Continuous Correctness of Business Processes against Process Interference

N.R.T.P. van Beest
Department of Operations
Faculty of Economics and Business
University of Groningen

Doina Bucur
Johann Bernoulli Institute
Faculty of Mathematics and Natural Sciences
University of Groningen

***Abstract*—In distributed business process support environments, *process interference* from multiple stakeholders may cause erroneous process outcomes. Existing solutions to *detect* and *correct* interference at runtime employ formal verification and the automatic generation of intervention processes at runtime. However, these solutions are limited in their generality: they cannot cope with interference occurring during the entire runtime of the business process, some of which is unknown at design time. In this paper, we present an automated framework for the runtime verification and correction of business processes against data interference, which guarantees continuous correctness of process execution in any distributed environment. The continuous detection of interference during process execution is achieved on-the-fly using temporal verification of the running process together with virtual external data transactions; this identifies a minimal set of process *checkpoints* where the execution of external transactions would change the outcome of the local process. To achieve continuous correction, runtime checks are made at these process checkpoints, and, whenever interference occurs, an *intervention process* is generated to correct the local process from its current state. Subsequently, any intervention process is itself continuously verified and corrected, thus guaranteeing correct execution throughout the lifetime of a process. The approach is evaluated on a real case-study from the Dutch e-Government.**

I. Introduction

Business process support is increasingly provided through distributed service-oriented information systems, in order to achieve higher flexibility and business agility. In such an environment, business processes can no longer be considered in isolation, since data resources are not necessarily proprietary to the organization and are no longer used exclusively by a single business process. As a consequence, data modifications by one process may affect one or more other concurrently executing processes, which may cause an undesired process outcome for one or more of these processes. This is referred to as *process interference* [1][2].

Process interference is a massive problem in business process support [3], in particular in heavily distributed service environments. The disruptions caused by process interference have a considerable effect in the real world, leading to unacceptable situations from a business perspective (i.e. wrong invoices, ordering new products with wrong customer data, etc.). There exist runtime solutions to resolve process interference, by on-the-fly automatic generation of an *intervention process* (IP) [2], which is triggered by an external data change; this process brings the original process back to a correct state, after the external data change. The execution thus continues with the intervention process, followed by resuming the execution of the original business process from the corrected state.

However, this solution does not provide a full guarantee that the resulting process is correct: (i) intervention processes cannot be generated in such a way that they are resilient against any form of interference, since external data changes cannot be predicted, and (ii) intervention processes themselves are not monitored for external data changes other than the one that triggered the generation of that intervention process. As a result, the ensuing process may still be liable to interference, particularly in complex and dynamic service environments.

In this paper, a general runtime framework is presented for the monitoring and correction of a business process executing in a dynamic business environment prone to process interference. We categorize operations upon data effected by a business process into the following types: READ and WRITE operations access from and modify data into a database, and USE operations are reads or write in the local variable cache of the process. We use the formal definition of process interference expressed in a Linear Temporal Logic safety formula over READ, WRITE, and USE data operations, as in [3]; essentially, this definition models process interference as a concurrent process execution trace in which one of the processes unknowingly uses data previously read from a database, whose database value has been in the meantime updated by the other process.

Based on this definition, we design the runtime framework for correct process execution as follows:

1) *(Determining monitoring checkpoints)* Given a process design, for all data which the process reads from a database, a sound, yet minimal, set of program checkpoints is automatically calculated on-the-fly for the business process. The calculation is done by formally model-checking the process, placed in an environment of concurrent, virtual external processes, against the formal definition of process interference; all program points where a violation is reported by the verification process becomes a checkpoint.
2) *(Monitoring and correcting process execution)* At these checkpoints, runtime checks are inserted in the process execution to determine whether an external data modification does occur in reality. (a) If such a modification does not occur, the process resumes execution. (b) If a data update does happen, the process has to reach a correct state before resuming execution.

For this, an intervention process is generated, also on-the-fly. The generation method is orthogonal to our framework, and any such method may be used; here, we incorporate an algorithm based on automated planning from [2]. The newly generated intervention process is then treated similarly to the original process, with first determining monitoring checkpoints on-the-fly, then monitoring and correcting its own execution.

This integrated approach has the following advantages over the existing method of automated dependency scope specification [4]: critical sections of a business process are defined strictly based on their liability to process interference, rather than purely based on data usage. As such, a far more efficient runtime environment is obtained. Secondly, this integrated approach covers the execution of the process throughout its entire lifecycle. That is, any intervention process is also verified for and corrected against interference. Consequently, continuous correctness can be ensured.

Accordingly, the paper is structured as follows: in Section II, the related work is discussed. Next, in Section III, a formal definition of process interference is provided. This definition is the basis for the continuous correctness framework, which is described in Section IV. The developed framework is applied to a real case-study from the Dutch e-Government, and evaluated in Section V. Finally, a discussion is provided in Section VI.

