Increasing Availability of Workflows Executing in a Pervasive Environment

David Richard Schäfer Thomas Bach Muhammad Adnan Tariq Kurt Rothermel
Institute for Parallel and Distributed Systems, University of Stuttgart, Germany
Email: ⟨firstname.lastname⟩@ipvs.uni-stuttgart.de

***Abstract*—Workflows have gained enormous importance to organize and manage business processes. With the recent advent of smartphones and mobile applications, traditional business process management is shifting. Now, long-running business processes (workflows) have to be executed in large-scale distributed and pervasive environments. Due to the heterogeneity and high dynamicity of such environments, they are vulnerable to frequent communication and device failures and, thus, impose new requirements on the execution of workflows. To increase the availability, we concurrently executed restructured replicas of workflows on multiple nodes. We developed techniques to generate differently structured replicas and propose a metric that identifies the set of replicas that ensures the highest availability during execution. Finally, we presented a distributed algorithm to coordinate and synchronize the concurrent execution of the identified replicas while maintaining the original workflow semantics. Our methods approximately double the availability during execution, while our generation techniques produce almost optimal replicas over a hundred times faster.**

I. Introduction

Workflows are the de facto standard to create, organize and maintain automated business processes [1]. The modular structure of workflows allows to refine and optimize complex business processes in a simple and flexible manner. Hence, workflows are widely used in many areas, such as healthcare [2], logistic [3], manufacturing and urban mobility [4]. Workflows model a process as a set of interrelated activities, where each activity may trigger applications or services over time. Thus, a workflow explicitly defines the temporal structure of a complex (possibly long-running) process. The recent advances in pervasive computing tremendously influence the way traditional businesses are operated. New trends such as *customer tailored value added services* and *bring your own device* are becoming popular. The business processes that were previously running in isolation in back-end computer systems now need to interact with processes (or services) on mobile devices located virtually anywhere. This arises the need to execute business workflows in heterogeneous, distributed, and highly dynamic environments. Workflows provide a flexible, powerful, and platform independent *programming-in-the-large* paradigm to handle the complexity in such environments [4]. Workflows can be manipulated and adjusted to the dynamic changes in the environment (e.g., changing user behavior, lack of required resources, etc.) making them ideally suited for pervasive applications.

Clearly, pervasive environments raise some new requirements on the execution of workflows. Due to varying network coverage, high bit error rates, power constraint mobile nodes, unreliable services and frequent disconnections, pervasive and mobile applications are prone to failures. Workflows need to be executed such that they can survive the failures of computing nodes, networks, services, service providers and other resources. Existing approaches usually focus on increasing the availability of workflow executions by incorporating fault handling and recovery mechanisms into the workflow model [5]. These mechanisms, however, do not consider faults of the infrastructure (such as failure of computing nodes, networks, etc.). Instead, they assume the presence of a separate layer which is capable of dealing with such failures. The approaches that explicitly consider infrastructure failures to increase the availability of workflow executions mostly employ replication strategies. Following the primary-backup strategy, only one replica is executing the workflow at any time and the execution state is transferred to all other replicas [6], [7]. However, this can only provide availability against the failures of computing nodes. In addition to that, active replication strategies [8], [9] can mask failures of the network because they allow the concurrent execution of activities (or processing operators) on more than one replica. Active replication, however, lacks appropriate mechanisms to safeguard that the replicated execution has the same effect as the non-replicated execution. In a nutshell, executing workflows such that they provide availability in a pervasive and mobile environment is still an open challenge.

This paper presents a novel approach to ensure availability by concurrently executing workflow replicas with differently ordered and alternative activities, i.e., concurrently executing structurally different replicas. To ensure that all these structurally different replicas provide the same functionality, we use a workflow specification which has to be satisfied by all replicas. These replicas can be obtained by manipulating the set of interrelated activities and the temporal structure of the workflow. Executing structurally different replicas of a workflow increases the availability because the replicas may differ in the services or resources they use, the set of activities they execute, and the execution order of the activities they have in common. For instance, the dependency on the availability of services will significantly decrease if the intersection of the services, used by two functionally equivalent replicas, is small or even empty. However, even if two replicas use the same set of services, letting them access these services in a different order will substantially reduce their vulnerability, in particular, if executions are long-lived.

The main contributions of this paper can be summarized as follows. 1) We propose an *availability metric* that analyzes the degree of availability of concurrently executing a set of replicas. 2) We developed algorithms to create structurally different but functionally equivalent replicas, which guarantee

Published in Proceedings of 2014 IEEE International Conference on Services Computing (SCC), 2014, Anchorage, Alaska, USA

high availability during execution. 3) We present a distributed algorithm to coordinate the concurrent execution of workflow replicas. The coordination algorithm ensures that the replicated execution has the same effect as the non-replicated execution. 4) Through extensive evaluations, we show the benefits of our proposed approach in terms of availability, timeliness, and scalability.

II. System Model and Problem Formulation

The workflows are modeled in a declarative workflow language based on ***l**inear **t**emporal **l**ogic* (LTL), such as Declare [10]. The declarative workflow language allows us to constrain the order in which the activities of a workflow are allowed to execute by constructing LTL formulas (on the activities) from a defined set of linear time operators [11]. All other activity-sequences, which are not explicitly forbidden by the LTL formula, are allowed. In the following, the term *workflow specification* is used for the LTL formula that specifies a workflow.

