Flexible Failure Handling for Cooperative Processes in Distributed Systems

Artin Avanes Johann-Christoph Freytag

Database and Information System Group
Humboldt-Universität zu Berlin
Unter den Linden 6, 10099 Berlin, Germany
{avanes,freytag}@informatik.hu-berlin.de

***Abstract*—Distributed systems will be increasingly built on top of wireless networks, such as sensor networks or hand-held devices with advanced sensing and computational abilities. Supporting cooperative processes executed by such unreliable and dynamic system components poses a various number of new technical challenges. In terms of recovery, limited resource capabilities have be considered during re-scheduling of failed process activities. In terms of concurrency, a non-blocking protocol is required to allow a high degree of parallelism. In this paper, we introduce a *flexible* and *resource-oriented failure handling mechanism* for cooperative processes in hierarchical and distributed systems. The objective is to ensure both - transactional semantics as well as the selection of suitable nodes with respect to available resource capabilities. Based on a nested execution model, we develop a multi-stage algorithm that uses constraint solving techniques in a parallel fashion thus achieving a more efficient recovery. We evaluate our proposed techniques in a prototype implementation, and demonstrate significant performance gains by using a parallel re-scheduling.**

I. Introduction

Distributed information systems are increasingly built by commodity hardware and software, usually in form of clusters of workstations that are organized in hierarchies and connected via a network. Distributed processes in turn are sequences of computational steps executed over different clusters and platforms. *Business processes* are a well-known example. Corresponding instances of a business process are usually executed by *Workflow Management Systems* to accomplish important tasks within an organization or across different organizations. In recent years, the computing and resource capacities of small, embedded and wireless networking devices have matured sufficiently to enable the execution of real-time and cooperative processes on top of these devices. Hence, process supporting systems can be utilized in other application domains beyond business computing, such as disaster recovery, wilderness exploration and military operations. However, the integration of resource constrained devices poses new technical challenges for ensuring a robust process execution.

To this end we propose the development of a general-propose framework for supporting **coo**perative **p**rocesses in dynamic and **u**nreliable systems (CooPiuS). The CooPiuS framework allows the process designer to efficiently deploy and query processes in such challenging environments. Overall, we expect the CoopiuS framework to consist of three major components; see also Figure 1.

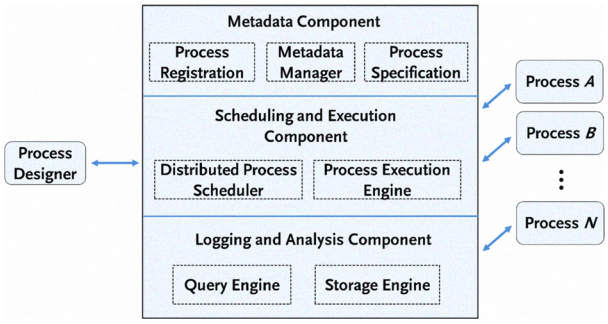

Fig. 1. Components of Our CooPiuS Framework

Rather on focusing on the *Metadata Component* which deals with metadata management, registration and publishing of new processes [1], [2], [3], [4], we investigate the *Scheduling and Execution Component* in this paper. While previous work of our research has been focused on the design of a heuristic, data-flow oriented scheduling at design-time [5], we study in this paper how to ensure an efficient recovery in the case of physical failures during run-time. Our objectives for building a robust *Execution Engine* are twofold: On the one side, the failure handling mechanism must ensure transactional semantics, i.e. serializability and atomicity. On the other side, it must select most suitable nodes with respect to available resource capabilities to guarantee a reliable re-execution.

A. Failure Recovery for Cooperative Processes

Due to the unreliable and dynamic execution environment, an appropriate recovery algorithm is required to guarantee correct process execution even in the case of unexpected physical failures, e.g. node failures, cluster crash, or temporary communication errors. Cooperative processes are not independent from each other, but they access shared, persistent data thus being subject to transactional semantics. Even within one process instance, several concurrent threads may access the same resource. Hence, regarding concurrency control, the control flow of cooperative processes is more complex than a flat transaction. Since locking of resources is not acceptable for long-running processes, the challenge is to find the right balance between allowed interaction and parallelism on the one side while still guaranteeing correctness on the other side. In terms of recovery, different alternatives can be chosen to

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8306
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8306

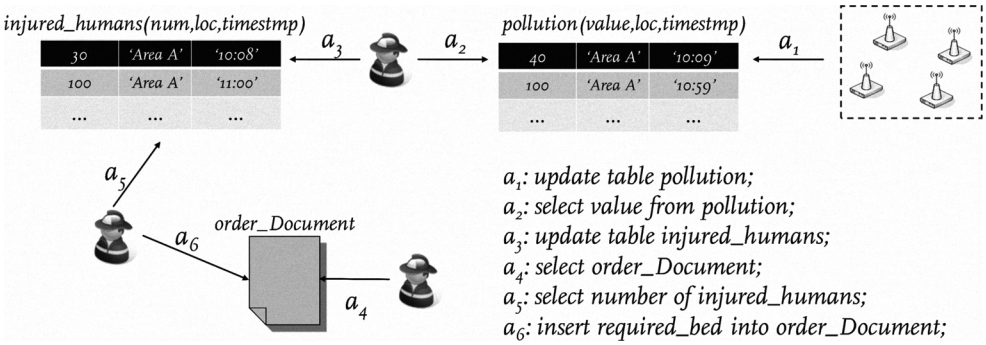

Fig. 2. Example of a Cooperative Process

re-execute failed activities. Here, the challenge during failure recovery is to find most suitable candidate devices with respect to current resource capabilities. Since resource capabilities are limited, those devices must be selected which have sufficient resources to re-execute the failed process activities. Furthermore, due to operational dependencies, the failure of one activity may force the compensation and re-execution of previous, successfully executed activities.

