Instance Migration Validity for Dynamic Evolution of Data-Aware Processes

Wei Song, Member, IEEE, Xiaoxing Ma, Member, IEEE, and Hans-Arno Jacobsen, Senior Member, IEEE

Abstract—Likely more than many other software artifacts, business processes constantly evolve to adapt to ever changing application requirements. To enable dynamic process evolution, where changes are applied to in-flight processes, running process instances have to be migrated. On the one hand, as many instances as possible should be migrated to the changed process. On the other hand, the validity to migrate an instance should be guaranteed to avoid introducing dynamic change bugs after migration. As our theoretical results show, when the state of variables is taken into account, migration validity of data-aware process instances is undecidable. Based on the trace of an instance, existing approaches leverage trace replaying to check migration validity. However, they err on the side of caution, not identifying many instances as potentially safe to migrate. We present a more relaxed migration validity checking approach based on the dependence graph of a trace. We evaluate effectiveness and efficiency of our approach experimentally showing that it allows for more instances to safely migrate than for existing approaches and that it scales in the number of instances checked.

Index Terms—Data-aware process, dynamic evolution, instance migration, migration validity, trace slicing

1 INTRODUCTION

Proceses are widely used in inter-organizational applications in networked environments, e.g., in business process management [1], [2] and service compositions [3], [4], [5], scientific and cloud workflows [6], [7], [8], Web applications and Internetware [9], [10], [11], to name a few. Since networked environments are open, dynamic, and uncontrolled [12], [10], [9], processes must often evolve to adapt to new business requirements, strategies, and changing customer needs. Process evolution may also result from fixing bugs and from process enhancements [13]. It is considered a key challenge to handle in-flight process instances when a process is updated from its current version to a new version [14], [15], [16], [17], [18]. Below, we often refer to the old and the new versions of the process; the new process version results from changes applied to the old one.

A naive strategy is to withdraw the running instances and restart them from scratch according to the new process version. Despite its simplicity, this strategy may be inapplicable in practice for several reasons. First, this strategy is not an option for handling migration of mission-critical processes (e.g., an online auction service, real-time monitoring) [19]. Otherwise, severe or unacceptable consequences may ensue. Also, the inefficiency of this strategy may not be acceptable even for ordinary business processes and applications [20]. For example, if a process instance is in a non-compensable state [21], it might be infeasible to restart the instance. Finally, it is both time-consuming and cost-ineffective to restart a running instance according to the new version, particularly when the process is long-running [15].

A more promising strategy is to migrate running instances dynamically to the new process version. This mechanism is referred to as instance migration [14], [15], [16]. The crux of instance migration is to ensure migration validity, that is, to determine whether there is a state in the new process version that is consistent with the current state of the instance in the old version [15], [22]. Provided that migration validity is met, after the instance migrates to the new version, it behaves as if it had been executing according to the new version from the respective state.

Since executable processes involve control flow, data (variables), time-related properties, environmental properties, and access control policies, all these elements should be considered for instance migration. However, control flow and data (variables) are the most fundamental elements [23], [24], [25], [26], [27]. Therefore, in this work, we focus on instance migration validity of data-aware processes [23], [24], [27], where besides control flow, variables read and written are indispensable for migration validity checking. Specifically, variable state (i.e. the current value of a process variable at the migration time) needs consideration in the instance state. Unfortunately, as we establish in this paper, it is generally undecidable to check migration validity when variable state is taken into account. We elaborate on this result in Section 4.1. Undecidability implies that it is impossible to find a computable sufficient and necessary condition to check instance migration validity. As a consequence, the only option we have is to find sufficient conditions to tackle this problem. The sufficient conditions can be used as the migration criteria. Note that the updated process version is assumed to be sound (i.e., it is free of errors such as deadlocks, livelocks, etc.) [28]. Thus, if an instance satisfies the migration criterion, its migration does not introduce dynamic change bugs [15], [16], because the subsequent behaviour of the migrated instance is just similar to the behaviour of a process instance of the new version.

In practice, it is desirable to migrate as many running
instances to the new process version as possible. The reasons are two-fold. First, this allows migrated instances to benefit from the advantages and opportunities of process evolution. Second, this lightens the burden of maintenance from the perspective of process holders (process management systems) [15]. Specifically, if more instances are migrated, the old process version can be unmaintained in a short time so that subsequent changes can only be applied to the latest version. Nevertheless, it is challenging to find a migration criterion that is as relaxed as possible (i.e., allows to safely migrate as many instances as possible.) Most existing approaches simplify this problem by merely focusing on control-flow state of the running instance while variable state is not taken into account [14], [15], [17], [29], [30], [31]. Due to neglecting to consider variable state, these approaches may lead to both false negatives (i.e., reject instances as invalid, while, in essence, they could have been safely migrated) and false positives (i.e., allow instances to be migrated, whereas, in essence, the migrations are not valid) for migration validity checking. Although some work realizes the necessity of considering the variable state in migration validity checking [32], to the best of our knowledge, few approaches adopt variable state in the definition of migration validity, and no solution is known to us that checks whether a process instance ensures variable consistency before and after migration.

Our goal is to determine a more relaxed migration criterion such that migration validity is ensured, on the one hand, and more process instances can migrate to the new process version, on the other hand. To this end, in our previous preliminary study [22], we took into account both control-flow state and variable state in the instance state, and proposed a novel migration criterion based on the instance dependence graph of the process instance trace. This paper extends our previous work [22] and makes the following contributions:

1) We formulate the concept of migration validity by taking variable state into account. Based on this, we prove that it is generally an undecidable problem to check migration validity. We also discuss the sub-case of the problem that is decidable and specify the associated complexity.

2) We present a systematic solution of our approach, including a comprehensive description of our relaxed migration criterion, correctness proofs of both the traditional trace replaying and our approach, detailed algorithms of our approach to migration validity checking, and time complexity analysis of the proposed algorithms.

3) We realize our approach in a proof-of-concept tool Dilepis whose input consists of the old and the new data-aware processes specified in WS-BPEL [33] (i.e., the de facto standard for service-oriented executable processes), and traces in the XES format [34] (i.e., the de facto standard for event logs in the domain of process mining).

4) We evaluate the effectiveness and the efficiency of our approach by applying it and other approaches that are based on trace replaying to semi-synthetic data sets derived from real-world data-aware processes. We also present a case study to compare different approaches. The evaluation results demonstrate that our approach (i.e., the migration criterion) is more relaxed than existing ones and that it scales well in practice.

The remainder of the paper is organized as follows. Section 2 reviews related work. In Section 3, we first introduce some background material and provide a running example to motivate our work. In Section 4, we present our approach to migration validity checking of running process instances. In Section 5, we present the design of our implemented research prototype. In Section 6, we conduct experimental evaluation to investigate the effectiveness and the efficiency of our approach. In Section 7, we discuss the merit and complements of our approach. Section 8 concludes the paper and envisions future work.

2 RELATED WORK

In this section, we review related work on instance migration for dynamic process evolution. Existing approaches fall into two major categories. The first kind defines the migration criteria from the perspective of process structures before and after change, while the second kind defines the criteria based on execution traces (i.e., completed activity sequences) of running instances. Many existing approaches focus on control-flow state consistency to ensure migration validity, while variable state consistency is not fully taken into account.

Let us first review representative work of the first kind. In [15], Aalst and Basten provide transfer rules to map a state of the old process version to a consistent state of the new version. These rules can guarantee proper termination of the migrated instances from the control-flow perspective. The approach requires that the two process versions are in an inheritance relation. Liske et al. present in [29] a similar approach which relaxes the inheritance relation to the accordance relation, i.e., the new process version should preserve all compatible partners of the old version. In addition, Aalst proposes another approach without restricting the change types to the control-flow of processes [35]. The approach compares the old and new process versions to obtain so-called change regions. Once an instance is not in the change region, it is permitted to migrate.

Next, we turn to representative approaches of the second kind. In the project WIDE [14], Casati et al. present an instance migration criterion called trace replaying: If the trace of a running instance can be replayed in the new process version, it is allowed to migrate [14], [36]. It is worth mentioning that trace replaying is only proposed to guarantee instance migration validity from the control-flow perspective. In the project ADEPT [37], [32], [18], Rinderle-Ma and Reichert put forward a migration criterion based on the pruned traces of the running instances. A pruned trace is derived from the original trace by first deleting activities which are not contained in the new process, and then discarding the activity occurrences related to loop iterations other than the last one. In this approach, a running instance can migrate if its pruned trace can be replayed in the new process version. However, as the authors point out, their approach can only guarantee the correctness of the
migrated instances from the control-flow perspective [32]. When variable state is taken into account, the approach is still conservative and the migration criteria are not relaxed enough.

Ryu et al. [17] establish that a running instance can be migrated to the new process version if both backward compatibility and forward compatibility are met. Backward compatibility is just like the criterion of trace replaying. Forward compatibility requires that the new version should preserve traces of the old version. In our opinion, forward compatibility is not necessary in general evolution cases. Zeng et al. [30] propose an instance migration criterion which relaxes the traditional trace replaying criterion. Unfortunately, this criterion is only applicable to processes without loops.

To summarize, most existing approaches only consider control-flow state, and thus may produce false positives and false negatives when employed to check instance migration validity of data-aware processes. Our approach belongs to the second kind. Different from existing approaches, our approach takes into consideration both control-flow state and variable state for migration validity checking. Moreover, our migration criterion is more relaxed than existing approaches based on trace replaying [14], [32].

Notably, the above-reviewed literature and our work, all focus on instance migration within a single-controlled process (orchestration), where changes are limited to the orchestration itself. When changes extend beyond the scope of an orchestration, environment properties such as partner orchestrations (services) should be considered [38], [39], [40], [41], [42]. This situation is referred to as choreography evolution [41]. Existing work concentrates on the problems of change propagation, change negotiation, and orchestration co-evolution [41], [43], [42], [44]. For a more detailed review of work on instance migration and work on process change (including choreography change), we refer to surveys on the subject [16], [27].

3 BACKGROUND
In this section, we first introduce preliminaries necessary to understand our work; then we motivate our approach with a running example.

3.1 Preliminaries
In this paper, data-aware processes are used as an abstraction of executable business processes, e.g., WS-BPEL processes. We employ directed graphs to represent data-aware processes with the help of four structured activities, including XOR-split, XOR-join, AND-split, AND-join [45].

Definition 1 (Data-aware process). A data-aware process is a five-tuple \( P = (N, F, D, I, O) \):

- \( N \) is a union of a set of basic activities and a set of structured activities including XOR-split, XOR-join, AND-split, AND-join.
- \( F \) is a set of flow relations representing activity orders.
- \( D \) is a set of variables defined or used in \( P \).
- \( I, O : N \rightarrow 2^D \) are functions assigning input and output variables to each activity in \( N \).