To clarify our contributions, we constructed a scenario in the context of drone package delivery services, which have gained an enormous attention lately. DHL already successfully tested a drone delivery and companies like Amazon have shown great interest in the technology[1]. The package delivery companies usually compete for delivery times. Drones are especially interesting to these companies because drones are not hindered by traffic jams and can deliver packages around the clock without much human involvement. In particular, drones can be used to pickup (one or more) packages from customers and deliver them to the nearest post office (or warehouse) and vice versa. However, between post offices (or warehouses) it will still be much more efficient to use large trucks because of the amount of packages to be transported. The whole package delivery process chain can be automated and modeled as a workflow. Fig. 1 shows an exemplary workflow for picking up one package. To keep delivery times low, the execution of the workflow should be finished latest when the drone reaches the post office. The LTL formulas in Fig. 1 depict that all activities in the workflow specification need to be executed. This is specified by the $\diamond$ (*finally*) operator. For example, $\diamond A$ specifies that A has to hold *finally*, i.e., it needs to be executed at least once. Also, D can only be executed after A has finished, because a picture can only be analyzed after it has been taken. The respective LTL formula $\neg D\mathcal{U}A$ specifies that D must not execute *until* A was executed and that A finally has to be executed. With these LTL constraints, the workflow specification of Fig. 1 can be constructed.

Activity G should be executed only once because otherwise the system assigns one package multiple times. This leads to trucks that reject other packages because the system falsely assumes the capacity limit of the truck is already reached. Hence, activity G is *non-idempotent*. In contrast, activity B, checking the truck availability, can be executed arbitrarily often without any harm. Thus, this is an *idempotent* activity. We assume that the workflow designer defines a *cardinality* $\theta(X)$ for every activity X that specifies how many times the activity can be executed maximally. We consider the cardinality to be part of the workflow specification because it can be specified

[1] http://www.cbsnews.com/news/dhl-testing-delivery-drones/

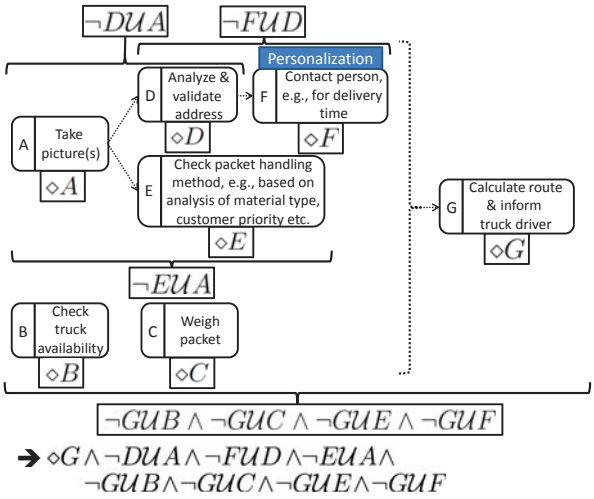

Fig. 1. Workflow specification of a drone picking up one package. An arrow means that the activity at the end of the arrow is only allowed to execute after the activity at the root of the arrow has terminated (inspired by Declare [10]). The boxes show the LTL formulas that model the specific behavior.

within LTL. Returning to our previous example, we specify the cardinality of the non-idempotent activities F and G to be $\theta(F) = 1$ and $\theta(G) = 1$, whereas all other activities are idempotent.

A workflow is executed by an ***e**xecution **e**ngine* (EE). However, the EE needs a sequence of activities that it executes step-by-step. This means that a sequence of activities that does not violate the workflow specification needs to be generated and given to the EE. For example, $ABCDEFG$ is a valid sequence of the workflow specification of Fig. 1. To execute an activity, the EE calls a service that implements the functionality required by the activity. The service can be either available locally on the device running the EE or offered by a service provider. For example, when executing activity D the EE calls an image analysis and address validation service. This service returns if the address is valid or not.

The EE can run on any computing node of the system, such as a smartphone, a tablet, an embedded system of any type or even on a virtual machine in the cloud. These computing nodes may use different technologies to communicate with each other or with the services provided by 3rd party service providers. Pervasive and mobile environments are characterized by high bit error rates and frequent disconnections due to handover, fading and bandwidth shortages. This may result in the failure of communication between computing nodes and service providers. Consider that the drone flies over an area with no reception. Then, it cannot execute the activities which require services that are only available on the internet.

Moreover, any computing node or service provider can experience temporal or permanent failure. For example, the smartphone of the person that is contacted in the personalization step (activity F) might have ran out of battery. Any failure and disconnection means at least a delay in workflow execution. However, to ensure value added service to the customers (i.e., expedite shipment of packages) the workflow should be finished in a timely fashion. For instance, in the drone delivery scenario, the workflow that needs to be executed for every picked up

package should be finished before reaching the post office, so that the packages can be dropped directly to their allocated trucks and the shipment (of the packages to the next post office) proceeds without delays.

Given a set of computing nodes and service providers in a highly dynamic environment and a declarative workflow specification in LTL, our objectives are

- to increase the availability of a workflow execution in the presence of i) failures of execution engines, ii) failures of services and service providers and iii) failures of the communication, and
- at the same time to reduce the execution time of the workflow.

III. Approach Overview

To increase the availability in our drone delivery scenario, we could install redundant hardware on the drone and replicate the execution engine (EE) there. In these EEs, we concurrently execute activity-sequences that comply with the workflow specification. In the following, we call these activity-sequences *replicas*. If one of the EEs fails during the execution of its replica, the redundant EE will still continue. Thus, the workflow stays available even in the presence of node failures. However, this does not ensure availability in case of failures of the communication. For instance, if the drone has no reception, all replicas cannot communicate with services hosted on the internet. Thus, placing a replica on a server on the internet improves the availability. If the replica on the drone fails during execution because of no reception, a replica on the internet can still continue execution.