B. Motivating Examples

To illustrate the challenges above, consider following application scenario in the context of a disaster response (as depicted in Figure 2): During a disaster response, multiple wireless networks operate at the disaster location, such as groups of fire workers equipped with hand-held devices and sensor networks. They may execute location-based emergency activities and gather important data about the disaster. Following problems may appear during a concurrent execution of such response activities:

(a) Isolation Problem: Concurrent execution on the same data between two processes or even within one single process may lead to temporal data inconsistencies. Therefore, a concurrency control mechanism is needed to detect such conflicting executions.

Example 1.1 Suppose a group of firefighters counting the number of injured people in a certain area and organizing the subsequent ordering of required beds respectively. Now, consider the scenario where multiple activities $\langle a_3, a_4, a_5, a_6 \rangle$ *are executed concurrently, i.e. any concurrent execution order is possible. So, it may happen that one group member retrieves the number of injured people* (a_5) *to update the order document while another group member concurrently updates the number of injured people* (a_3). *It may also happen that a third firefighter accesses the order document* (a_4) *to forward it to its regional operation center before the current number about injured humans is inserted into the document* (a_6). □

(b) Atomicity Problem: Due to operational dependencies, recovery of failed activities in a process may affect activities executed previously in the same process as well activities running in other processes. Hence, a recovery mechanism is needed that aborts, compensates and eventually re-executes already completed or running activities.

Example 1.2 Suppose a sensor network that measures the pollution in a certain area where the group of firefighters operates (a_1). *The pollution values must be always retrieved before any emergency operation starts* (a_2). *If for some reasons the emergency activities executed by the group fail and must be re-executed by another group, the pollution value must be retrieved again before a re-assignment of failed emergency activities can be conducted. Hence, the emergency activities are not only re-executed but also previous executed activities are indirectly affected by the failure and must be executed again.* □

C. Contributions and Outline

This paper aims to develop an unified model for both - concurrency control and recovery applicable for hierarchical processes executed on top of wireless networks. In particular, we develop a failure handling mechanism that considers limited resource capabilities and selects optimal compensation nodes for re-execution of failed activities. In summary, the key contributions made in this paper are the following:

- A nested execution model is introduced that covers the hierarchical execution of cooperative process activities in distributed applications, such as the disaster recovery, more accurately than earlier research.
- We introduce correctness criteria for cooperative processes applicable on each level of the execution tree or across multiple execution trees.
- We develop an optimistic recovery mechanism that consists of the following steps: (a) building the failure scope graph, (b) generating the corresponding constraint graph, and (c) a physical re-scheduling (instantiation) of the constraint graph optimized with respect to available (possibly limited) resource capabilities.

The remainder of the paper is organized as follows: Section 2 discusses related work in the context of process recovery and compares different failure handling protocols. In Section 3, we introduce the system and process model and present our recovery algorithm in Section 4. Experimental results of a parallel re-scheduling using multiple constraint solvers are summarized in Section 5. We conclude and outline future research questions in Section 6.

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8306
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8306

II. Related Work

There is a plethora of research work on failure handling and recovery for processes. We have evaluated related research in following fields of computer science:

Database Systems: In the transaction theory, there are several nested transaction models [6][7]. They support failure recovery in different ways but they have all in common that there is a lack of flexibility required for long-running, hierarchical processes. Using our approach the process designer can decide how far recovery should be pursued in the process hierarchy. Additionally, our recovery algorithm identifies a most suitable compensation and re-execution with respect to limited resource capabilities. In [8] failure handling for transactional hierarchies are proposed based on a hierarchical, nested execution model. Our work can be seen as an extension of [8] whereas our research work introduces an unified model for both - recovery and concurrency. Additionally, we develop an optimistic forward recovery algorithm that finds suitable compensation and re-execution nodes with respect to available resource capabilities.

Workflow Management Systems: In the past, a series of commercial and research Workflow Management Systems (WFMSs) have been developed. In particular, the problem of recovery has been considered in a number of research projects. [9] introduces the notion of a sphere of joint compensation, which is a subset of a process' activities. If one activity of a sphere fails, the whole sphere has to be backed out (either by compensating each step that succeeded so far or by executing a special higher-level compensating activity). An extension of this work can be found in [10] where spheres of isolation are introduced for transactional workflows. Contrary to [9] we assume activities to be compensatable and retriable and so, we focus on identifying most suitable compensation and re-execution nodes to guarantee the continuation of failed activities. We deal with strict resource allocation constraints that must be considered during the recovery. In addition, we propose a hierarchical execution model where different recovery strategies can be performed.

[11] proposes a recovery mechanism in WFMSs by incorporating exception handling and transactional atomicity. The key idea is that process supporting systems, such as a WFMS, should be treated as a programming environment thus separating failure handling concepts from normal flow of control. While the incorporation of exception handlers with the concept of atomicity is an elegant way to combine backward and forward recovery, this framework does not support a flexible and efficient recovery for hierarchical processes in resource-constrained systems. In [12], an unified model for recovery and concurrency is proposed for transactional processes. Here, the goal is to design a middleware architecture that builds upon a strong theory regarding correct process execution. Again, our work can be seen as extension to [12] whereas we extend the process model and design an appropriate recovery algorithm that can be used in a flexible way. It is also important to note the differences between our notion of failure handling and the recent work on adaptive workflows and workflow evolution [13]. While our research is focused on finding an optimal set of nodes for compensation and re-execution of failed activities, they try to adapt the running workflow instance due to unexpected failures. These issues are beyond the scope of our work.