II. Background

Concurrent data access requires runtime consistency checking and management of data transactions to maintain data integrity [5]. In the study of multiple concurrent access for database systems, the focus has been primarily on implementing ACID transaction semantics (for a review, see [6]). However, the problem of process interference is not centered around the value that is stored, but the value that is used by the business process. More specifically, the business process may implicitly depend on the value of a process variable, which is assumed not to change. Consequently, ensuring consistency between the internal data representation and the external business reality is more complex and cannot be easily resolved by these transaction correctness properties.

Checking business processes for process interference is a specialized form of compliance checking. Of the many existing frameworks for model checking business processes against temporal compliance properties other than process interference, [7] supports generic temporal runtime checking for business processes expressed in a purely declarative description language, with a number of other solutions for different process description languages and specification formalisms. In [8], temporal logic is used for data-flow analysis in business processes to ensure soundness of both the control-flow and the data-flow. In [9], an extension of workflow with data operations is presented, in order to provide a precise soundness analysis of a workflow. However, these verification techniques analyze processes and their data-flow in isolation. Consequently, no external processes and data changes are taken into account.

While such methods for compliance checking may be adapted for detecting process interference at runtime, they do not provide a framework which also includes process recovery after a runtime error. In [7], only simplistic mechanisms are supported for recovery after a compliance violation; recovery amounts to ignoring or resetting process events or compliance constraints.

Adding means of process recovery from errors amounts to creating a framework for runtime process flexibility, i.e., "the ability to deal with both foreseen and unforeseen changes, by varying or adapting those parts of the business process that are affected by them, whilst retaining the essential format of those parts that are not impacted by the variations" [10]. Our framework is of the type *flexibility by change*, as defined in [10], which "requires the addition or removal of tasks or links from the process model". Flexibility by change is also supported in [11], with recovery processes which must be predefined.

In the context of process recovery from interference, the errors are not necessarily caused by failures. Rather, interference causes process to terminate but provide erroneous outcomes. As such, a dynamic framework is required that is able to anticipate on *any* concurrent data change and *any* potential subsequent disturbance. An automated framework for recovering from interference is presented in [2], where intervention processes are introduced to finish the process in a consistent state. This framework, however, has only limited support for ensuring correctness of the intervention processes, as no external data changes are monitored other than the one that triggered the generation of that intervention process.

III. Formal definition for process interference

Our runtime framework for continuous correctness addresses the real-world occurrence of data interference among business processes executed in an asynchronous distributed environment. In this section, we recall the fundamental concept of process interference [1][2] using temporal logic, in conjunction with a more intuitive graphical example.

A. Graphical example

In a database environment, CREATE, READ, UPDATE and DELETE are the four basic data operators [12]. In our framework, a business activity calls a READ operator for retrieving data from the database to the local variable cache used by the process, whereas CREATE, UPDATE and DELETE are modeled here by a single WRITE operator, which stores data from the local variable cache into the database. Furthermore, operations which *use* the data locally for writing to another local variable are modeled as USE operators. The data operators are described graphically in Fig. 1.

In Fig. 2, we show an example of interference. Two independent concurrent processes use a common data store. Activity A1 reads d. Subsequently, activity A2 reads a using the value of d. Finally, activity A3 uses the value of a to write to e. Implicitly, the value of e is by transitivity dependent on d. If a concurrent process (e.g., activity B2) updates the value of d *after* it has been read by Process 1 in the initial activity A1, then e may hold an outdated value.

Furthermore, an external data change may also affect subsequent subprocesses after the evaluation of a condition.

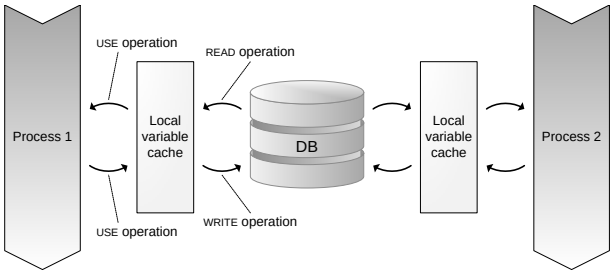

Fig. 1: READ, WRITE and USE operations explained.

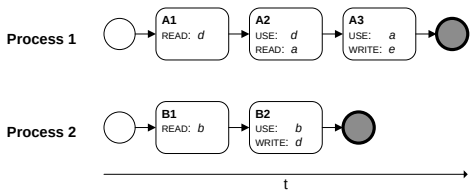

Fig. 2: Linear business process with concurrent data change.

In Fig. 3, for example, the XOR-split decision is made based on reading the value of d. This determines whether A2 or A3 is executed. If d is updated by Process 2 just after the execution of A1, the wrong branch of activities may be executed.

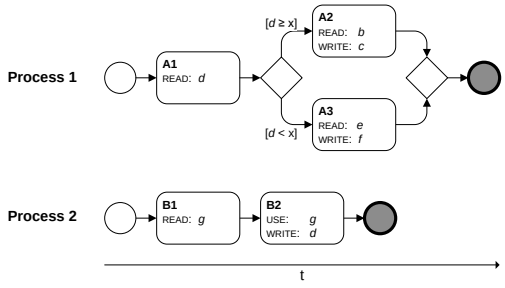

Fig. 3: Conditional branches with concurrent data change.