Nevertheless, two structurally identical replicas (i.e., replicas executing the same activity-sequences) may fail to execute an activity if the required service is not accessible. For example, in Fig. 2 both replicas $r_{e1} = ADFECBG$ and $r_{e2} = ADFECBG$ will fail (irrespective whether they are hosted on the drone or on the internet) if the service of activity F is not available because the smartphone of the receiver (on which the service is hosted) currently has no reception. To overcome this limitation, we use different activity-sequences in the replicas. The LTL specification allows us to reorder the activity-sequences. The replica $r_{e3} = BACDEFG$ is such an reordered replica. This replica will not fail because it executes F when the receiver has entered an area with reception.

Reordering increases the availability in the presence of transient failures. However, if the service is permanently not available, both replicas (with reordered activities) will still fail. Permanent failures of services can be masked by exploiting the possibility of defining alternative activities in the workflow specification (LTL). Consider that the activity D in Fig. 2 has the alternative to execute the activities H and I instead, such that $\neg I \mathcal{U} H$. Instead of using a service that directly analyzes and validates the address (as required by activity D), H analyzes the picture and I validates the address. Thus, these activities rely on different services. If the service required for the execution of activity D permanently fails, another replica $r_{e_4} = BACHIEFG$ can still execute. In conclusion, the structure of replicas (in terms of activity-sequences and their alternatives) has a significant influence on the availability of a workflow execution. In Sec. IV, we address the problem of efficiently creating k structurally different replicas, which guarantee high availability during execution.

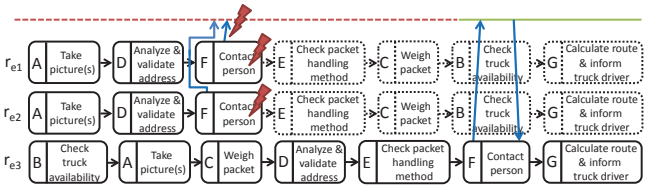

Fig. 2. Showing the influence of the availability of the service that activity F calls on the execution of three replicas.

In the presence of cardinality constrained activities (such as non-idempotent activities), it is necessary to synchronize the replicas to ensure that the replicated execution has the same effects as a non-replicated execution. For instance, in Fig. 2 activities G and F should not be executed more than once, as mentioned in Sec. II. Thus, the replicas need to synchronize such that these activities are only executed by one of the EEs. This means that in the case of structurally identical replicas only one EE is making progress while the other replicas wait. However, our approach of using structurally different replicas speeds up the execution by avoiding these waiting times and reusing the results of activities that were already executed by other replicas. In Sec. V, we present an approach to coordinate the concurrent execution of structurally different replicas, such that the execution complies with the workflow specification.

IV. Gerneration and Identification of Highly Available Replicas

In this section, we present a metric that rates a set of replicas depending on their structure to predict the availability during concurrent execution. Then, we show how to generate structurally different replicas from a workflow specification. Afterwards, we analyze the complexity of the generation process and present more efficient techniques.

A. Availability Metric

Our goal is to select a set of replicas that provides high availability when executed concurrently on disjoint computing nodes. We first define a metric that rates replica sets according to their expected availability during concurrent execution. As explained in Sec. III, a replica set provides higher availability during execution, the more the replicas of the set structurally differ. Therefore, the proposed availability metric rates replica sets according to the requirements that 1) the time offset between two executions of one activity in different replicas should be as big as possible, 2) alternative activities should be used as much as possible, and 3) replicas with few activities should be preferred. In the following, we first define the availability rating of a set of two replicas r_1 and r_2 denoted as $D(r_1, r_2)$. Let a_{max} be the number of activities of a replica with the longest possible activity-sequence that complies with the workflow specification. Likewise, let a_{min} be the number of activities of the replica with the fewest activities.

To account for the above mentioned first requirement, we calculate the time offset between the two executions of one activity A in the two replicas r_1 and r_2. The time offset of

an activity A is denoted as $O(A, r_1, r_2)$ and its calculation is based on the position of the activity A in replica r_1 and the position in replica r_2. To incorporate the second requirement, we need to consider the cases where an activity A might not occur in one of the replicas because of alternatives. For these disjunct activities, $O(A, r_1, r_2)$ returns the maximum possible offset a_{max}.

To calculate the rating, we add the time offsets of all activities occurring in the two replicas. This gives a metric on how much these two replicas differ. In general, the time offset based rating increases with the number of activities in a replica. However, if a replica has more activities, more failures can occur. Therefore, to fulfill the third requirement, the rating of replicas with many activities should be decreased. Let function $L(r_1)$ calculate how many activities replica r_1 has more than a_{min}. The rating is reduced by a_{max} for every additional activity, i.e., it is reduced by $L(r_1) \cdot a_{max}$. Thus, the availability rating $D(r_1, r_2)$ is calculated by adding the offset of all activities in the two replicas and then subtracting the reduction for the additional activities (Eq. 1).

$$D(r_1, r_2) = \left(\sum_{A \in r_1 \cup r_2} O(A, r_1, r_2) \right) - (L(r_1) + L(r_2)) \cdot a_{max} \quad (1)$$

Now, we generalize Eq. 1 to rate a set S of k replicas. We calculate the availability rating of all pairs of replicas and sum up the results (Eq. 2).

$$R(S) = \sum_{(r_i \in S)} \sum_{(r_j \in S \setminus r_i)} D(r_i, r_j) \quad (2)$$

Note that we defined the time offsets based on the positions of the activities in the replicas. This means that the availability metric is only accurate if all activities have the same duration. In reality, the activities may have varying execution times. However, in our evaluations, we show that even if we vary activity execution times, the replica availability is not effected (cf. Sec. VI).