Mobile and Distributed Systems: In [14], workflows are modeled by task graphs which are embedded into a mobile ad-hoc network to discover suitable devices. If node failures appear, a re-instantiation of the task graph is performed with respect to node capabilities. [14] uses different metrics to find a good embedding, e.g. the average dilation that indicates the path length between communicating nodes. However, this approach does not consider any transactional semantic, such as concurrency or atomicity. [15] discusses transactional issues that appear when sensor networks are integrated into traditional business processes. In particular, a concurrency control mechanism is introduced that deals with continuous queries and conflicting updates. The key idea is that during the validation phase of the proposed optimistic protocol, update transactions have always a higher priority compared to so-called continuous query transactions. In [16], logging mechanism are investigated to improve execution and recovery performance. In terms of recovery, the idea is to save messages to the log in an application checkpoint. In a failure case, the component state can be recovered by going back to the last logged message and to continue from there. In this work, there are no resource allocation constraints and the recovery mechanism is very simple. However, recovery in a hierarchical process execution with many operational dependencies becomes much complicated.

III. System and Process Model

System Model. For this work, we assume the accomplishment of distributed processes, e.g. for disaster recovery, in a hierarchical fashion by different system layers. Thereby, each system layer is organized into cluster of devices or groups of humans respectively. Each cluster consists of several resources with a bounded capacity, e.g. pre-defined threshold for energy consumption or number of activities to be executed. On the top level, a global system component, referred to as *controller*, allows us to to maintain a consistent global view on the processes executed on lower system layers. For example, a national crisis squad may act as a controller during a disaster response. Networks on the bottom system levels execute *location-based process activities* and either passively forward their resulting data to higher layers or actively receive process directives from those. Examples for such local networks may be groups of firefighters operating at the disaster location. We also rely on *cluster services*, such as standard group communication software to manage membership as well other group functionalities [17].

Finally, we assume that there are one or more data containers on each system layer, e.g. a cluster of databases. Processes on the same system layer are allowed to concurrently access and to manipulate shared persistent data without waiting

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8306
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8306

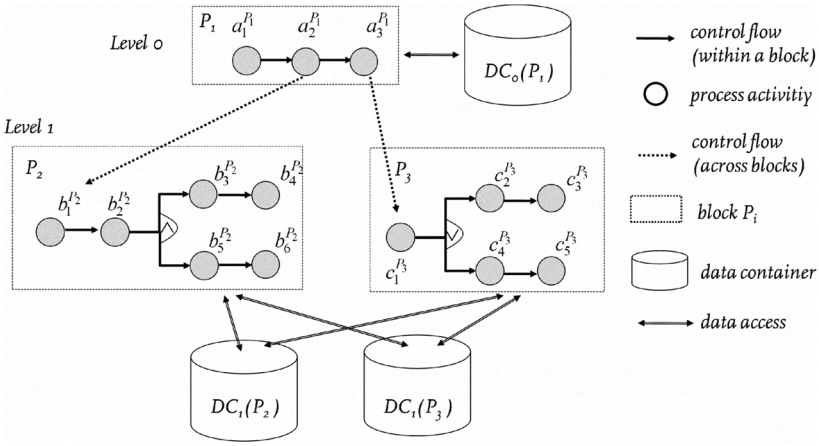

Fig. 3. Multi-Block Execution Tree $T(P) = \{P_1, P_2, P_3\}$

for a commitment. However, concurrent access of processes on higher system layers to data containers on lower layers is prohibited; instead data is exchanged through a request-response communication.

Process Model. In the following, we focus on the run-time *execution model* of a process that evolves during the execution of a concrete occurrence or instance of the corresponding *process specification model*. Thereby, a process instance P may contain any number of nested sub-processes $\{P_1, P_2, .., P_n\}$. P_1 represents the top level process that may trigger new sub-processes executed by lower system layers which in turn may initiate further sub-processes eventually forming a process tree. Sibling sub-processes may be executed sequentially or concurrently. While a sibling sub-process is running, its parent process may continue and spawn other child processes without any suspension. However, the failure of a child process may force the parent to a compensation and re-execution. Assume that the tree $T(P)$ in Figure 3 reflects a response process after a disaster. P_1 is the top-level process initiated by the controller, e.g. a national crisis squad. P_2 and P_3 are sibling sub-processes, whereas P_2 is the observed execution order of the process in Figure 2 (renaming a with b). We will use the execution tree $T(P)$ throughout the paper. Formally, such a nested process structure is defined as follows:

Definition 1: (Multi-Block Execution Tree). An execution tree of a process P, denoted as $T(P)$, consists of

(1) hierarchically organized blocks $T(P) = \langle P_1, ..., P_n \rangle$, where each block P_i with $i = 1..n$ forms a multi-activity node of the execution tree $T(P)$.
(2) control flow edges $\rightarrow \subseteq (P_i \times P_j)$ linking together blocks P_i,$P_j \in T(P)$ along the execution tree $T(P)$. □

Each block represents a sub-process P_i executed on one of the corresponding system layers. It consists of multiple (atomic) activities which are executed in a required order. A block can be formalized as follows:

Definition 2: (Block or Multi-Activity Tree Node). A block P_i of the execution tree $T(P)$, is a tuple $(\mathcal{A}_i, \prec_i)$ reflecting the execution of one sub-process where

(1) $\mathcal{A}_i = \{a_1^{P_i}, a_2^{P_i}, ..., a_j^{P_i}\} \cup \{C_i, A_i\}$ is the set of activities executed within a block P_i including designated *commit* C_i and *abort* activity A_i.
(2) $\prec_i \subseteq (\mathcal{A}_i \times \mathcal{A}_i)$ reflects the partial order between two activities in $\mathcal{A}_i$, referred to as required execution order. □

Blocks are linked with each other according to required control and data flow whereas conditional branching, parallel splitting and joining can be used as control structures within a block.

A. Commit Scope

Each block P_i of an execution tree has its own data container, e.g. a database. This way, several data containers are formed to a cluster on which sub-processes executed on the same system layer may concurrently access and store their results. The data container on the top-level reflects the process data container where all resulting data are gathered from lower levels. However, each block also has the opportunity either to store its result in its own data container or to make it visible to data containers belonging to other blocks on the same system level. Hence, a block of activities can terminate by

- committing to the block-root, i.e. checking-in its results to the data container of the next higher block, making its effects visible to the parent process, and by
- committing to block-neighbor, i.e. checking-in its results to the data container of the neighbor block on the same system level, and by
- committing to its own data container, i.e. checking-in its results to a database, making its effects visible for neighbor blocks (sibling sub-processes) on the same system level.

B. Process Executions

Additional to the regular activities observed during the execution, a correct execution may also contain activities required to compensate sub-processes or even the whole process itself. As a consequence, the execution characteristics of blocks are based on the notions of *compensatable* and *retriable* activities

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8306
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8306

[11], [18]. A compensatable activity is one that can be undone (its effects) in case the process fails. The compensating activity may be directly related to the failed activity (e.g., a transactional undo) or be a semantic compensation (e.g., a letter is sent notifying the user of a given mistake). Further on, retriable activites are the one that can be re-executed several times in the case of a failure. Either an alternative path is chosen or the same failed activity is re-executed again. Note that the execution characteristics are not mutually exclusive, i.e. an activity may be both compensatable and retriable or just retriable or compensatable. We introduce $\mathcal{A}_i^{-1} = \{(a_j^{P_i})^{-1}\}$ as the set that consists all activities required to compensate failed activities within a block P_i.

Figure 4 shows possible failures and re-executions that may appear in block P_2 (process ID omitted due to convenience reasons). The superscript c indicates a compensatable activity while r stands for retriable. In Figure 4, the standard execution is compared with possible failures of $b_1^{P_2}$ and $b_5^{P_2}$. Note that the compensation may not always consists of one - ideally the inverse activity (i). In our example here, there is another alternative to compensate failed activity $b_5^{P_2}$ (ii).

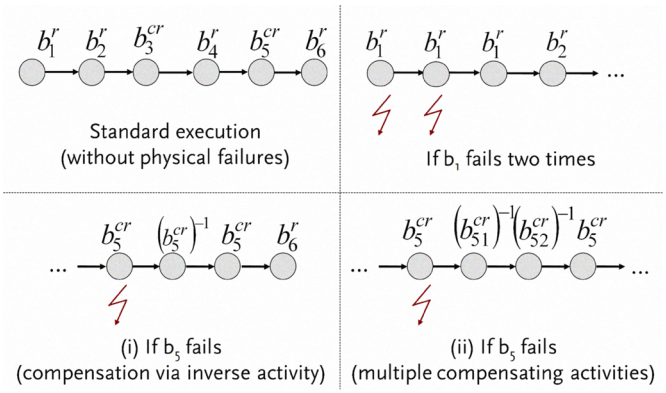

Fig. 4. Possible (Re)-Executions and Compensations

C. Constraints on Process Executions

So far, we have seen a block as a multi-activity node where its activities are executed in a required order. So, *ordering constraints* within a block (denoted as $\prec_i$) and ordering constraints along tree blocks (denoted as $\rightarrow$) are given as one constraint class to be fulfilled during the process executions. However, there are also *resource allocation constraints* as second constraint class that need to be considered during the execution [1], [5].

Example 3.1. (resource allocation constraint): if activity 1 is executed by *some resource*, *activity 2* should be executed by the *same or similar resource* (i.e. a resource within the same cluster or a certain distance), whereas the *total time* for the execution should not exceed a *certain threshold*. □

In general, resource allocation constraints can be subdivided into so-called *control constraints* and *cost constraints*. While control constraints refer to the control and data flow dependencies of executing tasks, cost constraints describe a certain cost function which has to be fulfilled during task execution. With respect to cost constraints, integrating devices or humans with a finite energy capacity requires an optimal selection of available resources, especially in the case of physical failures. So, let $\mathcal{C}_P = \{c_1, c_2, ..., c_m\}$ be the set that consists of all given resource allocation constraints for one process P where each $c_i \in \mathcal{C}_P$ may be either a control or a cost constraint.