It is assumed that a process has a unique start activity \( A_s \) and a unique exit activity \( A_e \). The operational semantics of data-aware processes is described in our previous work [46]. A data-aware process \( P \) can execute and each execution corresponds to a process instance. If executed activities of an instance are recorded in the order of timestamps, a trace of \( P \) is obtained. In the rest of the paper, data-aware processes and processes are used interchangeably.

In data-aware processes, different activities may be interrelated by dependences, e.g., control and data dependences. Control and data dependences are well-established concepts in the domain of programming language and software engineering [47]. To make this paper self-contained, we review these concepts, defined in the context of business processes.

Definition 2 (Post-dominance). In a data-aware process \( P \), an activity \( A_i \) is post-dominated by another activity \( A_j \) if each directed path from \( A_i \) to the exit activity \( A_e \) (excluding \( A_i \)) contains \( A_j \).

Definition 3 (Control dependence). In a data-aware process \( P \), an activity \( A_j \) is control-dependent on another activity \( A_i \) if and only if there exists a directed path \( \rho \) from \( A_i \) to \( A_j \) with any activity \( A_k \) in \( \rho \) (excluding \( A_i \) and \( A_j \)) post-dominated by \( A_j \) and \( A_i \) is not post-dominated by \( A_j \).

If an activity \( A_j \) is control-dependent on another activity \( A_i \), \( A_i \) is usually a conditional activity such as an XOR-split. Data dependences are classified into three categories: true dependence, anti-dependence, and output dependence (cf. Definition 4) [47], [48].

Definition 4 (Data dependence). In a path \( \rho \) of a data-aware process \( P \), an activity \( A_j \) is true-dependent (anti-dependent, or output-dependent) on another activity \( A_i \) iff

- There is a variable \( v \in I(A_j) \cap O(A_i) \) \((O(A_j) \cap I(A_i))\), or \( O(A_j) \cap O(A_i) \).
- In \( \rho \), there is no \( A_k \) between \( A_i \) and \( A_j \) such that \( v \in O(A_k) \).

Assume \( A_i \) precedes \( A_j \) in a loop of a process \( P \). \( A_i \) is true-dependent on \( A_j \) if the latter activity writes a process variable read by \( A_i \); Definition 4 covers this case, and the dependence can be analyzed (cf. Algorithm 1 in Section 4.3.1).

Apart from control dependence and data dependence, another type activity dependence exists in WS-BPEL processes, i.e., the async-invocation dependence [49], which is caused by the asynchronous communication mechanism of WS-BPEL processes. Informally, a \(<\text{receive}>\) activity \( A_j \) is async-invocation dependent on a preceding one-way \(<\text{invoke}>\) activity \( A_i \) if and only if \( A_j \) is responsible for receiving the “response” (i.e., the invocation result) of \( A_i \) [49].

Table 1 summarizes the notations used in this paper.

3.2 A Running Example
In this section, we illustrate a dynamic evolution scenario of a data-aware process (WS-BPEL process) to motivate our work. The UML activity diagram depicted in Fig. 1a shows a WS-BPEL process of a Chinese tour agency that provides overseas travels, where the input and output of each activity are also depicted. The process works as follows. After receiving a tour query from a customer, it invokes the
query search service to obtain the response for the query. Then, it sends the response to the customer and meanwhile records the query. If the customer does not acknowledge (confirm) the tour, the process will go back to the first stage to wait for another query from the customer. Otherwise, it first invokes the passport service to apply for a passport for the customer. Based on the passport details and the customer’s tour request, the process then invokes the visa service to apply for the visa needed by the customer. Next, it concurrently books the flight and hotel for the customer. After all these steps, it combines the results, which are finally sent back to the customer. In 2014, Indonesia passed a new law, stating that from Jan 1st, 2015, Chinese tourists can travel to Indonesia without a visa. In the meantime, the tour agency finds that it can record the query in parallel with the query search followed by the response to the customer. In addition, it also finds that most of the hotel services have QoS (i.e., cost) correlations with the flight services [50], [51]. For example, if a client selects Lion Air, she will enjoy a discount when booking certain hotels in Indonesia. For these reasons, the Chinese tour agency modifies its WS-BPEL process for clients who plan to visit Indonesia. The modified process is depicted in Fig. 1b.

We refer to the WS-BPEL processes in Fig. 1a and 1b as $P_1$ and $P_2$. Suppose that there is an instance $I$ of $P_1$. Its current state and trace (completed activity sequence) are $s$ and $\sigma = A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9$, respectively. Since the WS-BPEL process is long-running, it is preferable to utilize the mechanism of instance migration to cope with dynamic evolution. The central question here is how to justify whether the migration of $I$ from $P_1$ to $P_2$ is valid or not.

To make existing approaches applicable in this scenario,
an activity $A_i$ in a trace $\sigma$ can be replayed in a state $s'$ of a process (e.g., $P_2$) if $A_i$ can be executed in $s'$ and its input and output should be consistent with those in $P_2$. Under this improvement, let us investigate whether existing approaches are applicable to address this problem. If the migration criterion based on the traditional trace replaying is used [14], $I$ will fail to be migrated because $\sigma$ cannot be replayed in $P_2$. If the migration criterion based on the relaxed trace replaying [32] is used, $I$ will also fail to be migrated because the reduced trace $\sigma' = A_1A_2A_3A_4A_5A_6A_7A_8A_9$ of $\sigma$ still cannot be replayed in $P_2$. On the other hand, the projection of $I$'s state $s$ onto $P_2$ corresponds to a reachable state of $P_2$. Consequently, in fact, the migration of $I$ from $P_1$ to $P_2$ is valid. This suggests that existing migration criteria are still overly conservative. Below in Section 4.3, we present a more relaxed migration criterion.

4 Approach

In this section, we first reformulate the notion of migration validity and then present our approach to instance migration validity checking.

4.1 Migration Validity

Since most existing migration validity criteria merely focus on control-flow state, they are not directly applicable to support the dynamic evolution of data-aware processes. In the following, we first reformulate migration validity checking by considering both control-flow state and variable state. Then, we draw an important conclusion for migration validity checking of process instances.

A running instance is characterized by the process definition according to which the instance executes and its runtime state. Informally, the instance state is a characterization of all variables of the instance, more formally defined next.

Definition 5 (Instance state). A state $s$ of an instance of process $P$ is defined as a partial function which maps variables in $P$ to values in the respective ranges (codomain), i.e., $s : D \rightarrow R$, where $D$ and $R$ are the sets of variables including a virtual variable called $PC$ which points to the current control-flow state) of $P$ and their ranges, respectively.

The virtual variable $PC$ is explained as follows. It always points to the current control-flow state which specifies the next activity (activities) to be executed. That is, $s(PC) = \{A_i | A_i$ is enabled $\}$, where the condition “$A_i$ is enabled” denotes that $A_i$ can be executed next. If Petri nets [32] are employed to represent the control flow of processes, a marking $M$ of the Petri net denotes a control-flow state of the process. For simplicity, we can use $s(PC) = M$ to replace $s(PC) = \{A_i | A_i$ is enabled $\}$.

Typically, an instance begins in an initial state with initial values assigned to some variables (i.e., input) in $D$. Its state changes during the execution of the instance. A state $s$ is said to be a reachable state of a process if and only if there is an instance which can reach $s$ at some time in the future from its initial state. The reason why $s$ is defined as a partial function is that some variables in $D$ have not been assigned values in $s$, and certain variables will never be assigned values even when the instance completes, for example, if

the process involves alternative structures. Under control of the process engine, an instance can suspend its execution in state $s$ (at time $t_1$) and can resume execution from the suspended state $s$ at a later time $t_2$.

When the dynamic evolution of a process occurs, we need to substitute the new version $P_2 = (N_2, F_2, D_2, I_2, O_2)$ for the old version $P_1 = (N_1, F_1, D_1, I_1, O_1)$, while there is one or more instance of $P_1$ still running. It is assumed that both $P_1$ and $P_2$ are “sound” processes in the sense that they can execute properly and produce the expected results [28]. The principal problem here is whether or not an instance of $P_1$ can migrate from $P_1$ to $P_2$ such that it can go on with its execution according to $P_2$. To this end, we need to explicitly specify the expected execution behaviour of an instance after migration. In general, the behaviour of an instance which undergoes a change is not the same as the behaviour of an instance of either $P_1$ or $P_2$. The best we can hope for is that after the change, the subsequent behaviour of the migrated instance is like that of an instance of $P_2$.

We find that two different instances $I_1$ and $I_2$ of $P_2$ can reach the same state $s$ from the same initial state but with different execution histories (i.e., traces). We note that their subsequent executions from $s$ may be identical, that is, we cannot differentiate them according to their subsequent behaviour. This indicates that if a suspended instance $I_3$ of $P_1$ resumes its execution from state $s$ following $P_2$, we can neither differentiate $I_3$ from $I_1$ nor from $I_2$ based on their subsequent executions. Moreover, $I_3$ will not get stuck, because $I_1$ and $I_2$ never get stuck either, which is ensured by the soundness of $P_2$ [28]. Bearing this in mind, we define instance migration validity as follows.

Definition 6 (Migration validity). A migration of a running instance $I$ from the old process version $P_1 = (N_1, F_1, D_1, I_1, O_1)$ to a new version $P_2 = (N_2, F_2, D_2, I_2, O_2)$ is valid if there is a partial function $S: s \mapsto s'$ which maps $I$’s current state $s$ in $P_1$ to a reachable state $s'$ in $P_2$. The state $s'$ is called the target state for the migration.

There is no restriction on the changes from $P_1$ to $P_2$. The changes can involve activity/variable deletion, activity/variable addition, activity/variable substitution, activity reordering, variable type modification, activities’ input/output changes, etc. Notably, if an activity of $P_1$ is preserved in $P_2$, its input and output should remain the same (at least type-compatible); otherwise, the updated activity is regarded as a new activity in $P_2$, even if its name remains the same.