B. Specifying process interference using temporal logic

LTL (Linear Temporal Logic) is a temporal logic providing linear-time operators over an execution path of a process model [13]. LTL provides the following core temporal operators:

Fϕ	Finally: ϕ eventually must hold.
ϕUψ	Until: ϕ has to hold at least until ψ holds.

Definition 1 (LTL syntax): Given a finite set of atomic propositions AP, well-formed LTL formulas are generated by the following grammar:

$$\phi ::= p \mid (\neg\phi) \mid (\phi \wedge \psi) \mid (\phi \vee \psi) \mid \mathrm{F}\phi \mid \phi \mathrm{U} \psi$$

where $p \in AP$ and ϕ, ψ are LTL formulas.

We denote a process execution path π by an ordered sequence of labels $\pi = x_0, x_1 \ldots$. Given an execution path $\pi = x_0, x_1 \ldots$, where $x_i \in 2^{AP}$, the semantics of an LTL formula ϕ state what conditions must be met for ϕ to hold over π, i.e., when $\pi \models \phi$ is true. We write $\pi[i]$ for the execution step i in π. We write $\pi[i:j]$ for the execution of steps i to j in π, and $\pi_1 \cdot \pi_2$ for the concatenation of two execution paths.

Given any LTL formulas ϕ, ψ, the definition of the *validity* (denoted by $\models$) of composed LTL formulas over a given process execution path π is defined inductively in [13].

Using LTL, the temporal characteristics of process interference between two concurrently executed processes can now be defined and verified formally. For the remainder of the paper, the READ, WRITE and USE operations of $process_i$ on data element $d \in D$ are denoted by $\mathbf{r}_i(d)$, $\mathbf{w}_i(d)$ and $\mathbf{u}_i(d)$ respectively. Then, for a given data set D, the corresponding set of atomic propositions AP for $process_i$ is defined as

$$AP := \bigcup_{d \in D} \{\mathbf{r}_i(d), \mathbf{w}_i(d), \mathbf{u}_i(d)\}$$

Definition 2 (Process Interference): If two distinct processes $process_1$ and $process_2$ are executed concurrently, process interference occurs on the execution path where the following LTL formula is valid:

$$F\Big[\mathbf{r}_1(d) \wedge F\big[\mathbf{w}_2(d) \wedge \big(\neg\mathbf{r}_1(d)\ U\ (\mathbf{u}_1(d) \vee \mathbf{w}_1(d))\big)\big]\Big]$$

Intuitively, two processes interfere if there exists an execution path where d is read ($\mathbf{r}_1(d)$) and subsequently either used ($\mathbf{u}_1(d)$) or written back ($\mathbf{w}_1(d)$) by one process, with the other process overwriting d ($\mathbf{w}_2(d)$) between the two operations of the first process.

IV. Framework for continuous correctness of process execution against process interference

We use the definition of process interference to identify critical sections that are vulnerable to interference, in a given business process, and for any distributed execution environment. For this, the business process needs to be verified, against the specification given by Definition 2, together with an artificial *virtual process* which "models" an external data change. This verification step uncovers concrete interference cases, which are used to position *process checkpoints*.

Take a branching business process P. Our runtime framework consists of the following iteratively executed steps:

- *Compose interleavings.*
  Virtual interleavings are composed from all possible serializations of the process P with all possible virtual processes consisting of a single activity calling a WRITE data operator.
- *LTL verification for process interference.*
  Each virtual interleaving is verified against process interference (as per Definition 2).
- *Derive checkpoints for the original process.*
  The subset of virtual interleavings, which are validated in the verification step as interference, are used to automatically generate checkpoints for the original branching process P; these are the process points where the process is vulnerable to external data changes.

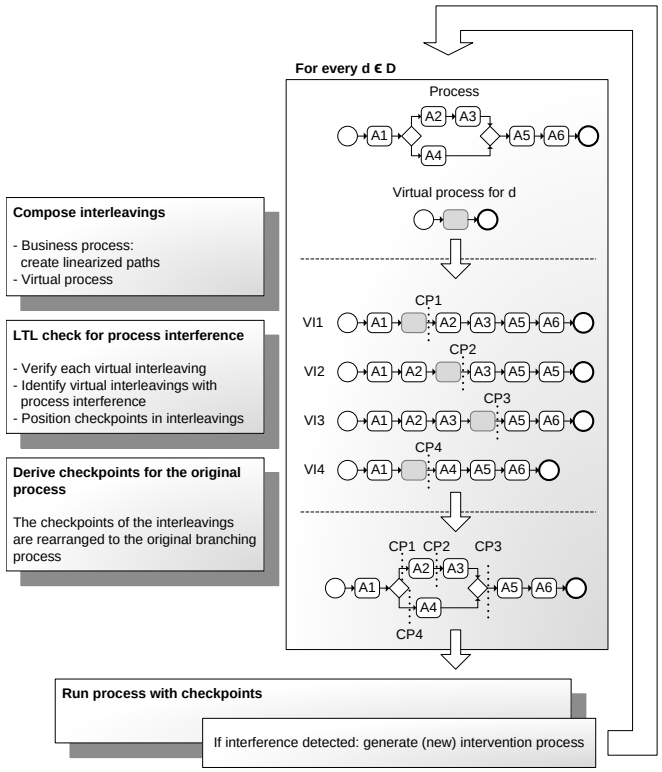

Fig. 4: Runtime framework for continuous correctness.