D. Process Termination

Given the hierarchical structure of a process with guaranteed termination (either an abort or a commit), a process schedule reflects the concurrent execution of multiple tree blocks. So, the definition of a schedule is defined over a set of subprocesses, $\mathcal{P}_S$, which includes both - regular as well as compensation and recovery related activities. In summary, we formally define a process schedule as follows:

Definition 3: (Process Schedule). A process schedule S is a quadruple $(\mathcal{P}_S, \mathcal{A}_S, \prec_S, \mathcal{C}_S)$ where

(1) $\mathcal{P}_S$ is the set of (sub-)processes or blocks respectively observed during the sequential or concurrent execution.
(2) $\mathcal{A}_S$ is the set of all executed block activities.
(3) $\prec_S \subseteq (\mathcal{A}_S \times \mathcal{A}_S)$ reflects the partial order between two activities in $\mathcal{A}_S$, referred to as observed execution order.
(4) $\mathcal{C}_S$ is the set of all given resource allocation constraints to be considered during the execution. □

This definition includes partial as well as complete schedules. We also consider the set of given resource allocation constraints in the definition. In the case of failures where rescheduling of activities is required, these constraints must be considered to guarantee a correct process schedule.

IV. Failure Handling

In this section, we propose our failure handling mechanism based on the system and execution model. In particular, we are interested in a recovery algorithm that finds a new correct process schedule that optimizes given resource allocation constraints. Thereby, we assume that the process activities are both - *compensatable* and *retriable*, i.e. it is guaranteed that failed activities eventually commit after a finite sequence of invocations or by choosing alternative paths. Therefore, our failure handling belongs to the class of *optimistic forward recovery* algorithm.

A. General Failure Handling Steps

In general, the recovery algorithm consists of the following three major steps:

1.Determining the failure scope. We first need to select all activities that are affected by failures and conflicts. Hence, in a first step, the failure scope graph $F_{SG} := \boldsymbol{\sigma}(T_1, ..., T_m)$ is determined for a given set of execution trees $\{T_1, ..., T_m\}$ using the operator $\boldsymbol{\sigma}$.

2.Constructing the constraint graph. In a second step, the corresponding *constraint graph* $C_G := \boldsymbol{\mu}(F_{SG})$ is built using the construction operator $\boldsymbol{\mu}$. The constraint graph is an extension of the failure scope graph F_{SG} reflecting given resource allocation constraints associated with F_{SG}.

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8306
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8306

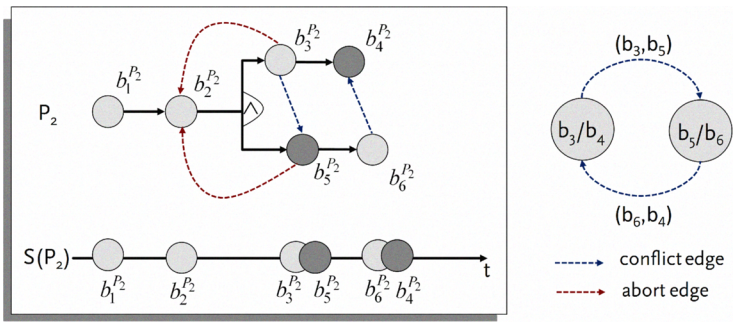

Fig. 5. Process Schedule $S(P_2)$ with corresponding Serialization Graph

3.Re-Scheduling. After the failure scope graph is created, a re-scheduling of the failure scope graph F_{SG} to distributed resource cluster $\{G_1, ..., G_k\}$ is executed by instantiating its corresponding constraint graph C_G. For this purpose, the mapping operator $\phi_{mapp}(C_G)$ selects from the set of possible candidate mappings an optimized mapping with respect to $\mathcal{C}_S$.

Note that multiple failure scope and constraint graphs may exist due to possible alternatives that can be executed in a failure case. The goal is to find a best mapping for a constraint graph that optimally fulfills given resource constraints. Moreover, since failures and conflicting activities may also appear during the re-execution phase, the recovery algorithm may be invoked several times until a correct process schedule is found. In the following, we will distinguish between failure handling inside a block and across multiple blocks respectively.

B. Failure Handling Inside a Block

Failure Handling inside a block or in a flat sub-process respectively is based on *conflicting activities* and *abort dependencies* between different activities inside a block.

Conflicting Activities. Two activities $(a_i^{P_j}, a_{i+1}^{P_j})$ are in *conflict* if they do not *commute*. Let $< v a_i^{P_j} a_{i+1}^{P_j} \omega >$ describe a prefix of activities v executed before activity pair $< a_i^{P_j} a_{i+1}^{P_j} >$ and a suffix of activities ω executed after these two activities. Then, two activities $< a_i^{P_j} a_{i+1}^{P_j} >$ are in conflict if following holds:

$$< v a_i^{P_j} a_{i+1}^{P_j} \omega > \neq < v a_{i+1}^{P_j} a_i^{P_j} \omega > \quad \textbf{(1)}$$

The resulting effects of two activities $(a_i^{P_j}, a_{i+1}^{P_j})$ are different for different invocation orders of these activities, i.e. commutativity between $(a_i^{P_j}, a_{i+1}^{P_j})$ is not given.

Abort-Dependent Activities. Two activities $(a_k^{P_j}, a_{k+1}^{P_j})$ may have abort dependencies, i.e. if activity $a_{k+1}^{P_j}$ fails then activity $a_k^{P_j}$ previously executed must be also compensated and re-executed. In other words, if $a_{k+1}^{P_j}$ succeeds, then $a_k^{P_j}$ can succeed. Formally, we can specify an abort or failure dependency as follows:

$$a_{k+1}^{P_j} \rightarrow_{fail} a_k^{P_j} \quad \textbf{(2)}$$

While conflict pairs are detected by the execution engine during run-time, abort-dependencies are marked by the process designer at design-time. Based on (1) and (2), we can now define correctness criteria for a single block P_j within an execution tree $T(P)$.