Theoretically, the partial function $S$ for the state mapping can be arbitrary. To simplify the problem, $S$ is defined as follows. For any non-virtual variable $var \in D_1$ that has been assigned a value in $s$, if $var \in D_2$, $S(s)(var) = s(var)$ (i.e., $S(var, *) = (var, *)$, where $*$ is the value of $var$ in $P_1$); otherwise, $S(var, *)$ is undefined, because there is no variable in $V(P_2)$ that corresponds to $var$. For the virtual variable $PC$ in $P_1$, its value cannot be directly copied to $P_2$, because the activities which $PC$ points to in $P_1$ could be removed from $P_2$ or moved to another place in $P_2$, or new activities could be inserted before them in $P_2$. Thus, the control-flow state mapping needs to be defined additionally. The state mapping between non-virtual variables may need the help of programmers. For example, if a variable $a$ in $P_1$
is renamed to \( b \) in \( P_2 \), only the programmers who maintain the process are aware of this substitution. To define state mapping without the help of programmers, we assume that when a process evolves from \( P_1 \) to \( P_2 \), activity/variable renaming without substantial change is not considered. Unless otherwise stated, the partial function \( S \) for state mapping defined above is used in the remainder of this paper.

**Example 1** (Continuation of the running example from Section 3.2). For the instance \( I \) whose trace and current state are \( s = A_1A_2A_3A_4A_5A_6A_1A_2A_3A_5A_6A_7A_8A_{10}A_9 \) and \( s' \), respectively, we check whether the migration of \( I \) from \( P_1 \) to \( P_2 \) is valid. If control-flow state is mapped in the same way as non-virtual variables, a state mapping \( S \) is obtained such that \( S(s) = s' = \{PC = \{A_{11}\}, tourRequest = *, tourResponse = **, tourAcknow = ***, passportInfo = ****, flightInfo = ********, hotelInfo = *********\}, \) where * to ******* represent the values of corresponding variables. Since \( s' \) is reachable state of an instance (say, \( I' \)) whose trace is \( A_1A_2A_4A_3A_5A_6A_9A_{10}(o) \) of \( P_2 \), the migration of \( I \) from \( P_1 \) to \( P_2 \) is valid and \( s' \) is the target state.

Although Example 1 is straightforward, in the general case, unfortunately, the problem of migration validity checking is undecidable. To this end, we first show that determining whether a given state \( s \) is reachable in a process can be reduced to the halting problem [53]. The halting problem is to determine whether a given program will eventually halt on a given input, which is well-known to be undecidable.

**Lemma 1.** Given an arbitrary data-aware process \( P \), with initial state \( s_0 \), determined by the given input, it is undecidable to check whether or not a given state \( s \) is reachable from \( s_0 \).

**Proof.** Let us construct a process \( P' \). It first executes an arbitrary process \( P'' \). Then, if \( P'' \) terminates in a state \( s \), \( P' \) copies the current variable values in \( s \) to the corresponding variables of \( P \). As a consequence, \( P'' \) would halt when executed from the initial state \( s_0 \) if and only if state \( s \) is reachable in \( P \). However, it is undecidable whether a program (e.g., a data-aware process) ever halts when executed from its initial state. Hence, it is also undecidable to determine whether a given state can be reached in a process.

Since data-aware processes (e.g., WS-BPEL processes) are provided with all features of the programming model used in [54], Lemma 1 can be seen as an extension of the result presented in [54] which states that the reachability of the final state of a process is in general undecidable. With Lemma 1, we can prove the undecidability of checking the migration validity of a process instance.

**Theorem 1.** Given two process versions \( P_1 \) and \( P_2 \), a partial function \( S \) for state mapping, and an instance \( I \) of \( P_1 \) whose current state is \( s \), it is undecidable whether the migration of \( I \) from \( s \) to \( P_1 \) via the state mapping \( S \) is valid.

**Proof.** To check the migration validity of \( I \), according to Definition 6, we need to determine whether \( S(s) \) can be reached from the initial state of \( P_2 \) (we assume that \( P_1 \) and \( P_2 \) share the same input and thus the same initial state). Lemma 1 ensures the undecidability of this problem. Hence, Theorem 1 holds.

Theorem 1 implies that we cannot find a sufficient and necessary condition for determining migration validity. The best we can do is to find a more relaxed sufficient condition to address the problem. Generally speaking, since the undecidability of the problem is caused by a possibly infinite number of reachable states of the new process \( P_2 \), in theory, the problem becomes decidable when \( P_2 \) involves a bounded number of states reachable from its initial state \( s_0 \). However, even in that case, the number of reachable states in \( P_2 \) grows exponentially with the number of activities concurrently executing, and thereby it is still intractable to determine whether the migration of an instance is valid.

### 4.2 Trace Replaying Revisited

The approach of trace replaying is commonly used for migration validity checking from the control-flow perspective. Given the trace \( \sigma \) of a running instance \( I \) of the old process version \( P_1 \), if \( \sigma \) can be replayed in the new process version \( P_2 \), i.e., \( \sigma \) is also a trace of \( P_2 \), then \( I \) is allowed to migrate from \( P_1 \) to \( P_2 \). Otherwise, it is not permitted to migrate. In this paper, we improve trace replaying by requiring that, in addition to the replayed trace \( \sigma \), for each activity \( A_i \in \sigma \), the input and output variables of \( A_i \) in \( P_1 \) should be consistent with those in \( P_2 \). With this improvement, we show that, trace replaying also ensures migration validity (cf. Definition 6) from the data perspective.

**Definition 7 (Reaching definitions).** Reaching definitions at a given activity \( A_i \) in a trace \( \sigma \) are a set of variables (defined by earlier activities in \( \sigma \)) which can reach \( A_i \) without intervening redefinitions.

Given a trace \( \sigma \) of a process \( P \), we use \( \text{ReachingDefs}(P, \sigma) \) to represent the reaching definitions (i.e., variables that have been assigned values) at the end of \( \sigma \).

**Theorem 2.** Given two process versions \( P_1 = (N_1, F_1, D_1, I_1, O_1) \), \( P_2 = (N_2, F_2, D_2, I_2, O_2) \), and an instance \( I \) of \( P_1 \) whose current state and completed trace are \( s \) and \( \sigma \), respectively, if \( \sigma \) can be replayed in \( P_2 \), the migration of \( I \) from \( P_1 \) to \( P_2 \) is valid.

**Proof.** Suppose that \( s' \) is the state obtained after all activities in \( \sigma \) have been replayed in \( P_2 \), starting from its initial state. \( \text{ReachingDefs}(P_1, \sigma) \) and \( \text{ReachingDefs}(P_2, \sigma) \) summarize variables in \( D_1 \) and \( D_2 \) that have been assigned values in state \( s \) and \( s' \), respectively. Since \( \sigma \) is a trace of both \( P_1 \) and \( P_2 \), it follows that \( \text{ReachingDefs}(P_2, \sigma) = \text{ReachingDefs}(P_1, \sigma) \cap \text{ReachingDefs}(P_1, \sigma), s(v) = s'(v). \) Therefore, (I) for each variable \( v \in \text{ReachingDefs}(P_1, \sigma) \cap \text{ReachingDefs}(P_2, \sigma) \), we have \( (S(s))(v) = s(v) \) (I); (II) for each state \( w \in (D_1 \cap D_2) \setminus \text{ReachingDefs}(P_2, \sigma) \), \( s(u) \) and \( s'(u) \) are both undefined, thus \( (S(s))(u) \) as well; (III) for each variable \( w \in D_2 \setminus D_1 \), \( s'(w) \) is undefined since no activity of \( P_2 \) in \( \sigma \) can assign value to \( w \). (IV) Let \( S(PC, M_1) = (PC, M_2) \), where \( M_1 \) and \( M_2 \) are control-flow states obtained after \( \sigma \) is replaced in \( P_1 \) and \( P_2 \), respectively. Altogether, \( S(s) = s' \). According to Definition 6, the migration of \( I \) from \( P_1 \) to \( P_2 \) is valid and \( s' \) is the target state in \( P_2 \) for the migration.

**Example 2** (Continuation of the running example from Section 3.2). Assume that there is an instance \( I_1 \) of \( P_1 \). Its trace...
is \( \sigma_1 = A_1A_2A_4A_3A_5A_6A_7 \) and its current state and control-flow state are \( s \) and \( M_1 = \{ A_8 \} \), respectively. Since \( \sigma_1 \) can be replayed in \( P_2 \), the migration of \( I_1 \) from \( P_1 \) to \( P_2 \) is valid. Let \( S(\text{PC}, M_1) = (\text{PC}, M_2) \), where \( M_2 = \{ A_9 \} \) is the control-flow state reached after \( \sigma_1 \) is replayed in \( P_2 \). The state mapping of non-virtual variables are defined as before. \( S(s) \) is the target state for \( I_1 \)’s migration. Therefore, after migration, \( I_1 \) can resume execution from activity \( A_9 \).

### 4.3 Migration Validity Checking

The approach of trace replaying is conservative because it is too rigid in requiring that activity occurrences in a trace \( \sigma \) to also complete in the same order in the new version \( P_2 \). Our observation is that there is no need to require activity occurrences with no direct or indirect dependences to follow the order set by \( \sigma \). Instead, they can be replayed concurrently, i.e., their replay can be in any order. From this, we can obtain a more relaxed migration criterion (cf. Theorem 3), and our approach to migration validity checking involves the following three steps (cf. Fig. 2):

1) **Constructing the instance dependence graph.** We capture the activity dependences of \( \sigma \) into an instance dependence graph (IDG). We also summarize the reaching definitions at the end of \( \sigma \) into a set \( \text{ReachingDefs}(P_1, \sigma) \), where \( P_1 \) represents the old process version.

2) **Deriving the pruned instance dependence graph.** For every variable \( \text{var} \in \text{ReachingDefs}(P_1, \sigma) \uparrow D_2 \), we exploit a backward slicing technique to obtain the activity occurrences (trace slice) which have direct or indirect impact on \( \text{var} \). If an activity occurrence in \( \sigma \) is not contained in any such trace slice, this activity occurrence and the edges associated with it are removed from the IDG, thus obtaining a pruned instance dependence graph (PIDG).

3) **Checking migration validity.** We show that if there exists a topological sort of the activity occurrences in the PIDG such that it can be replayed in the new process version \( P_2 \), the instance migration is valid.

#### 4.3.1 Constructing Instance Dependence Graph

Before elaborating on Step 1 of our approach, we first present the definition of the IDG.

**Definition 8 (Instance dependence graph, IDG).** An instance dependence graph of a trace \( \sigma \) is a directed graph \( \text{IDG} = \langle N, E \rangle \), where

- \( N \) is a node set which represents activity occurrences in \( \sigma \).

Due to loops, some activities may occur more than once in trace \( \sigma \). According to the definition of the IDG, different occurrences of the same activity are represented by different nodes instead of the same node. Therefore, a node (activity occurrence) in the IDG cannot be dependent on an activity occurrence after it in \( \sigma \). That is, the IDG is a directed acyclic graph (DAG) by nature. If not stated otherwise, activities and activity occurrences are used interchangeably in the rest of the paper. Note that data dependences can be directly obtained from the trace. However, if structured activities, like an XOR-join, are not recorded in the trace, it is difficult to determine the control dependences. Hence, control dependences in a trace are determined from the original process. When the IDG of a trace \( \sigma \) is constructed, the reaching definitions at the end of \( \sigma \) can be obtained as well.