B. Generation and Selection of Replicas

In this section, we first present a method to generate differently structured replicas from a workflow specification. Afterwards, we select the highest rated set of replicas w.r.t. the availability metric. To generate the replicas, we use model checking methods to translate the LTL specification into an automaton by expanding the LTL formula step-by-step. The automaton provides the information to deduce all possible replicas that conform to the workflow specification. First, the LTL formula is associated with the entry node of the automaton (cf. Fig. 3 node X). Then, the automaton node is expanded according to a set of rules [12], [13] to simplify the formula into three parts (cf. Fig. 3): I) Activities that need to be executed directly, II) promises ($\mathcal{P}$) that need to be fulfilled eventually, and III) a formula that defines what needs to hold after the activities of part I) have been executed [13]. Part I) is associated with the label of the edge to the next node, i.e., the activities need to be executed to reach the next node (in Fig. 3: activity A to reach automaton node Y from X). The part II) (i.e., the promises) is used to check if the current activity-sequence fulfills the LTL specification. If there are promises, then the workflow specification is not yet fulfilled, i.e., the automaton node after this transition needs to be expanded further. This information is stored in the edge between the respective nodes. In Fig. 3, there is the promise to eventually execute B, which means that the next automaton node Y needs to be expanded further. Finally, part III) is attached to the next node of the automaton because it specifies what has to hold after the activities of the part I) have been executed. In Fig. 3, the formula of part III) is associated with node Y. Applying the tableau techniques to this new automaton node, i.e., to the associated formula, will expand this node to the following ones. This procedure of expansion repeats until the formula is fully expanded to the complete automaton. From the labels of the edges, all n replicas

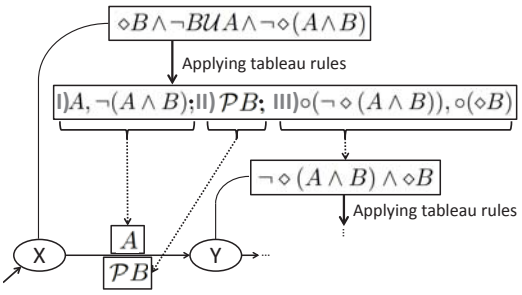

Fig. 3. One step of an exemplary expansion of the LTL formula: $\diamond B \wedge \neg B \mathcal{U} A \wedge \neg \diamond (A \wedge B)$. [13]

can be constructed by strategically going through the transitions of the automaton. This solves the generation problem.

Out of all n generated replicas, we need to select the set of size k that will achieve the highest availability during concurrent execution. We solve this by rating all sets of size k using the availability metric and select the one with the highest rating.

Complexity

The proposed generation technique has a high complexity. In fact, LTL satisfiability is a PSPACE-complete problem [14] and, therefore, finding satisfying traces, i.e., replicas, is a PSPACE-complete problem. This means that the generation process might result in a large number of replicas.

The generated replicas are the input of the selection problem, which is to find the best rated set of size k from n replicas. It can be mapped to the ***m**aximum **e**dge-**w**eighted **c**lique **p**roblem* (MEWCP), which is NP-hard [15]. The MEWCP is a generalization of the maximum clique problem [15]: Given a graph $G = (V, E, f)$, where f is a function that assigns a weight to every edge, i.e., $f : E \to \mathbb{R}$, find the clique in which the sum of all of its edges is maximal and the number of nodes of the clique $\leq k | k \in \mathbb{N}$.

Lemma 1: Selecting the best rated set of k replicas out of n replicas, can be mapped to the ***m**aximum **e**dge-**w**eighted **c**lique **p**roblem* (MEWCP).

Proof sketch Given all n replicas, compute the availability ratings for every pair of replicas and write them into a 2D-matrix. Using this as an adjacency matrix, a complete graph with n vertices (representing the replicas) can be created, where the availability ratings are the edge weights. The problem of

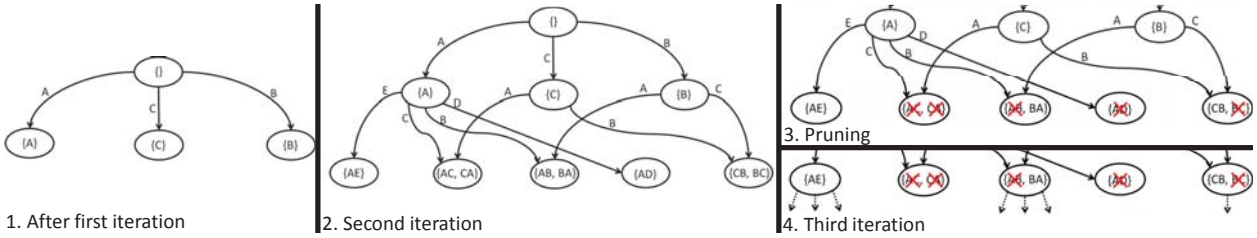

Fig. 4. This is an example that shows how pruning is applied while expanding the workflow specification of Fig. 1. After each iteration three replicas are selected, i.e., $f_{Flows} = 3$.

finding the best rated set of size k is equal to solving the MEWCP with a clique of size k in the graph.