- *Run the process with monitoring at checkpoints.* The process P is executed with added monitoring for data changes at the checkpoints. If interference is detected during execution, an intervention process is generated on-the-fly. This new process is then itself analyzed, monitored and executed according to this framework.

The runtime framework is shown graphically in Fig. 4. The steps of the framework are detailed in the next subsections.

A. Compose virtual interleavings

Prior to composing the virtual interleavings, a sound syntactic serialization of the process is computed. That is, if the original branching process may cause process interference when ran concurrently with another process operating upon the same data $d \in D$, then the serialized version of the process should also preserve this behaviour.

To serialize a single branching process, we do the following translations:

- A XOR split conditional over the value of d is translated into a new activity, which reads d.
- XOR splits are linearized by generating a linear process for each XOR branch.
- AND splits are serialized by generating all possible interleavings among the parallel execution branches.

Furthermore, we serialize all loops in a process by the *bounded unwinding* of each loop: an unwinding depth of two is provably sufficient to preserve interference errors. Here, we give an intuitive summary of the proof. Any process P with a loop is effectively the concatenation of three subprocesses $P_{init}P^{\omega}_{loop}P_{final}$, where the ω notation P^{ω}_{loop} indicates the potentially infinite execution of the loop subprocess. The same process after a bounded unwinding with depth two is denoted by $P_{init}P^{2}_{loop}P_{final}$. To prove that bounded unwinding preserves errors, we prove that: whenever the infinitely looping process $P_{init}P^{\omega}_{loop}P_{final}$ interferes with any given virtual process, then also $P_{init}P^{2}_{loop}P_{final}$ interferes. To prove this, first we prove that any process will interfere with a virtual process for a given d if and only if the process has an execution in which $\mathbf{r}_1(d)$ occurs, followed later by $(\mathbf{u}_1(d) \vee \mathbf{w}_1(d))$, with $\neg\mathbf{r}_1(d)$ true in all intermediate activities. Then, we can prove that, whenever this execution occurs in the infinitely looping process, it also occurs in the process with bounded unwinding, by iterating through all possible orders of finding these activities in the subprocesses P_{init}, P_{loop}, and P_{final}.

Subsequently, a formal model of the serialized process can be extracted for the purpose of LTL checking: this formal model is an execution path $\pi = x_0, x_1 \ldots$, i.e., a sequence of activity *labels* $x_k \in 2^{AP}, \forall k \geq 0$, where the set of atomic propositions AP for the process is the set of all data operators upon all data data $d \in D$, as defined in Section III-B. An activity label x_k thus records all data operations executed by the kth activity in a sequential process execution.

For each $d \in D$, a *virtual process* is created out of a single activity effecting a WRITE operation upon d. This models, minimally, a single interference from an external process upon shared data.

The serialized execution paths of the original process, and the set of virtual processes are composed into a set of all possible *virtual interleavings*. For this, first all linearized paths are obtained from the original process P. Each virtual activity is then interleaved with each of P's linear executions, by inserting the virtual activity between any two consecutive activities in a path. As each interleaving comprises only a single virtual activity, the amount of interleavings can be calculated as follows: $|interleavings| = |D| * |path|$.

B. LTL verification for process interference

Each virtual interleaving consists of activities from the process and a virtual activity containing a WRITE operation on a data element d. Subsequently, each virtual interleaving is verified against the LTL Definition 2 for process interference. If a virtual interleaving is found to show interference, a checkpoint is positioned in the original business process, at the precise location of the virtual activity found through formal verification to be able to cause interference.

Algorithm 1 shows the verification of virtual interleavings, followed by placing a checkpoint on each interfering interleaving. This algorithm takes as input the set of interleavings as composed above (Section IV-A). Next, a specialized check for the LTL specification for interference is implemented, which corresponds to the second step in our framework (Fig. 4). We choose to implement and optimize this specialized model-checking algorithm instead of using an off-the-shelf model-checker for reasons of efficiency.

