS. (2009) 425–439
5. Xiao, Y., Urban, S.: Using data dependencies to support the recovery of concurrent processes in a service composition environment. In: Proc. of the 16th Int. Conf. on Cooperative Inf. Systems. (2008) 139–156
6. Urban, S., Gao, L., Shrestha, R., Courter, A.: The dynamics of process modeling: New directions for the use of events and rules in service-oriented computing. In: The Evolution of Conceptual Modeling. Volume 6520 of LNCS. (2011) 205–224
7. van Beest, N.R.T.P., Kaldeli, E., Bulanov, P., Wortmann, J., Lazovik, A.: Automated runtime repair of business processes. Technical Report 2012-12-2, University of Groningen (2012) www.cs.rug.nl/∼eirini/papers/tech_2012-12-2.pdf.
8. Ouvans, C., Dumas, M., ter Hofstede, A., van der Aalst, W.: From BPMN process models to BPEL web services. In: Int. Conf. on Web Services. (2006) 285–292
9. Kopp, O., Martin, D., Wutke, D., Leymann, F.: On the choice between graph-based and block-structured business process modeling languages. In: Modellierung betrieblicher Informationssysteme (MobIS 2008). Volume 141 of Lecture Notes in Informatics (LNI)., Gesellschaft für Informatik e.V. (GI) (2008) 59–72
10. Rozsnyai, S., Vecera, R., Schiefer, J., Schatten, A.: Event cloud - searching for correlated business events. In: 9th IEEE Int. Conf. on E-Commerce Technology / 4th IEEE Int. Conf. on Enterprise Computing, E-Commerce and E-Services. (2007)
11. Juhnke, E., Dornemann, T., Freisleben, B.: Fault-tolerant BPEL workflow execution via cloud-aware recovery policies. In: 35th EUROMICRO Conference on Softw. Eng. and Adv. Applications (SEAA). (2009) 31 – 38
12. Sravan Kumar, R., Saxena, A.: Data integrity proofs in cloud storage. In: 3rd Int. Conf. on Communication Systems and Networks (COMSNETS). (2011) 1 – 4
13. Hao, Z., Zhong, S., Yu, N.: A privacy-preserving remote data integrity checking protocol with data dynamics and public verifiability. IEEE Trans. on Knowledge and Data Engineering **23**(9) (2011) 1432–1437
14. Ko, S.Y., Hoque, I., Cho, B., Gupta, I.: Making cloud intermediate data fault-tolerant. In: 1st ACM Symposium on Cloud computing. (2010) 181–192