Definition 4: (Block Serializablity (B-SR)). A block or sub-process P_j respectively is serializable, if its committed and active projection $CA(P_j)$ is conflict-equivalent to a serial process schedule $S(P_j)$. □

Several concurrent threads, e.g. due to an AND-branch (see also Figure 5), may access on shared data causing conflicts as defined above. Hence, a block P_j is serializable if there is a conflict-equivalent serial execution of concurrent threads in P_j, i.e. the resulting effects are the same as the (observed) concurrent thread execution. Formally, block serializability implies that the corresponding serialization graph for block P_j is acyclic.

Definition 5: (Block Recoverability (B-RC)). A block or sub-process P_j respectively is recoverable, if for each failed activity the set of abort-dependent activities $A(P_j)$ is compensated and re-executed successfully including the failed activity itself. □

As shown in Figure 5, activity $b_2^{P_2}$ is abort-dependent on activities $\left\langle b_3^{P_2}, b_5^{P_2} \right\rangle$. If either of activities $\left\langle b_3^{P_2}, b_5^{P_2} \right\rangle$ fails, already completed activity $b_2^{P_2}$ must be re-executed.

In the following, we describe the recovery steps for handling failures within one single block P_j which consider conflicting, aborting and abort-dependent activities.

1.Step (σ-operator): First, we determine the failure scope. Let the set $CP(P_j) = \{cp_1(a_k^{P_j}, a_l^{P_j}), ..., cp_n(a_q^{P_j}, a_r^{P_j})\}$ describe all conflict pairs cp_i within block P_j, i.e. the nodes contained in the respective cyclic serialization graph of P_j.

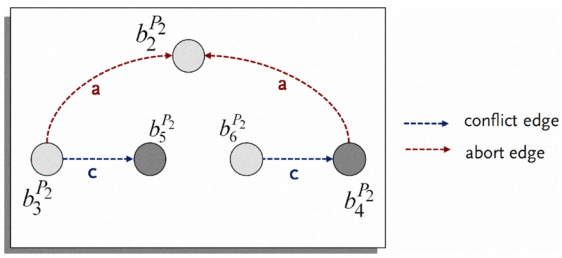

Fig. 6. Failure Scope Graph $F_{SG}(P_2)$ for Block P_2

Further on, assume that k activities in P_j fail and must be aborted. Then, $\bigcup_{i=1}^{k} \mathcal{A}_i(P_j)$ is the set of all abort-dependent

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8306
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8306

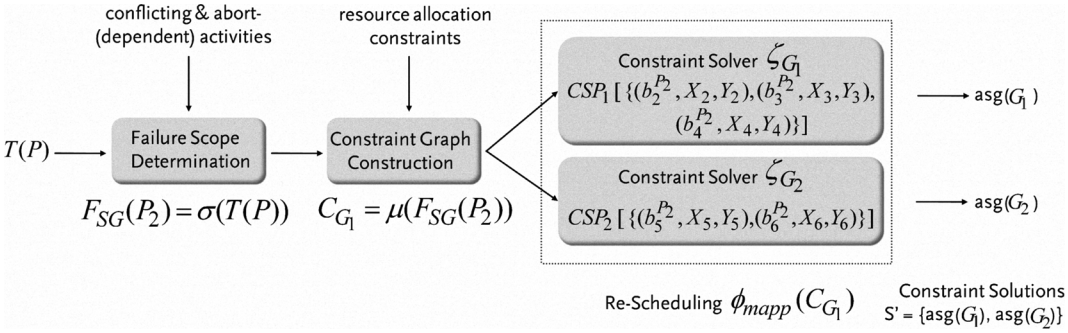

Fig. 7. Recovery Steps - Overview

activities including the k aborted activities. The failure scope graph $F_{SG} = (V, E)$ is formally constructed as a set of nodes V and edges E with

- $V := CP(P_j) \cup \bigcup_{i=1}^{k} \mathcal{A}_i(P_j)$
- $E := E_c$ *(conflict edges)* $\cup\ E_a$ *(abort edges)*

Example 4.1. In Figure 6, the failure scope graph is depicted for our running example P_2. The block P_2 consists of two conflict pairs $cp_1(b_3^{P_2}, b_5^{P_2})$ and $cp_2(b_6^{P_2}, b_4^{P_2})$ and there are two abort dependencies between $(b_2^{P_2}, b_3^{P_2})$ and $(b_2^{P_2}, b_5^{P_2})$ respectively. □

2.Step (μ-operator): Given the failure scope graph F_{SG}, we derive the corresponding constraint graph C_G from F_{SG} by annotating the nodes of F_{SG} with a *control variable* X_i and *cost variable* Y_i. These variables represent possible control and cost constraints for each activity a_i that must be considered during the re-scheduling step. The edges between the nodes of C_G reflects constraints linking participating nodes together - also referred to as *constraint scope*.

Example 4.2. In Figure 8, C_{G_1} represents one possible constraint graph for $F_{SG}(P_2)$. Assume the presence of several resource allocation constraints between activities, such as $c_1 \in \mathcal{C}_S : \left\langle b_2^{P_2}, b_3^{P_2}, b_4^{P_2} \right\rangle$ *must be executed by cluster* G_1 and $c_2 \in \mathcal{C}_S : \left\langle b_5^{P_2}, b_6^{P_2} \right\rangle$ *should be executed by cluster* G_2.