Algorithm 1 Calculate checkpoint positions using LTL verification.

```
1:  procedure CALCULATECHECKPOINTS(set of interleavings G)
2:    for each g ∈ G do
3:      tmp = false
4:      virtualpos = findGhostActPos(g)
5:      data = getGhostDataElement(g)
6:      pos = virtualpos
7:      while (pos ≥ 0) ∧ (¬g[pos].reads(data)) do
8:        pos = pos - 1
9:      end while
10:     if pos ≥ 0 then
11:       pos = virtualpos + 1
12:       while (pos < |g|) ∧ (¬g[pos].uses(data)) do
13:         pos = pos + 1
14:       end while
15:       if pos < |g| then
16:         for r = virtualpos ... pos do
17:           if g[r].reads(data) then
18:             tmp = true
19:           end if
20:         end for
21:         if ¬tmp then
22:           g.addCheckpoint(virtualpos, data)
23:         end if
24:       end if
25:     end if
26:   end for
27: end procedure
```

First, the position of the virtual activity is obtained (line 4) and stored in $virtualpos$. Next, the data element affected by the WRITE operation of the virtual activity is stored in $data$. The algorithm proceeds by searching for a READ operation on $data$ before the virtual activity (line 7). If the READ activity is found, the matching USE or WRITE operation is located (line 12). If a USE/WRITE operation exists in the interleaving g, the activities between the virtual activity and the USE/WRITE activities are checked (line 16, as none of these are allowed to have a READ operation according to Definition 2. Finally, a checkpoint is added in g after the virtual activity (line 22).

C. Derive checkpoints for the original process

Checkpoints are added in Algorithm 1 to the linear virtual interleavings. However, the business process that is executed may include branching. Consequently, the checkpoints need to be combined and rearranged to be used in the original process. The placing of the checkpoints into the original business process is done by Algorithm 2.

First, all checkpoints from all interleavings are added to the business process P (line 4), at the same position as they were in the respective interleaving (i.e. before the respective activity). For the purpose of readability, the details of the procedure $addCheckpoint$ are omitted. Subsequently, overlapping checkpoints (i.e., checkpoints on different data elements, but on the same position in the process) need to be merged into a single checkpoint with multiple data elements (line 12). Finally, multiple checkpoints on the same position and on the same data are merged into one single checkpoint (line 14).

D. Run the process with monitoring at checkpoints

The framework presented in Fig. 4 shows the annotation of business processes with checkpoints, to ensure continuous correctness during execution. In Fig. 5, the architecture incorporating the framework is represented graphically.

Algorithm 2 Placing checkpoints in the business process.

```
1:  procedure PLACECHECKPOINTS(set of interleavings G, process P)
2:    for each g ∈ G do
3:      for each cp ∈ g.checkpoints do
4:        P.addCheckpoint(cp)
5:      end for
6:    end for
7:    cp = P.checkpoints
8:    for i = 0 to |cp| - 1 do
9:      for j = i + 1 to |cp| do
10:       if cp[i].position = cp[j].position then
11:         if cp[i].data ≠ cp[j].data then
12:           cp[i].addDataElement(cp[j].data)
13:         end if
14:         P.remove(cp[j])
15:         j = j - 1
16:       end if
17:     end for
18:   end for
19: end procedure
```

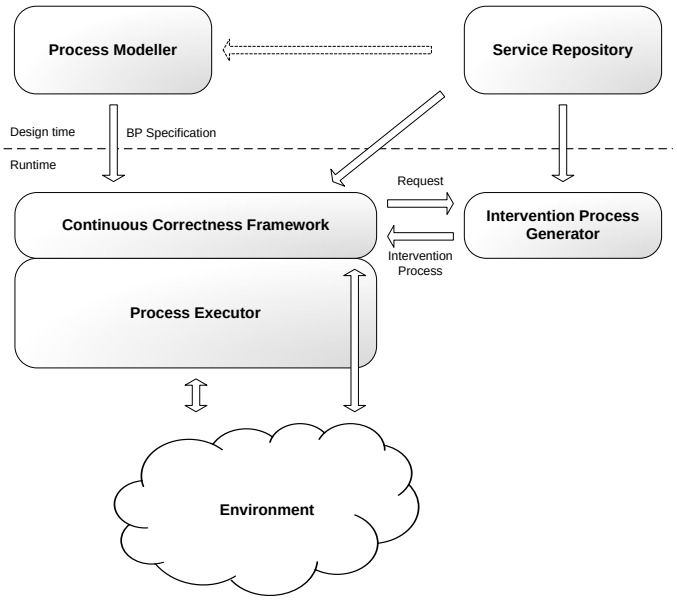

Fig. 5: Architectural overview of the framework.

The *Process Modeller* (PM) provides the design tools for the specification of the business process. The business process modelled by the PM uses activities that are available in the *Service Repository* (SR) by means of service operations. The SR contains a list of service instances (providers) that offer a set of service operations. Each service instance implements a service type, which specifies the interface of the service.

The *Continuous Correctness Framework* (CCF) provides each business process designed by the PM with a set of checkpoints as shown in Fig. 4. The *Process Executor* (PE) is responsible for executing the annotated business processes, and takes care of the discovery, binding and invocation of the respective service operations residing in the *Environment*, as specified in the SR. The CCF is responsible for runtime monitoring of the checkpoints. As such, the CCF determines

for each checkpoint preceding an activity, whether the current state indicates a modification of one of the data elements covered by the checkpoint. If a modification is detected, the CCF triggers the *Intervention Process Generator* (IPG), which automatically generates an intervention process based on the current state of the process, as described in [2]. The intervention process restores the consistent state of a business process, by taking into account the characteristics of the business process in execution, the available compensation activities, and the properties that have to be fulfilled to recover from the erroneous situation. The generated intervention process is sent back to the CCF, which annotates it with checkpoints prior to being executed by the PE.