C. Refinements

In the following, we propose two techniques to tackle the high run-time and memory complexity of the presented generation and selection approach. Our first technique significantly reduces the run-time of the selection process. The second technique omits the generation of all replicas and, thereby, reduces the memory consumption of the generation problem.

Selection Problem: To solve the MEWCP, we used a ***b**inary **q**uadratic **p**rogramming* (BQP) formulation of the problem [15]: Given a symmetric $n \times n$-matrix Q composed of all availability ratings (as presented above), find the vector $v \in \{0,1\}^n$, such that the outcome of Eq. 3 is maximal, where the sum of all elements of vector v must be k.

$$f(v) = \frac{1}{2} v^T Q v \tag{3}$$

If the element i of vector v is equal to 1, the replica of column i in the matrix is part of the set. If it is 0, it is not. To maximize the outcome of Eq. 3, we apply ***s**imulated **a**nnealing* (SA), as it is an established strategy to produce almost optimal results within a short amount of time for BQP problems [16], [17].

Generation Problem: Our evaluations show that SA drastically reduces the run-time of the selection problem, while producing almost optimal replicas (cf. Sec. VI). However, if the workflow specification has many activities, the generation of all replicas easily becomes infeasible due to run-time and memory consumption. Thus, for formulas that lead to huge automata, the full generation has to be omitted and a different strategy has to be applied. To tackle this problem, we introduce the ***a**vailability **p**rediction **t**echnique* (APT). The idea is to prune the automaton during expansion by only expanding those paths of the automaton that might lead to replicas that are part of sets with high availability ratings. The activity-sequences that are created during the expansion and still have to fulfill promises (cf. Sec. IV-B) are called *intermediate replicas*.

In the following, we describe how to perform pruning during the expansion of the automaton. We expand all automaton nodes that currently need further expansion. This step is referred to as an *iteration* (cf. Fig. 4). After each iteration, all newly created (intermediate) replicas, generated by the iteration, are rated by the APT. The difference between APT and the availability metric is twofold. Firstly, we do not know the replica with the fewest/most activities during the expansion of the automaton. Therefore, to determine the ratings (of the replicas), APT calculates a_{max} and a_{min} based on the currently available satisfying and intermediate activity-sequences. Secondly, to omit the selection problem, APT only calculates the rating for every pair of replicas (i.e., only sets of two replicas) and ranks the replicas according to the pairwise ratings. Then, we select the f_{Flows} best rated replicas, where $f_{Flows} \geq k | f_{Flows} \in \mathbb{N}$. For instance, in Fig. 4, the second iteration leads to eight intermediate replicas. Three are selected, while the rest gets pruned before the third iteration. The paths of the automaton that produce the selected replicas are expanded further if the replicas are intermediate (cf. Fig. 4). For fully expanded branches of the automaton that do not satisfy the workflow specification, i.e., that are *dead-ends*, the expansion algorithm goes backwards and selects a new branch to expand. For more details on the generation and selection of replicas, we refer the reader to our previous work [18]. In our evaluation, we show that the pruning strategy significantly improves the run-time, however, compared to SA, it produces results that have a lower, but reasonable availability during execution (cf. Sec.VI).

V. Replica Coordination

In the previous sections, we presented methods to efficiently generate and select a set of replicas that has a high availability rating. This section addresses concurrent execution of these replicas on different computing nodes such that the workflow specification is preserved (i.e., the activities are executed respecting their cardinality in the workflow specification).

Before the computing nodes begin to execute the replicas (i.e., activity-sequences of a workflow) they elect a coordinator. The election of a coordinator under dynamic conditions is a well researched topic [19] and therefore, not addressed in this paper. The coordinator maintains a data structure to annotate i) a list of all activities of the workflow specification and their cardinality constraints, ii) activities currently being executed by the computing nodes, iii) the number of executions of each activity along with the execution results. This data structure is updated during the execution of replicas and is used for coordinating the execution of cardinality constrained activities.

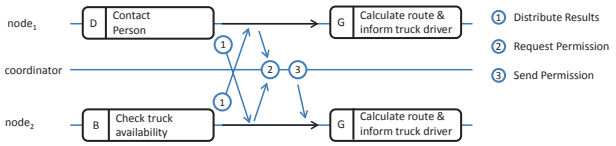

Fig. 5. This figure shows the synchronization and coordination of activities. To increase the clarity of this figure the coordinator is depicted separately.

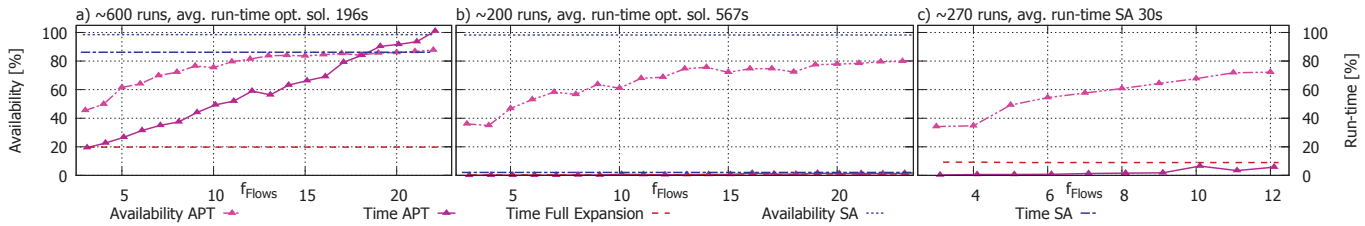

Fig. 6. Comparing different generation and selection strategies, normalized to the run-time/availability of the optimal solution (a,b) or SA (c).