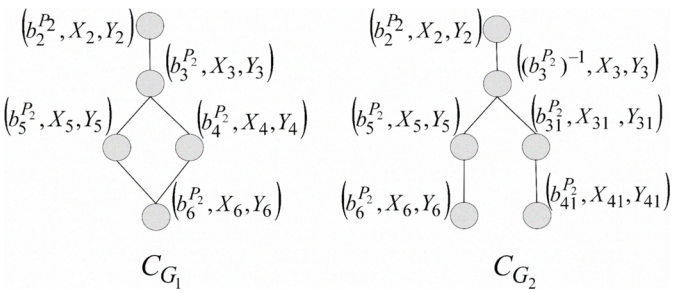

Fig. 8. Different Constraint Graphs for $F_{SG}(P_2)$

3.Step (ϕ-operator): After the determination of the failure scope, we are now able to re-schedule the activities of F_{SG} by instantiating its constraint graph. The objective is to identify physical nodes with sufficient resource capabilities to carry over failed activities.

For this purpose, we select from the set of possible instantiations for the constraint graph the best instantiation with respect to given resource constraints. Re-scheduling can be formally described as a *mapping* or embedding process of a constraint graph to a network graph of k strong components $\{G_1, ..., G_k\}$ where each component corresponds to a cluster of resources:

$$\forall C_{G_i} : \phi_{mapp}(C_{G_i}) \rightarrow \{G_1, ..., G_k\}.$$

Constraint programming has proven its feasibility in solving real-world scheduling and planning problems where various complex constraints must be considered [19], [20]. Therefore, we translate the re-scheduling problem into a constraint satisfaction problem (CSP) as follows:

Definition 6: (Re-Scheduling as a CSP). The mapping operator ϕ_{mapp} gets a triple $\left\langle \{(a_i^{P_j}, \mathcal{X}_i, \mathcal{Y}_i)\}, \{G_1, ..., G_k\}, \mathcal{C}_S \right\rangle$ as input where

- $V_\phi = \{(a_i^{P_j}, \mathcal{X}_i, \mathcal{Y}_i) | a_i^{P_j} \in V\}$ is the set of **variables** where $\mathcal{X}_i$ is the *control variable* and $\mathcal{Y}_i$ is the corresponding *cost variable*.
- $D_\phi = \{G_1, ..., G_k\}$ is the corresponding **domain**.
- $\mathcal{C}_\phi \subseteq \mathcal{C}_S$ is the set of **constraints**. Each constraint $c_i \in \mathcal{C}_\phi$ is a pair $\langle \alpha, \rho \rangle$ where α is a list of distinct variables from V_ϕ and ρ is a $|\alpha|$-ary relation over D_ϕ. □

V_ϕ is assembled by conflicting and aborting activities. It also includes all abort-dependent activities as well as the necessary compensation steps. Each of these activities is assigned to a control and cost variable thus building a triple of the form $(a_i^{P_j}, \mathcal{X}_i, \mathcal{Y}_i)$ or $((a_i^{P_j})^{-1}, \mathcal{X}_i, \mathcal{Y}_i)$ respectively. The value for the control variable indicates the resource which the activity will be re-executed on while the value for the cost variable must fulfill an individual cost functions.

A **solution** is a mapping $\phi_{mapp} : V_\phi \rightarrow D_\phi$ such that, for each $\langle \alpha, \rho \rangle \in \mathcal{C}_\phi$, $\phi(\alpha) \in \rho$. Informally spoken, a solution is an assignment of resources (chosen from D_ϕ) to the control variables of the corresponding activities to be re-scheduled. The value for the cost variable depends on the chosen value for the control variable. The assignment should fulfill control as well as cost constraints. Consequently, a **best solution** is a dedicated solution from the set of possible solutions that optimizes a given objective function. For our purpose, a best solution fulfills control constraints whereas cost constraints are

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8306
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8306

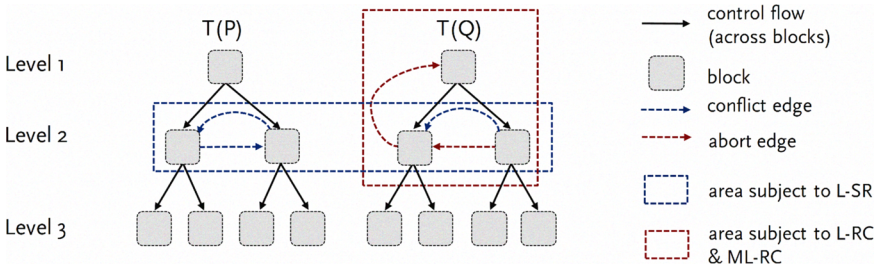

Fig. 9. Conflicts and Abort Dependencies Across Multiple Blocks

optimized, e.g. minimized for each individual cluster G_i. To achieve an efficient re-scheduling, we parallelize the mapping process of a single constraint graph by utilizing *multiple constraint solvers*. Thereby, each cluster has its own constraint solver acting as a local scheduler only having knowledge about its cluster members. Each constraint solver, denoted as ζ_{G_i}, only re-calculates a subset of the constraint graph, i.e. it takes those activities $a_i^{P_j} \in V_\phi$ as input which have been executed by resources within its cluster G_i. This way, the re-scheduling task is divided into several smaller *CSPs* which are solved in parallel. In addition, the constraint solver will use a reduced domain for its re-calculation. It excludes resources being currently down and not available. After all, we achieve a fast re-scheduling by allowing a distributed and parallel constraint solving with reduced domains. The question how to initially schedule activities to blank resources at design time is out of scope in this paper, instead we refer to [5] for the interested reader.