Let \( P_1 \) be a data-aware process and \( S_{\text{CD}} \) be the set of control dependences between activities in \( P_1 \). If \( \sigma \) is a trace of an instance of \( P_1 \), then Algorithm 1 is provided to construct the instance dependence graph \( \text{IDG}(N, E) \) of \( \sigma \) and to obtain the dynamic reaching definitions \( \text{ReachingDefs}(P_1, \sigma) \) at the end of \( \sigma \). More specifically, Line 5 inserts a new node into the IDG for each activity in \( \sigma \). For the new node, Lines 6-8 add edges of control dependence into the IDG; Lines 9-11 add data dependence edges with the current node as the target into the IDG; Lines 12-17 update the dynamic reaching definitions. When the outer “for” loop terminates, we obtain the IDG of \( \sigma \). The dynamic reaching definitions at the end of \( \sigma \), i.e., \( \text{ReachingDefs}(P_1, \sigma) \), is also stored in the output \( \text{Defs} \).

#### Algorithm 1 Construct the instance dependence graph

**Input:** the trace \( \sigma \) of a running instance of the old process version \( P_1 \); the set of control dependences \( S_{\text{CD}} \) in \( P_1 \)

**Output:** the instance dependence graph \( \text{IDG}(N, E) \) of \( \sigma \); the reaching definitions \( \text{ReachingDefs}(P_1, \sigma) \) at the end of \( \sigma \)

1: \( \text{IDG}(N, E) \leftarrow \emptyset \)
2: \( \text{Defs} \leftarrow \emptyset \)
3: \( \text{Uses} \leftarrow \emptyset \)
4: for \( j \leftarrow 1 \) to \( |\sigma| \) do
5: \( N \leftarrow N \cup \{ \text{var}[j] \} \)
6: for each \( \text{var}[i] \in N \) do
7: if \( \langle \text{var}[i], \text{var}[j] \rangle \in S_{\text{CD}} \) then
8: \( E \leftarrow E \cup \{ \text{var}[i], \text{var}[j] \} \)
9: for each \( \text{var} \in \text{inPut}(\text{var}[j]) \) do
10: \( E \leftarrow E \cup \{ \text{var}, \text{var}[j] \} \) if \( \text{var} \in \text{Defs} \)
11: \( \text{Uses} \leftarrow \text{Uses} \cup \{ \text{var} \} \)
12: \( \text{Defs} \leftarrow \text{Defs} \setminus \{ \text{var}[j] \} \)
13: \( \text{Kill}(\text{var}[j]) \leftarrow \emptyset \)
14: for each \( \text{var} \in \text{outPut}(\text{var}[j]) \) do
15: \( \text{Def}(\text{var}[j]) \leftarrow \text{Def}(\text{var}[j]) \cup \{ \text{var} \} \)
16: \( \text{Kill}(\text{var}[j]) \leftarrow \text{Kill}(\text{var}[j]) \cup \{ \text{var} \} \) if \( \text{var} \in \text{Defs} \)
17: \( \text{Defs} \leftarrow (\text{Defs} \setminus \text{Kill}(\text{var}[j])) \cup \text{Def}(\text{var}[j]) \)
18: return \( (\text{IDG}(N, E), \text{Defs}) \)
is an asyn-invocation dependence from \( A_3 \) to \( A_5 \) [49]; there are control dependences from the first occurrence of \( A_6 \) to the second occurrences of \( A_1-A_5 \), because the first occurrence of \( A_6 \) determines the execution of \( A_1-A_5 \) (the second occurrences); other edges are all data dependences. It can be seen that the IDG is a DAG, though the process \( P_1 \) involves a loop (cf. Fig. 1). The reaching definitions at the end of \( \sigma \) is \( \text{ReachingDefs}(P_1, \sigma) = \{ \text{tourRequest}_{A_1}, \text{tourResponse}_{A_2}, \text{tourAcknow}_{A_3}, \text{passportInfo}_{A_4}, \text{visaInfo}_{A_5}, \text{flightInfo}_{A_6} \} \), where \( \text{var}_{A_i} \) represents that the variable \( \text{var} \) is defined by activity \( A_i \).

### 4.3.2 Deriving Pruned Instance Dependence Graph

The aim of Step 2 of our approach is to remove from the IDG those activities that do not affect the state mapping of the instance. To this end, we define a trace slice as follows.

**Definition 9 (Trace slice).** A trace slice of a trace \( \sigma \) with respect to a variable (i.e., reaching definitions) \( \text{var} \) at a given activity \( A_i \) in \( \sigma \) is a set of activities that have direct and indirect impact on the value of \( \text{var} \), which is denoted as \( TS(\sigma, \text{var}) \).

The trace slice is acquired from the IDG of the trace. We obtain the trace slice with respect to a variable \( \text{var} \in \text{ReachingDefs}(P_1, \sigma) \uparrow D_2 \) by first finding the activity \( A_i \) that lastly defines \( \text{var} \) in \( \sigma \) and, then, using a backward slicing approach, to find all activities in the IDG that are reachable to \( A_i \), i.e., the activities on which \( A_i \) is directly and transitively dependent.

We note that only trace slices with respect to variables in \( \text{ReachingDefs}(P_1, \sigma) \uparrow D_2 \) are relevant to the state mapping. If an activity \( A_i \) in \( \sigma \) is not involved in any such trace slices, we draw the conclusion that \( A_i \) exerts no impact on the variables in \( D_2 \), and, thus, there is no need to replay \( A_i \) in the new process version \( P_2 \). Once we remove all such activities and their edges from the instance dependence graph, a pruned instance dependence graph (PIDG) can be obtained, which is also a DAG.

Algorithm 2 is presented to derive the PIDG from the corresponding IDG. Apart from the output of Algorithm 1, the input of Algorithm 2 also includes the variable set \( D_2 \) of the new process version \( P_2 \). Algorithm 2 proceeds as follows. For each variable \( \text{var}_{A_i} \) in \( \text{ReachingDefs}(P_1, \sigma) \uparrow D_2 \), Line 4 determines the activity \( \sigma[i] \) which defines the variable \( \text{var} \). The loop (Lines 3-13) identifies the trace slice which has impact on the value of \( \text{var} \) at \( \sigma[i] \). Line 4 and Line 12 add activities in the trace slice into the node set \( N' \) of the PIDG; Line 10 inserts the dependence edges between activities in the trace slice into the edge set \( E' \) of the PIDG. After the trace slices with respect to all variables in \( \text{ReachingDefs}(P_1, \sigma) \uparrow D_2 \) are found, the PIDG \( \text{PIDG}(N', E') \) is obtained.

**Example 4 (Continuation of Example 3).** According to Algorithm 2, we obtain the trace slices for the variables in \( \text{ReachingDefs}(P_1, \sigma) \uparrow D_2 = \{ \text{tourRequest}_{A_1}, \text{tourResponse}_{A_2}, \text{tourAcknow}_{A_3}, \text{passportInfo}_{A_4}, \text{visaInfo}_{A_5}, \text{flightInfo}_{A_6} \} \). Since activity \( A_6 \) is not involved in these obtained trace slices, \( A_6 \) and its edges are not contained in the final PIDG depicted in Fig. 4.

### 4.3.3 Checking Migration Validity

Instead of trying to replay the original trace \( \sigma \) in the new process version \( P_2 \), our goal is to replay activities in the PIDG of \( \sigma \). To guarantee the consistent state, during replaying, the replaying order of the two activities in the PIDG can be arbitrary if there is no direct or indirect dependences between them. Otherwise, their replaying order should be consistent with the dependence between them. This observation inspires us to define a more relaxed sufficient condition for migration validity checking.

**Theorem 3.** Given two process versions \( P_1 = (N_1, F_1, D_1, I_1, O_1) \), \( P_2 = (N_2, F_2, D_2, I_2, O_2) \), and an instance \( I \) of \( P_1 \), whose current state and completed trace are \( s \) and \( \sigma \) respectively, if there exists a topological ordering \( \rho \) of the activities in the PIDG of \( \sigma \) such that \( \rho \) can be replayed in \( P_2 \), then the migration of \( I \) from \( P_1 \) to \( P_2 \) is valid.
Proof. IDG construction shows that the IDG (also the PIDG) is a DAG. Assume that there is a topological ordering \( \rho \) of the activities in the PIDG such that activities in \( \rho \) can be successively completed in \( P_2 \), and \( s' \) is the state of \( P_2 \) after \( \rho \) is replayed. \( \text{ReachingDefs}(P_2, \rho) \) and \( D_2 \setminus \text{ReachingDefs}(P_2, \rho) \) denote the variables in \( V_2 \) that have been assigned values and have not been assigned values in state \( s' \), respectively. Since \( \rho \) is a topological ordering of the activities in the PIDG of \( \sigma \), it follows that \( \text{ReachingDefs}(P_2, \rho) = \text{ReachingDefs}(P_1, \sigma) \uparrow D_2 \). According to compiler theory for parallel execution of program statements, [47], we conclude that for any variable \( v \in \text{ReachingDefs}(P_2, \rho) \), \( s(v) = s'(v) \). Therefore, (I) for each variable \( v \in \text{ReachingDefs}(P_2, \rho) \), we have \( (S(s))(v) = s(v) = s'(v) \); (II) for each variable \( u \in (D_1 \cap D_2) \setminus \text{ReachingDefs}(P_2, \rho) \), \( s(u) \) and \( s'(u) \) are both undefined, thus \( (S(s))(u) \) is as well; (III) for each variable \( w \in D_2 \setminus D_1 \), \( s'(w) \) is undefined since no activity of \( P_2 \) in \( \sigma' \) can assign value to \( w \). (IV) Let \( S(P, M_1) = (P, M_2) \), where \( M_1 \) and \( M_2 \) are control-flow states obtained after \( \sigma \) and \( \rho \) are replayed in \( P_1 \) and \( P_2 \), respectively. Altogether, \( S(s) = s' \). According to Definition 6, the migration of \( I \) is valid and \( s' \) is the target state in \( P_2 \) for the migration.

According to Theorem 3, the migration validity of an instance is checked as follows. First, only activities in the PIDG with no incoming edges can be selected to determine whether they can be replayed in the current state of \( P_2 \). After one such activity \( A_i \) is replayed, the current state of \( P_2 \) is updated; \( A_i \) and its edges are deleted from the PIDG. The above step is repeated until no activity satisfies this condition. When the procedure terminates, if the PIDG becomes empty, the instance’s migration is valid. Otherwise, the procedure is terminated in advance, which indicates that the migration of the instance is not allowed by our approach.