V. Framework application

In this section, the framework is evaluated using a real case study from the Dutch e-Government. Using the framework, the process is annotated with checkpoints and provided with dependency scopes accordingly. Disruptive situations are simulated and the automated recovery provided by the framework is assessed.

A. Case study description

The business process used for the evaluation concerns (a simplified version of) the handling of the requests from citizens according to the Dutch law of societal support (WMO law in Dutch) at one of the 408 municipalities in the Netherlands. We refer to this process as the WMO process.

Municipalities are obliged to have a WMO service desk, where the citizens can access all provisions under that law. The WMO process (shown in Fig. 6) is initiated by a citizen, who can apply for a provision at the local service desk or online. After receiving the application at the municipality office, the situation and the current living conditions of the citizen are investigated by means of a home visit. If the home visit is not sufficient to obtain all required information (concerning, e.g., the citizen's health), medical advice can be requested from a specialist. Subsequently, the municipality decides based on this information whether the citizen is eligible to receive the requested provision or not.

If the application is rejected, the citizen has the possibility for appeal. In case of a successful appeal, the provision is either granted as initially requested, or the process is restarted. If the application is approved, the appropriate activities are executed, depending on the requested provision. For domestic help, suppliers are contacted who can take care of the provision. A home modification involves a tender procedure to select a supplier who will be contracted for the execution of the home modification. A wheelchair is usually provided using a supplier contracted by the municipality. After acquiring the detailed requirements, the order is sent to the supplier. Next, the supplier delivers the provision and sends an invoice to the municipality. Finally, the invoice is checked and paid.

B. Derive checkpoints for the WMO process

The following data elements were considered in the WMO process: requested provision, address, requirements, medical condition, decision, marital status, and invoice details. The outcome of the WMO process is determined based on this data. Moreover, any of these variables may be updated via a process triggered concurrently by the citizen.

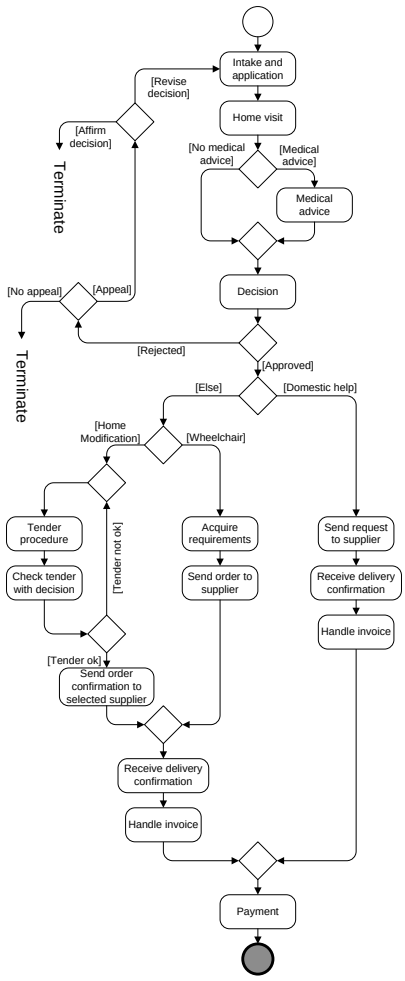

Fig. 6: WMO process.

The WMO process consists of 15 activities, which required 121 interleavings in total to verify for interference and generate all checkpoints. The request for a wheelchair required 35 interleavings with 5 different virtual activities. The request for home modification required 50 interleavings with 5 virtual activities. The request for domestic help required 36 interleavings with 6 virtual activities. The generated checkpoints are represented graphically in Fig. 7.

C. Simulating problematic situations

In order to test the effectiveness of the developed framework, a number of distinct disruptive events have been simulated during the execution of the WMO process. The scenario concerns a request for a wheelchair with interference caused by an address change: after sending the order to the supplier, the citizen decided to move. The interference is detected due to the monitoring checkpoint for the citizen's address having been placed accordingly in the WMO process (as shown in Fig. 7). This interference event has three severe potential consequences: (i) the wheelchair may be delivered at the wrong address, (ii) the specifications of the wheelchair may not match