In general, all computing nodes execute the activity-sequences of their individual replicas (cf. Fig. 5). When a node has executed an activity X successfully, an *update message* along with the execution results of that activity is sent to all other nodes (cf. Fig. 5, *message 1*). On receiving an update message for an activity X, each node that has not previously executed X marks it as finished and stores the results. Likewise, the data structure maintained by the coordinator is also modified on receiving an *update message*. While idempotent activities are executed without coordination, activities with cardinality constraints need to be coordinated. For this purpose, a node p requests execution permission from the coordinator before executing a cardinality constrained activity Y (cf. Fig. 5, *message 2*). Either the coordinator enables the node p to execute the activity Y (cf. Fig. 5, *message 3*) or the node p waits until it receives the results of Y in an *update message* from another node. In the latter case, the node p skips the execution of the activity Y, updates the state of the execution engine with the result of activity Y, and then continues with the execution of its replica.

To detect the failure of the coordinator and/or computing nodes, a timeout based error detection strategy is used. In case of a node failure, a new node is provisioned by the coordinator. If the coordinator fails, a re-election is started.

VI. Evaluations

In this section, we evaluate i) our replica generation techniques (APT and SA) w.r.t. availability and generation-time in comparison to the optimal solution, ii) our replica coordination algorithm in terms of the influence of cardinality constraints of activities on the (execution) speed-up and the message overhead (scalability), and iii) the accuracy of our availability metric to predict the availability of concurrently executing a set of replicas.

To evaluate our replica generation techniques, we use the SPOT library [13] for translating the LTL-based workflow specifications into automata. From the automata, we derive all replicas. We also integrated APT into SPOT. In the following, we used workflow specifications with eight activities by randomly combining LTL constraints [10] and selected sets of size $k = 3$ (unless otherwise noted). For APT, there is an additional parameter (f_{Flows}, cf. Sec. IV-C), for which we evaluated the range from k up to 22. Because of the high complexity of the generation and selection, we only expand the full automaton for workflow specifications that result in 1) 3 to 500 replicas (for the optimal solution) and 2) 500 to 2000 replicas (for SA) . The coordination algorithms are implemented as a protocol in the peer to peer simulator, PeerSim [20], where each peer (node) executes an individual workflow replica. Because all failure types (cf. Sec. II) lead to failed executions of activities, we modeled all failures as activity failures. Each activity has a failure probability of 5%. For evaluation, we generated failure traces, to expose the replicas, generated by the different techniques, to the same failure patterns. We also repeated all experiments with a varying execution speed of the nodes ($\pm 10\%$) and discovered that such variances do not influence the average availability.

A. Replica Generation

In this section, we compare the generation strategies (optimal solution, APT and SA) in terms of run-time and availability (w.r.t. the availability metric). Since the APT does not generate all possible replicas, it approximates a_{min} and a_{max} (cf. Sec. IV-C). To make the rating of the found replica set of APT comparable to the optimal solution and SA, we evaluated it with the correct values of a_{min} and a_{max}. Fig. 6 shows the run-time and availability of the different algorithms. In the following, we grouped our measurements by the number of replicas that were produced by the full generation of the automata.

Formulas resulting in 3 to 500 replicas (cf. Fig. 6a): SA on average needs 85% time compared to the optimal solution algorithm. APT is very fast for small f_{Flows}, however, always selects sets with a lower availability rating. For $f_{Flows} < 4$, APT is even a bit faster than only expanding the full automaton without starting any selection process. In contrast to that, for $f_{Flows} \geq 22$, APT is even slower than generating the optimal solution because of the overhead while deciding which branch to follow after each iteration during automaton expansion. Thus, APT is only suitable for small f_{Flows}.

Formulas resulting in 200 to 500 replicas (cf. Fig. 6b): In comparison to Fig. 6a), SA is significantly faster because we have to choose from at least 200 replicas. SA finds replicas with an availability rating of around 98% while taking only about 2% of the time of the optimal solution algorithm. In contrast to that, APT only generates results with 80% availability when following a large number of flows ($f_{Flows} \geq 11$). For small f_{Flows} the availability drops down to 38%. This is because there are more replicas in general and percentually less of them can be explored because of pruning. However, APT needs only a fraction of computation time compared even to SA.

Formulas resulting in 500 to 2000 replicas (cf. Fig. 6c): For more than 500 replicas, the full generation of the automaton is not feasible. Thus, we use SA as a reference for normalization. APT again only reaches an availability of 80% for $f_{Flows} > 10$. It, however, is clearly faster than only the full expansion of the automaton without even starting SA. This is a great advantage

for large workflows and might even be the only feasible strategy in these cases.

These measurements show that both SA and APT reduce the run-time of the replica generation and selection procedure significantly. Both strategies have trade-offs as they reduce the availability. For large workflows, however, APT is the only practical choice in terms of run-time.

B. Coordination Methods

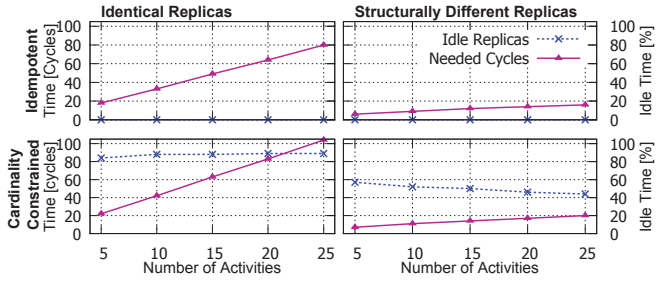

Fig. 7. Execution of replica sets ($k = 10$) consisting of idempotent and cardinality constrained activities using structurally different and identical replicas.