Algorithm 3 is presented to check migration validity of an instance \( I \) (with trace \( \sigma \)) from the old process version \( P_1 \) to a new version \( P_2 \). The input of Algorithm 3 involves \( P_2 \) and the PIDG of the trace \( \sigma \). Its output includes the migration validity result and the corresponding target state.

Algorithm 3 works as follows. First, Lines 4-6 compute the indegree of each node (activity) and Lines 7-8 insert the nodes whose indegrees are zero into a set \( \text{zeroInNodeSet} \). Line 10 checks whether there is an activity in \( \text{zeroInNodeSet} \) that can be executed in the current state \( (P_2, M_c) \) of \( P_2 \). The notation \( (P_2, M_c) \) denotes that \( A_i \) can execute in the state \( (P_2, M_c) \) and its execution results in a new state \( (P_2, M_c') \). If there is no such activity, migration validity is not satisfied and the algorithm terminates (cf. Line 18). Otherwise, Line 11 removes \( n_i \) from \( \text{zeroInNodeSet} \); Line 12 updates the current state of \( P_2 \); Lines 13-14 decrease the indegrees of the nodes adjacent to \( n_i \). A node is added to \( \text{zeroInNodeSet} \) once its indegree falls to zero (cf. Lines 15-16). The above procedure (i.e., Lines 9-18) iterates until \( \text{zeroInNodeSet} \) becomes empty. If there is a topological ordering of the nodes (activities) in the PIDG that can be replayed in \( P_2 \), migration validity is satisfied and the target (control-flow) state for the migration is also obtained (cf. Line 19).

Example 5 (Continuation of Example 4). From the PIDG in Fig. 4, it follows that the indegree of the first occurrence of \( A_1 \) is zero. According to Algorithm 3, the first occurrence of \( A_1 \) is selected to be replayed in the new process version \( P_2 \) which then goes from the initial state to a new state. At the same time, the first occurrence of \( A_1 \) and its outgoing edges are removed from the PIDG. Then, since only the indegree of the first occurrence of \( A_2 \) becomes zero, the first occurrence of \( A_2 \) is selected to be replayed in \( P_2 \). The above steps iterated until the PIDG becomes empty, i.e., all activities in the PIDG have been replayed in \( P_2 \) (A1-A6 are replayed twice because two occurrences of them are in the PIDG). The emptiness of the PIDG indicates that the migration of \( I \) from \( P_1 \) to \( P_2 \) is valid. The current control-flow state \( (A_{11} \text{ is ready to execute}) \) of \( P_2 \) is also obtained for state mapping (cf. Fig. 5).
4.4 Complexity Analysis

Finally, we analyze the time complexity of the algorithms introduced in Section 4.3.

Algorithm 1 is proposed to construct the IDG of $\sigma$, and to obtain the reaching definitions at the end of $\sigma$. The time cost of building the IDG is $O(|N|+|E|)$, where $|N|$ and $|E|$ represent the number of activities and activity dependences in $\sigma$, respectively. During each step of the IDG construction, the dynamic reaching definitions in the set $Defs$ needs updating and each updating has to traverse the set $Defs$. Assume $k$ is the maximum number of output variables for all activities in $\sigma$. It can be shown that the time cost of obtaining $ReachingDefs(P_j, \sigma)$ is $O(k^2 \times |N|^2)$. Altogether, the time complexity of Algorithm 1 is $O(K^2 \times |N|^2 + |N| + |E|)$. 

Algorithm 2 is used to derive the PIDG from the IDG. Assume that the maximum number of output variables for all activities in $\sigma$ is $k$, and thus the number of variables in $ReachingDefs(P_j, \sigma)$ is at most $k \times |N|$, where $N$ is the node set of the input IDG. If the number of variables in $P_2$ is $|D_2| = m$, the time cost of Line 2 in Algorithm 2 is $O(m \times k \times |N|)$. Since Lines 3-13 are responsible for constructing the PIDG, the running time cost is $O(|N| + |E|)$ but in the worst case it is $O(|N| + |E|)$. Thus, the time complexity of Algorithm 2 is $O(m \times k \times |N| + |E|)$.

The time complexity of Algorithm 3 is analyzed as follows. If the migration validity is satisfied, Algorithm 3 determines a topological ordering of activities in $PIDG(N', E')$ such that it can be replayed in the new process version $P_2$. In this case, the running time cost is $O(|N'| + |E'|)$. Otherwise, the replaying procedure terminates in advance, so the time cost is less than $O(|N'| + |E'|)$. To summarize, the time complexity of Algorithm 3 is $O(|N'| + |E'|)$.

5 IMPLEMENTATION

To validate our approach separately, we implemented it in Java as a proof-of-concept standalone prototype called Dilepis. The functionality realized in Dilepis serves to determine the migration validity of process instances subject to dynamic evolution. Dilepis works with three input files: an original process version $P_1$, a new process version $P_2$, and a collection of traces of the running instances of $P_1$. Without loss of generality, the first two input files in our implementation are formatted in WS-BPEL [33], which is the de facto standard for data-aware business processes. The third input is imported as an XES file [34], which summarizes a set of traces of running instances. XES is the standard for business process event logs in the domain of process mining [34]. The output of Dilepis is the migration validity results of all traces (instances) in the input XES file.

Fig. 6 illustrates the architecture of Dilepis that consists of three layers: data layer, analysis layer, and presentation layer. The analysis layer contains three principal levels of components. The components at the bottom level comprise parsers of data-aware (WS-BPEL) processes and traces. Process Parser transforms WS-BPEL processes into our data-aware process models (cf. Definition 1). Trace Parser extracts traces (event sequences) from XES files.

In the extracted event sequence $\sigma$, each event $e = (A_i, in, out) \in \sigma$ corresponds to an activity, where $A_i$ is the activity name and $in/out$ refers to activities’ input/output variables. At the second level, Dependence Analyzer obtains the control dependences in the process and analyzes data dependences in the traces parsed from the XES file. Reaching Definition Analyzer obtains the reaching definitions at the end of each trace according to the data flow analysis shown in Algorithm 1. The top level includes three components: IDG Generator, PIDG Generator, and Migration Validator. These three components implement the three steps (i.e., Algorithms 1, 2, 3) of our approach, respectively.

We also use Dilepis to perform the experiment in Section 6. Dilepis and the data sets employed in the experiment are all publicly available at http://bit.ly/myDilepis.

Finally, we discuss how our approach Dilepis can be used by process engines. Our approach needs to be realized as a module of the process engine. The module is used by the engine to determine, when the process evolves from $P_1$ to $P_2$, whether a running instance $I$ of $P_1$ in its current state can be migrated to $P_2$ to continue execution. If so, the process engine needs to copy the state $s$ of $I$ from $P_1$ to $P_2$, according to the state mapping $S$ provided by our approach. Then, the process engine orchestrates $I$ to continue execution from the state $S(s)$ in $P_2$. Otherwise, the process engine resumes execution of $I$ according to $P_1$.

It is worth mentioning that although the execution of some activities in the process is not instantaneous, these activities are atomic operations. That is, the effect (output) of an activity can only be stored in the instance state after the activity completes the execution [37], [32], [18], and thus there is no middle state during its execution. Since process engines manage the state of the instance, as well as the lifecycle of each activity, migration checking can be conducted when all active activities of the process instance complete.
6 Evaluation

In this section, we conduct an extensive experimental followed by an intensive case study to evaluate our approach, and compare it with state-of-the-art approaches that are based on trace replaying (cf. Table 2). Note that our evaluation only focuses on the migration criteria and the corresponding checking methods instead of comparing whether the approaches can allow the instances that do not satisfy migration validity to be migrated with the help of extra measures. Although the ADEPT framework provides more advanced strategies to improve the number of instances that can be migrated [55], to allow for a well-scoped and fair comparison, these strategies are not considered in our evaluation because they are orthogonal to migration validity checking.

6.1 Experiment

Initially, we present our experimental evaluation by comparing our approach with approaches in Table 2. The experiment is performed on a PC with 2.4GHz Intel i5-4210U processor and 4GB memory running Windows 8 and Java SE 6.0. Through the experiment, we study the following two research questions:

1) **RQ1:** Effectiveness - Empirically, does our approach allow more instances to migrate than existing approaches do?

2) **RQ2:** Efficiency - Does our approach scale?

6.1.1 Experimental Setup

**Data set and experimental procedure.** Ideally, we need a set of data-aware processes before and after evolution as well as a set of to-be-migrated instances to conduct the experiment. Unfortunately, there is no widely-acknowledged data set suitable for our experiment. To fully explore the performance of different approaches, we employ a total of 41 real-world WS-BPEL processes to generate the required data set. These WS-BPEL processes were downloaded from the Reo project website\(^2\), divided into several categories including “FromBPELSpec”, “FromIBM”, “FromOracle”, and “FromQUT”.

For each WS-BPEL process, we first generate a collection of its traces (not complete). Then, we randomly modify a certain proportion of the overall traces to derive trace variants. We note that each variant is obtained by one modification only, that is, deleting one activity, adding one activity, and swapping the order of two activities. Other change scenarios can be transformed into the above three cases. The manner of introducing variants is inspired by fault seeding practiced in software testing [56], [57], [58], [59], which is discriminative to study the performance of different approaches in different process evolution scenarios. Our experiment is performed on four data sets. The first one contains trace variants that are obtained by deleting one activity from the corresponding compliant traces. The second one involves trace variants which are obtained by adding one activity to the corresponding compliant traces. The third one includes trace variants that are acquired by swapping the order of two activities in the corresponding compliant traces. The last one involves trace variants of the above three different kinds. Each of these four data sets consists of nine subsets. Each of the subsets contains 30 traces with a proportion (from 10 percent to 100 percent) of trace variants. As a consequence, 41 WS-BPEL processes and 41 x 4 x 10 x 30 = 49,200 traces in total are employed in our experiment. Suppose that the variants are traces of the instances of the old process version \(P_1\) (that do not really exist) and the corresponding WS-BPEL process is the new process version \(P_2\) after evolution. Under this assumption, we check whether the migrations of those instances to \(P_2\) are valid according to different migration criteria (i.e., approaches). After this, different approaches are compared based on the evaluation criteria introduced later. Notably, the reason why we modify traces to obtain trace variants instead of modifying processes to obtain process variants is that process changes occurring to the “future” of instances do not prevent instances from being migrated.