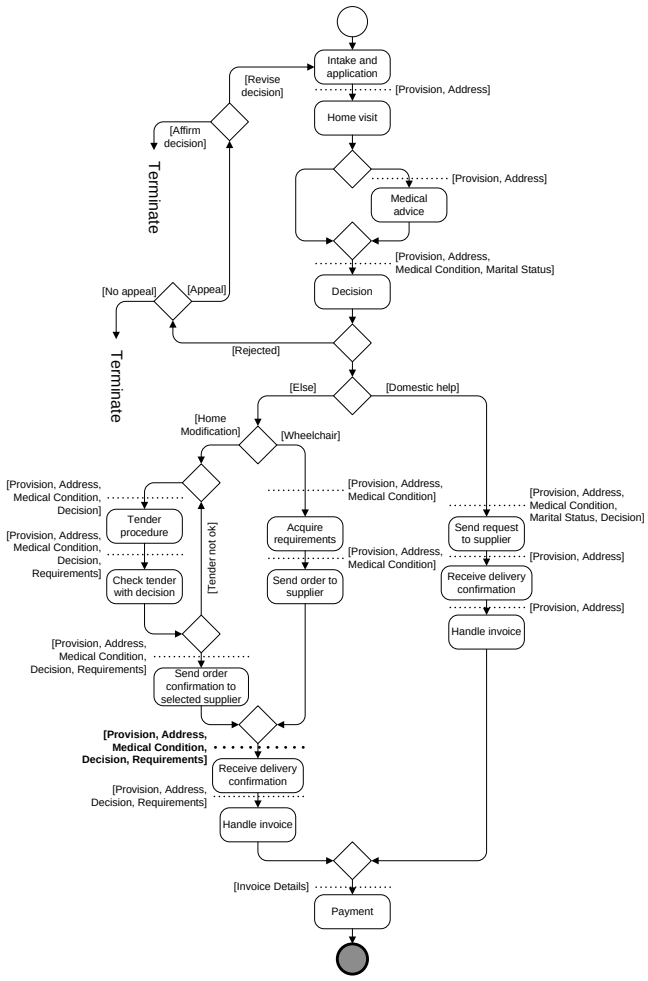

Fig. 7: WMO process including checkpoints.

the new residence (e.g., the door may be too narrow), and (iii) the citizen may not be eligible anymore for a custom wheelchair (e.g., due to a move to a nursing home).

To resolve this interference, an intervention process is generated. Fig. 8 shows the intervention process as generated by the IPG (see Fig. 5), including monitoring checkpoints generated and placed. Note that the checkpoints differ from Fig. 7, as the intervention process only concerns the request for a wheelchair. That is, it is not necessary to include checkpoints specific for the home modification.

First of all, the current order should be cancelled. Both a home visit and a medical advice may be necessary, in particular if the citizen has moved to, for instance, a nursing home. In case of a positive decision, the new requirements are acquired again and a new order is sent.

During execution of the intervention process, a second disruptive event was introduced. In this event, the medical condition of the citizen changes after 'Acquire requirements'. Such a change (distinct from the first change) may imply different requirements or additional requirements. As a result, the current specification of the wheelchair is potentially incorrect. Consequently, a new intervention processes is generated to

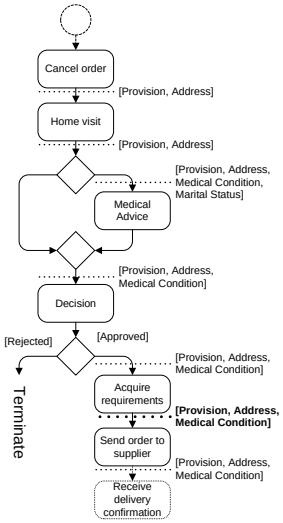

Fig. 8: Generated intervention process with checkpoints.

resolve this new disruptive situation. The new intervention process is shown in Fig. 9. After execution of the new intervention process, the original process P resumes its execution after the last checkpoint on that data element. In Fig. 7, this is right before 'Handle Invoice'.

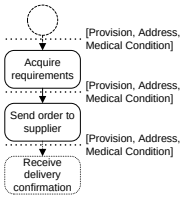

Fig. 9: New intervention process with checkpoints.

From this example, it becomes evident that relatively small and very realistic data changes can result in severely distruptive behaviour of a business process. The framework here clearly shows its value by continuously ensuring a correct execution of the process.

D. Performance

In order to test the developed framework with respect to performance, the WMO process shown in Fig. 6 was modelled and automatically annotated with checkpoints (shown in Fig. 7). All tests presented were performed on a computer with an Intel Core i7 processor @2,1GHz, with 8GB of RAM, running Java 1.7.0 09.

Table I provides an overview of the times required to generate checkpoints for all branches of the WMO process. In all cases, the time for analyzing the respective branch is below 2 milliseconds and, therefore, negligible.

The generated intervention process (Fig. 8) required 0.5 sec. The subsequent generation of the new intervention process (Fig. 9) required 0.2 sec. Subsequently, the scalability of our framework was evaluated with respect to the size of the

Path	|Path|	Virt.Act.	Interleavings	Time (ms)
Wheelchair	9	5	35	1.3
Home Mod.	10	5	50	1.7
Domestic Help	8	6	36	1.3

TABLE I: Performance results for checkpoint generation of the WMO process.

monitored process. As such, a number of further tests have been performed using an artificially constructed process with an increasing size from 25 to 100 activities. The results of these tests are summarized in Table II. The tests show that even for a linearized path of 100 activities and 594 interleavings, still less than 50ms is required to generate all checkpoints.

|Path|	Virt.Act.	Interleavings	Time (ms)
100	6	594	45.4
50	6	294	22.4
25	6	144	11.3

TABLE II: Performance results for checkpoint generation of large processes.

As shown in the tables above, the performance for analyzing a linearized path is linear with respect to the amount of interleavings, which is calculated as follows (see Section IV-A): $|interleavings| = |D| * |path|$.