In this section, we evaluate our coordination methods in terms of the influence of cardinality constraints of activities on the (execution) speed-up and the message overhead. Fig. 7 shows how the use of different replicas decreases the number of idle replicas and the execution time. The amount of cardinality constrained activities in general increases the execution time and number of idle cycles. This is because cardinality constrained activities can only be executed by a limited number of nodes, while the others have to wait.

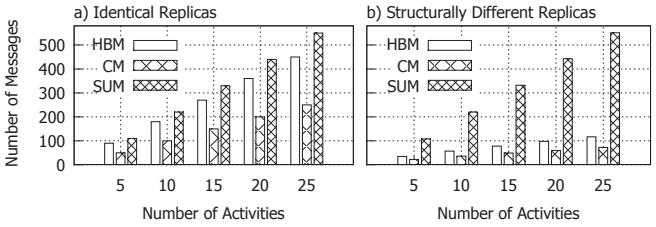

Fig. 8. Message overhead for 10 cardinality constraint replicas comparing identical to structurally different replicas.

Fig. 8 shows the scalability of our coordination mechanism. The diagrams depict that the number of messages grows proportional to the number of executed activities. If structurally different replicas are used, the number of coordination messages (CM) decreases. Fig. 7 shows that less replicas are idle when using structurally different replicas. Since we use a timeout based error detection, a decreased number of idle replicas also decreases the number of heartbeat messages (HBM). When executing differently structured replicas, less HBMs are needed because the controller receives state updates each time a node finishes an activity. These state update messages (SUM) are equivalent for structurally different and identical replicas because the number of activities is same for both cases.

C. Replicated Execution

So far, we evaluated the methods to generate replicas and to coordinate their execution. In this section, we verify that the availability rating calculated by our availability metric is in accordance with the availability during concurrent execution. The availability is represented by the execution time because a faster execution is congruent to a highly available execution (i.e., if the execution is not available, it does not proceed). For this evaluation, we use the replicas generated in Sec. VI-A. We compared the execution time of these replica sets with the concurrent execution of a set of identical replicas.

To simulate failures, we generated *failure traces* that specify at which time an activity fails. We used a failure probability of 5% and evaluated every replica set against ten different failure traces. When a failure occurs, the defective node is detected by the coordinator and a new node is initialized with the state of the failed node. This was modeled by a short delay. During this time a replica is unavailable.

Fig. 9, shows the execution time of the replicas generated in Sec.VI-A (cf. Fig. 6) for flows resulting in (a) $3 \leq n \leq 500$ replicas and (b) $500 \leq n \leq 2000$ replicas. It confirms that the optimal solution results in the most available execution, followed by SA, APT, and finally the set of identical replicas (which is represented by 100%). This shows that the ratings of our availability metric are correct. The graph also confirms that increasing f_{Flows} in the APT results in a higher availability. The replicated execution of the optimal solution sets take around 55% of execution time compared to identical replicas. SA results take around 56% of the time, followed by APT with around 65% ($f_{Flows} = 5$). When increasing f_{Flows}, APT can even reach 60% (which, however, increases the generation time). This shows that APT is useful for efficiently generating and selecting replicas, accepting a decreased availability. It, however, should only be used with a small or minimal f_{Flows}. If availability is very critical, SA can be used, which, on the downside, implies a higher generation overhead.

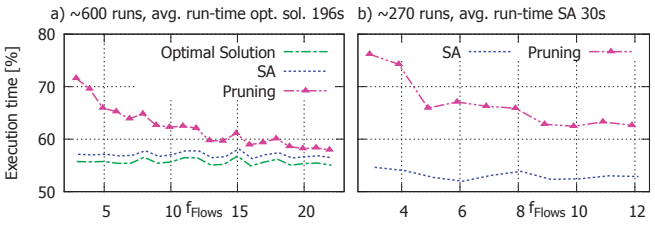

Fig. 9. Simulations comparing execution time of SA and APT, normalized to the execution time of identical replicas.

In addition, we generated replicas using APT for large workflows where SA is too time consuming due to the full generation of the automaton. We generated replica sets of size $k = 5$ for workflows with 15 different activities. The generation of those replicas (with $f_{Flows} = 5$) took an average time of $75s$. The execution took around 83% of the execution time of sets with identical replicas. This shows the benefits of APT for cases where SA and the full generation is no longer practical.

VII. Related Work

Since availability and robustness are important requirements in mobile and pervasive applications, these problems were

already approached in many different ways. In the following, we discuss such approaches and their shortcomings.

Checkpointing and logging are two approaches that are able to mask node failures [21]. Checkpointing records the state of a distributed system consisting of the state of all nodes and all communication channels. In the case of a failure, the system can rollback to the last checkpoint, and therefore, progress might be lost [21]. Logging overcomes this problem by recording all exchanged messages, so that they can be replayed to recreate the system state before the failure [21]. This, however, introduces an additional delay until the system is available again.

Another approach to mask node failures is the primary-backup strategy. Following this strategy, a primary node executes the workflow and transfers its state to all backups [6], [7]. In case of a failure, it takes some time to detect the failure and transform a backup into a primary copy during which the execution cannot proceed. Moreover, this scheme cannot mask service or communication failures.