**Evaluation criteria.** Since the problem of instance migration validity checking is undecidable, the oracle (ground true) of whether a process instance satisfies migration validity is unavailable in our experiment. Consequently, we cannot compute the false positives and false negatives as usual. As Dilepis and the improved WIDE (when variable state is considered) correspond to two sufficient conditions of the problem, we employ the following two criteria (that is, *migration rate*, *migration factor* [32]) to evaluate and compare the effectiveness of the three different approaches. We denote a collection of instances (traces) of a process \(P\) as \(InstanceSet_P\), and the set of instances that can be migrated to the new process version based on the migration criterion (approach) \(MC \in \{\text{WIDE, ADEPT, Dilepis}\}\) as \(InstanceSet_{MC}\). The migration rate \(MR_{MC}\) is used to measure the proportion of migratable instances by using different approaches, and the migration factor \(MF_{MC_1 \rightarrow MC_2}\) is utilized to measure the increase of the migration rate when going from the migration criterion \(MC_1\) to the migration criterion \(MC_2\):

\[
MR_{MC} = \frac{|InstanceSet_{MC}|}{|InstanceSet_P|} \tag{1}
\]

\[
MF_{MC_1 \rightarrow MC_2} = \frac{|InstanceSet_{MC_2}| - |InstanceSet_{MC_1}|}{|InstanceSet_P|} \tag{2}
\]

Furthermore, since each of the three approaches compared in the evaluation also returns a “target state” in the new process \(P_2\) if an instance is allowed to migrate, we can manually validate whether this “target state” \(s’\) is consistent with the instance state \(s\) before the migration. If they are not consistent (can be checked by comparing variable state),

<table>
<thead>
<tr>
<th>Table 2 Migration Criteria (Approaches) Compared in the Experiment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ref.</td>
</tr>
<tr>
<td>WIDE</td>
</tr>
<tr>
<td>ADEPT</td>
</tr>
<tr>
<td>Dilepis</td>
</tr>
</tbody>
</table>

---

2. [http://reo.project.cwi.nl/reo/](http://reo.project.cwi.nl/reo/)
a false positive is generated. Through this manual check, we confirm our theoretical results that both WIDE (when variable state is considered) and our approach Dilepis do not produce false positives. However, the ADEPT approach may produce false positives, because when taking the variable state into account, certain “target states” returned by ADEPT are not consistent with the instance states in $P_1$, i.e., the values of some variables are different.

Also, the efficiency of an approach to migration validity checking is an important consideration. An inefficient approach could impact the quality of service properties of a process or even result in the failure of some real-time safety-critical systems. Hence, in the experiment, we also investigate the efficiency (i.e., runtime overheads) of different approaches to migration validity checking.

6.1.2 RQ1: Effectiveness

Experimental results of migration rates. Fig. 7 illustrates the average migration rates generated by the three approaches summarized in Table 2 on the four data sets.

From Fig. 7a, it follows that the migration rates of all three approaches decrease with the increasing ratio of trace variants in the first data set. We also find that, when the ratio of trace variants is fixed, the migration rate remains stable regardless of the migration criteria (approaches) used. In other words, the performance of the three approaches on the first data set is similar. By manually checking the consistency between the “target states” returned by different approaches and the instance state before migration, we find that neither approach creates false positives in this case. Therefore, compared with the WIDE approach, both approaches of ADEPT and Dilepis fail to increase the number of migratable instances, when process evolution only adds new activities to the instance’s “history” (i.e., the completed process fragment) in the new process version. In fact, adding activities to the “history” is the major reason leading to migration failure, because it is hard to find in the new version a reachable state that is consistent with the current state of the running instance in the old version. So far, to the best of our knowledge, there is no better migration criterion that could raise the migration rate in this type of process evolution.

From Fig. 7b, it can be seen that the migration rates of all three approaches decrease with the growing ratio of trace variants in the second data set. Nevertheless, both the ADEPT approach and our approach (Dilepis) perform better than the WIDE approach (i.e., the traditional trace replaying). Specifically, the first two approaches allow more instances to be migrated than the WIDE approach does. This indicates that process evolution that deletes activities from the instance’s “history” in the new process version can greatly affect the migration rate of WIDE. Fortunately,
the influence on the other two approaches is not significant, because some deleted activities do not affect the state mapping. In this type of process evolution, ADEPT and Dilepis are more relaxed than WIDE. However, ADEPT is not sound when variable state is taken into consideration, and thus it may produce false positives. By manually checking the “target states” returned by different approaches, we find that the proportion of false positives generated by ADEPT is 23.4 percent, whereas WIDE and Dilepis do not produce false positives. In the experiment, we discarded the false positives generated by ADEPT, and therefore the performance of ADEPT and Dilepis is comparable.

Fig. 7c reveals that the migration rate of our approach (Dilepis) is higher than that of the other two approaches. That is, Dilepis allows more instances to be migrated than the other two approaches do. By manually checking the “target states” returned, we find that neither approach generates false positives in this case. This result demonstrates that process evolution that swaps activities in the instance’s “history” can greatly affect the migration rate of both WIDE and ADEPT. On the other hand, the impact on Dilepis is not significant, because swapping activities without dependencies does not affect the state mapping in our approach. Thereby, our approach is more relaxed than the other two approaches for this type of process evolution.

Fig. 7d illustrates that our approach (Dilepis) allows more instances to be migrated than the other two approaches (i.e., WIDE, ADEPT) do. This result demonstrates that, overall, our approach is more relaxed than the other two approaches. In summary, ADEPT may produce false positives for instance migration validity checking, when certain activities are deleted during process evolution. In contrast, WIDE and Dilepis do not create false positives in any case, which confirms Theorems 2 and 3.

Experimental results of migration factors. Figs. 8a-8d reveal the average migration factors on the four data sets, respectively. These figures reflect a similar rule as Figs. 7a-7d do.

As Fig. 8a shows, with the increasing ratio of trace variants in the first data set (the process evolution scenario is inserting activities into the instance’s “history”), there is few increase in migratable instances when going from the migration criterion WIDE to the migration criterion ADEPT and Dilepis. From Fig. 8b, it follows that with the growing ratio of trace variants in the second data set (the process evolution scenario is deleting activities from the instance’s “history”), there is an increase in the proportion (from 6 percent to 55.1 percent) of migratable instances when going from the migration criterion WIDE to the migration criteria ADEPT and Dilepis. Fig. 8c demonstrates that with the rising ratio of trace variants in the third data set (the process evolution scenario is swapping activities in the instance’s “history”), there is a significant increase in the proportion

![Graphs showing migration factors on four data sets.](attachment:image.png)
(from 4.2 percent to 32.5 percent) of migratable instances when going from the migration criteria WIDE and ADEPT to the migration criterion Dilepis. Fig. 8d illustrates that with the increasing ratio of trace variants in the fourth data set (with different evolution scenarios), there is an increasing proportion (from 3.4 percent to 29.2 percent) of migratable instances when going from the migration criterion WIDE to the migration criterion Dilepis, and also there is an increasing proportion (from 1.4 percent to 10.8 percent) of migratable instances when going from the migration criterion ADEPT to the migration criterion Dilepis.

6.1.3 ROQ2: Efficiency

To evaluate the efficiency of our approach, we record runtime overheads of different approaches in our experiment. Figs. 9a, 9b, and 9c illustrate the average runtime costs of different approaches with the increase in process scale (i.e., the number of activities in a process), trace length (i.e., the number of activities in a trace), and trace quantity (i.e., the number of to-be-migrated instances), respectively. From Fig. 9, it can be seen that:

Overall, WIDE is the most efficient approach. However, when the process involves loops and the trace contains many iterations of activities in loops (this is the reason why the trace length shown in Fig. 9b can be greater than 50), ADEPT is the most efficient approach. This is because activity iterations except the last one are deleted from the trace by ADEPT before the trace is replayed. Compared with the other two approaches, our approach (Dilepis) is the most inefficient. This is unsurprising, because our approach needs more time to obtain the IDG (also PIDG) and the reaching definitions at the end of the trace. However, it is worth the effort, as our approach gives rise to more migratable instances than the other two approaches do.

Our approach and the other two approaches scale well with the increase of process scale, trace length, and trace quantity, respectively. The runtime overhead of our approach can be neglected for long-running, real-world business processes. Even for some real-time, safety-critical business processes, the overhead (less than 1-2 milliseconds in our experiment) of our approach would be acceptable.

6.2 Case Study

Here, we present a case study to demonstrate the effectiveness and advantages of our approach. The case study stems from two real-world WS-BPEL applications: Auction and MarketPlace3. Although these processes are named differently, they share similar functionalities; that is, they both act as intermediaries between sellers and buyers. Based on these applications and their variants, four versions of the marketplace service (process) are constructed. Note, similar services and applications are frequently used in the literature on service-based software engineering, e.g., [48], [57], [17], [21], [60], [39], [61], [58], [59], [46].

Fig. 10a shows the UML activity diagram to present the first version V1 of the marketplace process. It initially receives the requests from the seller and the buyer in parallel. If the price (buyerInfo.offer) offered by the buyer is not lower than the expected price (sellerInfo.askingPrice) of the seller, the trade is successful, and, according to the marketplace operation policies, the marketplace process invokes the trade registration service to register the transaction; otherwise, the trade fails. In the end, the trade outcome, regardless of success or failure, is conveyed to seller and buyer. Operational analysis conducted at the marketplace reveals, that a buyer often waits for a long time for a seller due to the imbalance between supply and demand. In order to reduce the waiting time of the buyer, the marketplace owner modifies the process to first receive seller requests and then buyer request. In addition, the process is further improved by notifying seller and buyer of the trade outcome in parallel. These two changes result in a new process version V2 (cf. Fig. 10b). Also, the marketplace process may be affected by changes in the environment. For example, the trade registration service updates its interface, which calls for a request-response <invoke> activity to interact with it. To adapt to this change, the two activities A5 and A6 are replaced by a new activity A10 in the third process version V3 shown in Fig. 10c. Finally, the marketplace process evolves to the process version V4 shown in Fig. 10d, when the operational policy to register trades is cancelled.