VI. Discussion and conclusion

One of the main challenges of modern distributed service-oriented information systems comes from interference between concurrent processes that access common dataresources. In this paper, we have presented a novel runtime framework for the monitoring and correction of a business process executing, to ensure continuous correctness. The framework employs temporal verification of the running process together with virtual external data transactions; this identifies a set of process *checkpoints* where the execution of external data changes could lead to interference. To achieve continuous correction, runtime checks are made at these process checkpoints, and, whenever interference occurs, an *intervention process* is generated to restore consistency.

This integrated approach has the following advantages over the existing methods: critical sections of a business process are defined strictly based on their liability to process interference, obtaining a far more efficient runtime environment than existing approaches. Furthermore, this integrated approach covers the execution of the process throughout its entire lifecycle. As such, any intervention process is also verified for and corrected against interference, which ensures continuous correctness.

To evaluate the feasibility of the approach, an architecture has been designed and a prototype has been implemented. The framework has been applied to a real case study, involving the business process of the Dutch WMO law. The case study showed the ability of our framework to prevent severely distruptive behaviour of the business process. As such, the framework here clearly showed its value by continuously ensuring a correct execution of the process.

Although this paper has a primary focus on providing resilience against process interference, the overall approach can be applied more generically to recover from any possible runtime disturbance. As such, the presented framework provides a valuable contribution for building reliable and robust service-oriented information systems. It becomes increasingy important to design and deploy fully automated, correct-by-design runtime frameworks for business processes in practice, particularly as service environments will evolve towards more distributed and cloud-based enviroments.

Acknowledgments

This research is funded by the Flexigrid project, as part of the Edgar-program. The developed tools and techniques in this paper will be applied in the energy-sector.

References

[1] Y. Xiao and S. D. Urban, "Process dependencies and process interference rules for analyzing the impact of failure in a service composition environment," in *Proceedings of the 10th International Conference on Business Information Systems*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2007, vol. 4439, pp. 67–81.

[2] N. R. T. P. Van Beest, E. Kaldeli, P. Bulanov, J. C. Wortmann, and A. Lazovik, "Automated runtime repair of business processes," *Information Systems*, 2013, in print.

[3] N. R. T. P. Van Beest, D. Bucur, J. Wortmann, and A. Lazovik, "An automated analysis of process interference verified with LTL checking," University of Groningen, Tech. Rep. 2013-08-07, 2013, www.cs.rug.nl/~doina/TR/TR-2013-08-07.pdf.

[4] N. R. T. P. Van Beest, E. Kaldeli, P. Bulanov, J. C. Wortmann, and A. Lazovik, "Automatic detection of business process interference," in *International Workshop on Knowledge-Intensive Business Processes (KiBP'12), Rome, Italy.*, 2012, invited paper.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[6] Y. Xiao, S. D. Urban, and S. Dietrich, "A process history capture system for analysis of data dependencies in concurrent process execution," in *Data Engineering Issues in E-Commerce and Services*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006, vol. 4055, pp. 152–166.

[7] F. M. Maggi, M. Montali, M. Westergaard, and W. M. P. Van Der Aalst, "Monitoring business constraints with linear temporal logic: An approach based on colored automata," in *Business Process Management*, ser. Lecture Notes in Computer Science, S. Rinderle-Ma, F. Toumani, and K. Wolf, Eds. Springer Berlin Heidelberg, 2011, vol. 6896, pp. 132–147.

[8] N. Trčka, W. M. P. Van Der Aalst, and N. Sidorova, "Data-flow anti-patterns: Discovering data-flow errors in workflows," in *Proceedings of the 21st International Conference on Advanced Information Systems Engineering (CAiSE)*, ser. Lecture Notes in Computer Science, vol. 5565. Springer-Verlag, 2009, pp. 425–439.

[9] N. Sidorova, C. Stahl, and N. Trčka, "Soundness verification for conceptual workflow nets with data: Early detection of errors with the most precision possible," *Information Systems*, vol. 36, no. 7, pp. 1026–1043, 2011.

[10] H. Schonenberg, R. Mans, N. Russell, N. Mulyar, and W. M. P. Van Der Aalst, "Process flexibility: A survey of contemporary approaches," in *Advances in Enterprise Engineering I*, ser. Lecture Notes in Business Information Processing, J. Dietz, A. Albani, and J. Barjis, Eds. Springer Berlin Heidelberg, 2008, vol. 10, pp. 16–30.

[11] M. Adams, A. H. M. Hofstede, W. M. P. Van Der Aalst, and D. Edmond, "Dynamic, extensible and context-aware exception handling for workflows," in *Proceedings of the OTM Conference on Cooperative information Systems (CoopIS)*, ser. Lecture Notes in Computer Science, R. Meersman and Z. Tari, Eds. Springer Berlin Heidelberg, 2007, vol. 4803, pp. 95–112.

[12] J. Martin, *Managing the Data-base Environment*. Englewood Cliffs, New Jersey: Prentice-Hall, 1983.

[13] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, Cambridge, Massachusetts and London, UK, 1999.