A different area of research that addresses availability is task allocation, where activities (tasks) are mapped to relatively more reliable nodes [22], [8]. A probabilistic model predicts the possibility of node and communication failures. Activities are allocated such that the overall failure probability is minimized. Task allocation, however, does not directly mask any failures. To overcome this problem, some approaches incorporate replicated execution of activities [8]. These, however, do not provide methods to ensure that the replicated execution has the same effects as a non-replicated execution (e.g., they do not consider cardinality constrained activities).

Another replication approach to increase the availability is to schedule several services in parallel for a single service call of an activity [23], [6]. If none of the services responds, a second set of services is started. These service scheduling techniques are very capable of masking service failures, but do not provide availability in case of failing execution nodes. They also do not provide mechanisms to ensure that the replicated execution has the same effects as a non-replicated execution.

VIII. Conclusion

Distributed and pervasive environments are prone to many different kinds of failures. Thus, ensuring availability of workflows executing in such environments is an important issue. In this paper, we showed how to increase the availability of a workflow by concurrently executing structurally different replicas. We developed an availability metric that rates a set of replicas w.r.t. their availability during concurrent execution. Based on this, we defined methods to efficiently generate highly rated sets of replicas from a formally defined workflow. To execute such replicas sets, we created a distributed execution environment that is capable of running and coordinating several replicas in parallel without violating the original semantic of the workflow. Finally, we showed that in the presence of failures the coordinated execution of the generated replica sets take only 55% of the time of executing a set of identical replicas.

Acknowledgment

The authors would like to thank the European Union's Seventh Framework Programme for partially funding this research through the ALLOW Ensembles project (project 600792).

References

[1] R. K. Ko, S. S. Lee, and E. W. Lee, "Business process management (bpm) standards: a survey," *Business Process Management Journal*, 2009.

[2] H. Wolf, K. Herrmann, and K. Rothermel, "Flexcon–robust context handling in human-oriented pervasive flows," in *On the Move to Meaningful Internet Systems*. Springer, 2011.

[3] ——, "Modeling dynamic context awareness for situated workflows," in *On the Move to Meaningful Internet Systems Workshops*. Springer, 2009.

[4] K. Herrmann, K. Rothermel, G. Kortuem, and N. Dulay, "Adaptable pervasive flows-an emerging technology for pervasive adaptation," in *IEEE Self-Adaptive and Self-Organizing Systems Workshops*, 2008.

[5] N. Russell, W. Aalst, and A. Hofstede, "Workflow exception patterns," in *Advanced Information Systems Engineering*, E. Dubois and K. Pohl, Eds. Springer Berlin Heidelberg, 2006.

[6] H. Guo, J. Huai, H. Li, T. Deng, Y. Li, and Z. Du, "Angel: Optimal configuration for high available service composition," in *IEEE Web Services*, 2007.

[7] J. Lau, L. C. Lung, J. da Fraga, and G. Santos Veronese, "Designing fault tolerant web services using bpel," in *IEEE/ACIS Computer and Information Science*, 2008.

[8] I. Assayad, A. Girault, and H. Kalla, "A bi-criteria scheduling heuristic for distributed embedded systems under reliability and real-time constraints," in *Dependable Systems and Networks*, 2004.

[9] M. Völz, B. Koldehofe, and K. Rothermel, "Supporting strong reliability for distributed complex event processing systems," in *IEEE High Performance Computing and Communications*, 2011.

[10] M. Pesic, "Constraint-based workflow management systems: shifting control to users," Ph.D. dissertation, 2008.

[11] E. A. Emerson and J. Y. Halpern, ""sometimes" and "not never" revisited: on branching versus linear time temporal logic," *J. ACM*, 1986.

[12] J.-M. Couvreur, "On-the-fly verification of linear temporal logic," in *FM'99 — Formal Methods*. Springer Berlin Heidelberg, 1999.

[13] A. Duret-Lutz and D. Poitrenaud, "Spot: an extensible model checking library using transition-based generalized büchi automata," in *IEEE Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, 2004.

[14] A. P. Sistla and E. M. Clarke, "The complexity of propositional linear temporal logics," *J. ACM*, 1985.

[15] B. Alidaee, F. Glover, G. Kochenberger, and H. Wang, "Solving the maximum edge weight clique problem via unconstrained quadratic programming," *European Journal of Operational Research*, 2007.

[16] K. Katayama and H. Narihisa, "Performance of simulated annealing-based heuristic for the unconstrained binary quadratic programming problem," *European Journal of Operational Research*, 2001.

[17] P. Merz and B. Freisleben, "Greedy and local search heuristics for unconstrained binary quadratic programming," *Journal of Heuristics*, 2002.

[18] D. R. Schäfer, "Robust execution of workflows in a distributed environment," Diploma Thesis, University of Stuttgart, IPVS, 2013.

[19] B. Awerbuch, "Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems," in *ACM symposium on theory of computing*, 1987.

[20] Last visited: 2014-04-11. [Online]. Available: http://peersim.sourceforge.net/

[21] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*, no. 3, 2002.

[22] Y. Gu, Q. Wu, X. Liu, and D. Yu, "Improving throughput and reliability of distributed scientific workflows for streaming data processing," in *IEEE High Performance Computing and Communications*, 2011.

[23] S. Stein, T. R. Payne, and N. R. Jennings, "Flexible provisioning of web service workflows," *ACM Trans. Internet Technol.*, 2009.