Table 3 lists a set of running process instances of V1, where the trace of each process instance and the reaching

definitions at the end of the trace are given. In the following, we investigate whether these process instances can be migrated to V2, V3, and V4 according to the different approaches. Other scenarios, such as instance migrations from V2 to V3 and V4, are also possible. Let us first analyze I1. Since its trace σ1 can be replayed in V2 rather than V3 and V4, WIDE only allows I1 to migrate to V2. Similarly, ADEPT allows I1 to migrate to V2. Note, although activity A10 and A5 share the same name, their signatures (input and output) are different, thus, they constitute different activities. In order to investigate whether I1 can migrate to V3 according to ADEPT, we first need to obtain its reduced trace σ1′ = A1A2A3A4A5. Since σ1′ cannot be replayed in V3, ADEPT does not allow I1 to migrate to V3. Since the reduced trace σ1′ = A1A2A3A4A5A8 with respect to V4 can be replayed in V4, ADEPT allows I1 to migrate to V4. When Dilepis is used, we first obtain the IDG (instance dependence graph) of I1. Then, we derive the PIDG (pruned instance dependence graph) with respect to the new version V2 (V3, V4). Fig. 11a presents the IDG of I1, which is also the PIDG of I1 with respect to V2 (V4). Since answer ∈ ReachingDefs ∩ V(P3), activities A5 and A6 are retained in the PIDG. Fig. 11b presents the PIDG of I1 with respect to V4. Finally, since there is a topological sort of the nodes in Fig. 11a (Fig. 11b), i.e., p1 = A1A2A3A4A5A8 (p′1 = A1A2A3A4A8), which can be replayed in V2 (V4), Dilepis
TABLE 3
A Set of Process Instances of the Marketplace Process $V_I$

<table>
<thead>
<tr>
<th>Instance</th>
<th>Trace</th>
<th>Reaching definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_1$</td>
<td>$\sigma_1 = A_1A_2A_3A_4A_5A_6A_8$</td>
<td>$\text{sellerInfo}<em>{A_1}, \text{buyerInfo}</em>{A_2}, \text{outcome}<em>{A_4}, \text{answer}</em>{A_6}$</td>
</tr>
<tr>
<td>$I_2$</td>
<td>$\sigma_2 = A_2A_1A_3A_4A_5A_8$</td>
<td>$\text{buyerInfo}<em>{A_2}, \text{sellerInfo}</em>{A_1}, \text{outcome}<em>{A_4}, \text{answer}</em>{A_6}$</td>
</tr>
<tr>
<td>$I_3$</td>
<td>$\sigma_3 = A_1A_2A_3A_4A_6$</td>
<td>$\text{sellerInfo}<em>{A_1}, \text{buyerInfo}</em>{A_2}, \text{outcome}<em>{A_4}, \text{answer}</em>{A_6}$</td>
</tr>
<tr>
<td>$I_4$</td>
<td>$\sigma_4 = A_2A_1A_3A_4A_5A_6$</td>
<td>$\text{buyerInfo}<em>{A_2}, \text{sellerInfo}</em>{A_1}, \text{outcome}<em>{A_4}, \text{answer}</em>{A_6}$</td>
</tr>
<tr>
<td>$I_5$</td>
<td>$\sigma_5 = A_2A_1A_3A_4A_8$</td>
<td>$\text{buyerInfo}<em>{A_2}, \text{sellerInfo}</em>{A_1}, \text{outcome}_{A_4}$</td>
</tr>
<tr>
<td>$I_6$</td>
<td>$\sigma_6 = A_1A_2A_3A_4A_8$</td>
<td>$\text{sellerInfo}<em>{A_1}, \text{buyerInfo}</em>{A_2}, \text{outcome}_{A_4}$</td>
</tr>
<tr>
<td>$I_7$</td>
<td>$\sigma_7 = A_2A_1A_3A_4A_5$</td>
<td>$\text{buyerInfo}<em>{A_2}, \text{sellerInfo}</em>{A_1}, \text{outcome}_{A_4}$</td>
</tr>
<tr>
<td>$I_8$</td>
<td>$\sigma_8 = A_1A_2A_3A_4A_5$</td>
<td>$\text{sellerInfo}<em>{A_1}, \text{buyerInfo}</em>{A_2}, \text{outcome}_{A_4}$</td>
</tr>
<tr>
<td>$I_9$</td>
<td>$\sigma_9 = A_1A_2A_3A_4A_6$</td>
<td>$\text{sellerInfo}<em>{A_1}, \text{buyerInfo}</em>{A_2}, \text{outcome}_{A_4}$</td>
</tr>
<tr>
<td>$I_{10}$</td>
<td>$\sigma_{10} = A_1A_2A_3A_4A_6$</td>
<td>$\text{sellerInfo}<em>{A_1}, \text{buyerInfo}</em>{A_2}, \text{outcome}_{A_4}$</td>
</tr>
<tr>
<td>$I_{11}$</td>
<td>$\sigma_{11} = A_1A_2A_3$</td>
<td>$\text{sellerInfo}<em>{A_1}, \text{buyerInfo}</em>{A_2}, \text{outcome}_{A_4}$</td>
</tr>
<tr>
<td>$I_{12}$</td>
<td>$\sigma_{12} = A_1A_2A_3$</td>
<td>$\text{buyerInfo}<em>{A_2}, \text{sellerInfo}</em>{A_1}, \text{outcome}_{A_4}$</td>
</tr>
<tr>
<td>$I_{13}$</td>
<td>$\sigma_{13} = A_1A_2A_3$</td>
<td>$\text{sellerInfo}<em>{A_1}, \text{buyerInfo}</em>{A_2}$</td>
</tr>
<tr>
<td>$I_{14}$</td>
<td>$\sigma_{14} = A_2A_1A_3$</td>
<td>$\text{buyerInfo}<em>{A_2}, \text{sellerInfo}</em>{A_1}$</td>
</tr>
<tr>
<td>$I_{15}$</td>
<td>$\sigma_{15} = A_1A_2$</td>
<td>$\text{sellerInfo}<em>{A_1}, \text{buyerInfo}</em>{A_2}$</td>
</tr>
<tr>
<td>$I_{16}$</td>
<td>$\sigma_{16} = A_2A_1$</td>
<td>$\text{buyerInfo}<em>{A_2}, \text{sellerInfo}</em>{A_1}$</td>
</tr>
<tr>
<td>$I_{17}$</td>
<td>$\sigma_{17} = A_2$</td>
<td>$\text{sellerInfo}_{A_1}$</td>
</tr>
<tr>
<td>$I_{18}$</td>
<td>$\sigma_{18} = A_2$</td>
<td>$\text{buyerInfo}_{A_2}$</td>
</tr>
</tbody>
</table>

![Diagram](image)

**Fig. 11. IDG and PIDG of $I_1$ of process version $V_I$: (a) IDG of $I_1$ and PIDG of $I_1$ with respect to process versions $V_2$ and $V_3$, (b) PIDG of $I_1$ with respect to process version $V_4$.**

allows $I_1$ to migrate to $V_2$ ($V_4$), but does not allow $I_1$ to migrate to $V_3$, either.

Since both $\sigma_2$ and its reduced trace with respect to $V_3$ and $V_4$ cannot be replayed in $V_2'$-$V_4'$, neither WIDE nor ADEPT allows $I_2$ to migrate to $V_2'$-$V_4'$. However, similar to $I_1$, $I_2$ is allowed to migrate to $V_2$ and $V_4$ according to Dilepis, because $I_2$ and $I_1$ share the same IDG and PIDG. Since the target state, including control-flow state $PC = M = \{A_0\}$ and variable state (in terms of symbolic reaching definitions, i.e., $\text{buyerInfo}_{A_2}, \text{sellerInfo}_{A_1}, \text{outcome}_{A_4}, \text{answer}_{A_6}$), of $I_2$ returned by Dilepis corresponds to a reachable state of $V_2$, it is not a false positive to migrate $I_2$ to $V_2$. Similarly, the migration result that $I_2$ can migrate to $V_4$ is also not a false positive. In contrast, WIDE and ADEPT generate false negatives on the migration of $I_2$ to $V_2$ and $V_4$.

Neither approach allows $I_2$ ($I_1$) to migrate to $V_3$. This is because, the main idea of these three approaches is to reuse the completed activities of the process instance in $V_1$, whereas the newly-added activity $A_{10}$ is absent from $\sigma_2$ ($\sigma_1$). On the other hand, the effect of $A_{10}$ is equivalent to that of $A_5$ and $A_6$, which is only known to the process maintainer or programmer. Therefore, the current state $s$ of $I_2$ ($I_1$) is consistent (same) with a reachable state of $V_3$. Therefore, $I_2$ ($I_1$) can be migrated to $V_3$, which implies that all the three approaches produce false negatives in this case. This example also illustrates the limitation of our approach. In fact, it is difficult to avoid such false negatives without the help of process maintainers.

Next, we analyze the migration validity of $I_3$. Since the original trace $\sigma_3 = A_1A_2A_3A_4A_5A_6$ of $I_3$ can be replayed in $V_2$ and can neither be replayed in $V_3$ nor in $V_4$, WIDE only allows $I_3$ to migrate to $V_2$. When ADEPT is used, we first obtain the reduced trace $\sigma'_3 = A_1A_2A_3A_4$ of $I_3$ with respect to $V_3$ ($V_4$). Since $\sigma'_3$ can be replayed in $V_3$ ($V_4$), $I_3$ is allowed to migrate to $V_2$, $V_3$, and $V_4$ according to ADEPT. However, after $I_3$ is migrated to $V_3$, $A_{10}$ is the first activity to execute, and therefore the trade between seller and buyer is registered twice. This is not allowed by the registration service according to the registration policy. The migration of $I_2$ to $V_3$ may even lead to an exception. In fact, although activities $A_5$ and $A_6$ are deleted, their effect, i.e., the value of the variable $\text{answer}_{A_6}$ is still in the state $I_3$. According to our definition of migration validity (cf. Definition 6), the target state of $I_3$ returned by ADEPT is not a reachable state of $V_3$. Altogether, allowing $I_2$ to migrate to $V_3$ is a false positive of ADEPT. Similar to the migration validity analysis of $I_1$ and $I_2$, Dilepis allows $I_3$ to migrate to $V_2$ and $V_4$ rather than $V_3$.

The migration results of all process instances of $V_1$ listed in Table 3 can be determined similarly, which are summarized in Table 4. Since the number of reachable states of $V_2$ ($V_3$, $V_4$) is enumerable, false positives and false negatives of different approaches can be determined on the basis of the manual checking procedure outlined above. ADEPT produces two false positives, whereas WIDE and Dilepis do not produce false positives (cf. Table 5), which confirms our theoretical results. The number of false negatives generated by WIDE, ADEPT, and Dilepis are $29$, $24$, $4$, respectively. This indicates that our approach avoids many false negatives.
produced by the other two approaches. Based on these results, the migration rate of each approach can be calculated (false positives are excluded), as well as the migration factor from one migration criterion (approach) to another (cf. Table 6). These results demonstrate that, compared with WIDE and ADEPT, Dilepis allows more process instances to migrate to the new process version, without giving rise to false positives. Thus, we conjecture that Dilepis is promising in practice.

6.3 Threats to Validity

We analyze both construct validity and external validity for our experiment and case study.

Construct validity. The process models defined by WIDE, ADEPT, and Dilepis are different. In the process model of WIDE, only activity conditions are defined, whereas input and output variables are not defined. In the process model of ADEPT, activities are linked to data (variables) with “read” and “write” edges. In our process model (cf. Definition 1), each activity is associated with input and output variables, which constitutes a simpler representation. Since neither WIDE nor ADEPT are publicly available, and both approaches do not take variable state into account, we implement and improved them according to their description in the literature based on our own data-aware process model. The re-implementation may introduce a threat to construct validity in our experimental results; mainly affecting runtime performance metrics which heavily depend on the implementation, i.e., the data structures used.

External validity. There are two main threats to the external validity of our empirical results. For one thing, we employ totally 41 WS-BPEL processes and their adapted trace variants for the migration experiment. For each WS-BPEL process, the trace variants are obtained through man-

TABLE 4

Migration Validity Results of Process Instances in Table 3 according to Different Approaches

<table>
<thead>
<tr>
<th>Instance</th>
<th>Migratable to V₂?</th>
<th>Migratable to V₃?</th>
<th>Migratable to V₄?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>WIDE</td>
<td>ADEPT</td>
<td>Dilepis</td>
</tr>
<tr>
<td>I₁</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>I₂</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>I₃</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>I₄</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>I₅</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>I₆</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>I₇</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>I₈</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>I₉</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>I₁₀</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>I₁₁</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>I₁₂</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>I₁₃</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>I₁₄</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>I₁₅</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>I₁₆</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>I₁₇</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>I₁₈</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

TABLE 5

The Number of False Positives and False Negatives of the Results in Table 4

<table>
<thead>
<tr>
<th></th>
<th>False positives</th>
<th>False negatives</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>WIDE</td>
<td>ADEPT</td>
</tr>
<tr>
<td>V₁ → V₂</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>V₁ → V₃</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>V₂ → V₄</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Overall</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

TABLE 6

Overall Results on the Migration Rate and Migration Factor

<table>
<thead>
<tr>
<th></th>
<th>Migration rate</th>
<th>Migration factor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>WIDE</td>
<td>ADEPT</td>
</tr>
<tr>
<td>V₁ → V₂</td>
<td>50%</td>
<td>50%</td>
</tr>
<tr>
<td>V₁ → V₃</td>
<td>33.3%</td>
<td>33.3%</td>
</tr>
<tr>
<td>V₁ → V₄</td>
<td>33.3%</td>
<td>50%</td>
</tr>
<tr>
<td>Overall</td>
<td>38.9%</td>
<td>44.4%</td>
</tr>
</tbody>
</table>
ual modification of the original compliant traces, and like fault seeding in WS-BPEL process testing [57], [58], [59], each trace variant is planted with only one modification (deleting one activity, adding one activity, swapping the order between two activities), which may fail to represent real-world cases (In fact, the experimental result of our approach is even better when different kinds of modifications are involved in a trace.). Moreover, since the processes before evolution are not actually available but they only exist theoretically, our experimental setup is not the same as real scenarios. For another, WS-BPEL processes are only one kind of data-aware processes. It needs further investigation whether the empirical results of our experiment and case study can be generalized to other data-aware processes. To minimize these two threats, we will try to evaluate different approaches with more real-world evolution scenarios of data-aware processes.

7 Discussion

In this section, we first introduce how session management of process instances is handled in our approach. Then, we discuss the strengths and limitations of our approach, as well as complementary techniques to further improve instance migration possibility.

Since data-aware processes are stateful, “session” management ought to be considered in migration validity checking. Let us explain how session is handled in our approach. For business processes, e.g., WS-BPEL processes, a session of a process instance can be characterized by the sequence of messages sent to or received from its “environment” (external services) [58]. Assume that $P_1$ and $P_2$ are the processes before and after the evolution, and $I$ is a process instance of $P_1$. The “effect” of the session (message sequence) of $I$ is reflected by the values of corresponding variables in the process. In particular, a WS-BPEL process instance, say, $I$, uses a $<\text{receive}>$ activity to receive a message from its environment, and the received information is delivered to the output of the $<\text{receive}>$ activity; $I$ uses a one-way $<\text{invoke}>$ (or $<\text{reply}>$) activity to send a message, and the information sent to its environment is encapsulated as the input of the $<\text{invoke}>$ ($<\text{reply}>$) activity. $I$ may also use the request-response $<\text{invoke}>$ activity, which encapsulates the message sent in its input, and extracts the replied message in its output. In addition, WS-BPEL provides correlation mechanism ($<\text{correlation}>$) to guarantee that messages are delivered to the right process instances [33], [58]. Altogether, we conclude that the session is covered in the instance state, and thus our approach can handle it. This implies that if the session changes in the new version $P_2$, our approach does not allow the process instance to migrate.

The major reason for the migration invalidity is that the process evolution $\triangle$ changes the instance’s “history”. When variable state is taken into account, WIDE is too rigorous while ADEPT may create false positives for migration validity checking. The merit of our approach lies in that it allows the migration of $I$, if $\triangle$ refers to reordering, sequentializing, and parallelizing activities without dependences, or to just removing some irrelevant activities. However, our approach is limited to instance migration validity at the syntactic level (i.e., control flow and data flow), while semantic constraints (e.g., domain knowledge) and temporal constraints in business processes are not considered [62]. Consequently, some instances that violate semantic and temporal constraints may be mistakenly allowed to be migrated by our approach. In addition, our approach does not consider other related issues and scenarios that are orthogonal to migration validity checking, e.g., the migration of deviating instances that result due to ad-hoc changes, how semantic rollback impacts migration validity, and how to apply instance-specific changes to further increase the number of migratable instances [55].

We learn from our study that “late changes” can raise the possibility of valid instance migration, that is, if the change (i.e., evolution) is applied near the end of the process, the number of migratable instances could raise. Thereby, if the effect of changing the front half of a process is the same as that of changing the second half part of a process, the latter is recommended. For example, assume that for an evolution scenario several related activities should be inserted into a process. If these activities do not control or data-dependent on the activities of the original process, we can add them at the end of the process, which definitely enhances the instance migration rate.

To improve the possibility of instance migration and to complement our approach, we employ two strategies: delayed migration and rollback-based migration. Although these strategies have been studied before, variable state is not explicitly considered in prior work. Also, the combination of our approach with these strategies can achieve better result. Chances are that the migration of an instance from its current state is invalid; however, when the instance continues execution, it may enter a state in which its migration is valid. In this case, the strategy of delayed migration becomes feasible. For example, in the case study presented in Section 6.2, $I_{18}$ cannot be migrated to $V_2$ in its current state. Nonetheless, it can migrate in our approach, when it completes executing activity $A_2$. However, if the other two approaches, i.e., WIDE and ADEPT, are employed, $I_{18}$ is not allowed to migrate. The reason for the migration invalidity lies in the fact that the instance has progressed too far, that is, the execution of the instance has already passed an evolution region of the process. With this in mind, the main idea of rollback-based migration is to erase the “effect” (variable value) of certain activities in the trace so that the instance is delivered to a state in which the migration is valid. Besides these two, more advanced migration strategies proposed in the ADEPT framework can also be used for reference [55].

8 Conclusions

This paper formulates the problem of instance migration and migration validity checking in the context of the dynamic evolution of data-aware processes. We first prove that the problem of migration validity checking is in general undecidable. Then, we analyze activity dependences in the trace of the to-be-migrated instance and capture them in an instance dependence graph (IDG). Next, we propose to utilize trace slicing to remove irrelevant activities from the IDG that do not impact the state mapping for instance migration. Based on the pruned instance dependence graph
(PIDG), we come up with a novel migration criterion (approach) for migration validation checking: if there exists a topological sort of activities in the PIDG such that it can be replayed in the new process version, the migration validity of the corresponding instance is satisfied. We also present detailed algorithms realized in our publicly available proof-of-concept tool Dilepis. An extensive experiment and an intensive case study are conducted, the results of which demonstrate that our approach can achieve an improved instance migration rate as compared to state-of-the-art approaches that are based on trace replaying. Additionally, the improvement of the migration rate does not significantly compromise the efficiency of our approach; specifically, our approach scales well with increasing process scale, trace length, and trace quantity, respectively.

Since our work only focuses on instance migration from the perspective of orchestration evolution, the next stage is to extend our approach for the dynamic evolution of data-aware choreographies, where environment properties, i.e., external services, need to be considered.

ACKNOWLEDGMENTS

The authors are grateful to anonymous reviewers for their insightful and constructive comments. This work was supported in part by the National Key R&D Program of China under Grant No. 2017YFB1001801, the Natural Science Foundation of Jiangsu Province under Grant No. BK20171427, the Fundamental Research Funds for the Central Universities under Grant No. 30917011322, the China Scholarship Council under Grant No. 201606845006, NSERC, and the Alexander von Humboldt Foundation under Grant No. 5090551. The work was completed in parts at the University of Toronto, Canada and at the Technische Universität München, Germany. W. Song is the corresponding author of the article.

REFERENCES


**Wei Song** received the Ph.D. degree from Nanjing University, China, in 2010. He is currently an associate professor in the School of Computer Science and Engineering, Nanjing University of Science and Technology, China, and a visiting scholar at Technische Universität München, Germany. His research interests include software engineering and methodology, services and cloud computing, distributed computing, workflow management, and process mining. He was invited to the Schloss Dagstuhl Seminar “Integrating Process-Oriented and Event-Based Systems” held in August, 2018. He is a member of the IEEE and the ACM.

**Xiaoxing Ma** received the Ph.D. degree in computer science from Nanjing University, China, in 2003. He is a full professor with the State Key Laboratory for Novel Software Technology and the Department of Computer Science and Technology, Nanjing University, China. His research interests include self-adaptive software systems, cloud computing, software architecture, and middleware systems. He co-authored more than 60 peer-reviewed conference and journal papers, and has served as technical program committee member on various international conferences. He is a member of the IEEE and the ACM.

**Hans-Arno Jacobsen** received the PhD degree at Humboldt Universität in Germany. France; he engaged in postdoctoral research at INRIA near Paris, before moving to the University of Toronto in 2001. He is a professor of computer engineering and computer science and directs the activities of the Middleware Systems Research Group. He conducts research at the intersection of distributed systems and data management, with particular focus on middleware systems, event processing, and cyber-physical systems (e.g., smart power grids.). He received the Alexander von Humboldt-Fellowship award to engage in research at the Technische Universität of München, Germany. He is an IEEE Senior Member.