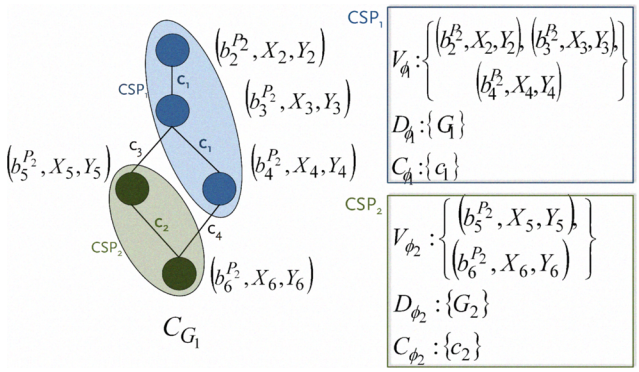

Fig. 10. Parallel Constraint Solving for C_{G_1}

Example 4.3. Continuing Example 4.2 with constraint graph C_{G_1}, assume the presence of two clusters (G_1, G_2), whereas activities $\langle b_2^{P_2}, b_3^{P_2}, b_4^{P_2} \rangle$ must be re-executed by resources of G_1 (control constraint c_1) and activities $\langle b_5^{P_2}, b_6^{P_2} \rangle$ must be re-executed by resources of G_2 (control constraint c_2). Control constraints $\langle c_3, c_4 \rangle$ are the negation of $\langle c_1, c_2 \rangle$ forcing an execution in different clusters. Then, we can build two constraint satisfaction problems CSP_1 and CSP_2, where each cluster acts as a local scheduler solving its own CSP as shown in Figure 10. □

C. Failure Handling Across Multiple Blocks

So far, we have just considered concurrency control and failure handling for one single block P_j of an execution tree $T(P)$. In practice however, multiple blocks may be executed concurrently at the same level. Hence, conflicts may also appear between these so-called *sibling blocks*. Additionally, abort dependencies will not be only limited to activities within one block but may also affect parent or sibling blocks. Therefore, we enhance our correctness criteria introduced in the previous subsection to allow failure handling across multiple blocks. *B-SR* and *B-RC* are necessary but not sufficient conditions for correct process execution across multiple blocks. Due to conflicts between sibling blocks, the concurrent execution may not be serializable even if each participating block fulfills *B-SR*.

Definition 7: (Level Serializability (L-SR)). Multiple blocks $< P_i, Q_i, ..., Z_i >$ on level i are level-serializable, if the committed and active projection $CA(P_i, Q_i, ..., Z_i)$ is conflict-equivalent to a serial process schedule. □

Accordingly, *B-RC* is not enough to guarantee recoverability across sibling blocks or along the execution trees. Instead, one failure in a single block may affect other sibling or parent blocks creating undesirable long *abort chains*.

Definition 8: (Level (L-RC) and Multi-Level Recoverability (ML-RC)). Multiple blocks $\langle P_i, Q_i, ..., Z_i \rangle$ on level i are level-recoverable, if for each failed activity the sets of abort-dependent activities $\mathcal{A}(P_i), \mathcal{A}(Q_i), ..., \mathcal{A}(Z_i)$ are compensated and re-executed successfully including the failed activity itself. If compensation and re-execution for abort-dependent activities along the affected execution tree is also guaranteed, then the corresponding process is multi-level recoverable. □

To ensure *L-SR* and *L-RC*, the same recovery steps take place as in the case of *B-SR* and *B-RC* respectively. The only difference is that all conflict pairs and abort-dependent activities *spawn over multiple blocks* on the same level are collected to build the corresponding failure scope graph (**1.Step**). Then the corresponding constraint graphs are constructed (**2.Step**) before the best instantiation with respect to available resource capabilities is chosen. (**3.Step**).

Deferred Recovery and Partial Commit. However, compensating and re-executing abort-dependent activities along an execution tree to ensure *ML-RC* is much more challenging. In particular, we have to deal with the problem of *abort chains* or

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8306
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8306

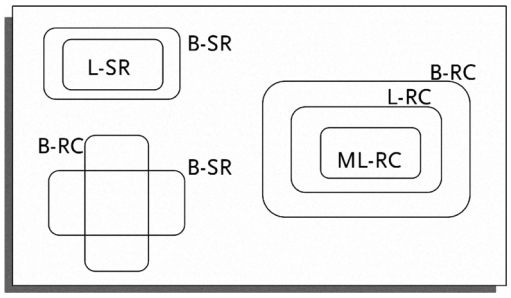

Fig. 11. Relations between B-SR,B-RC,L-SR,L-RC, and ML-RC

cascading aborts respectively. Even if it is assumed that most process activities are compensatable and retriable, a recovery along execution trees may negatively affect the interaction between sibling blocks. Important results of some process activities not affected by the failure may be hold back (data is not visible) until failed and abort-dependent activities are compensated and re-executed successfully. Meanwhile other processes may wait for results to continue their execution thus leading to an unnecessary blocking state.

The key idea to avoid long blocking phases is to mark sections within a block as *optional* before the execution starts, e.g. one specific path within a branch in a block. The process designer marks activities as optional if their results are not needed for further processing of other concurrent processes. This way, we are able to speed up the recovery by excluding these optional activities from the failure scope. In other words, we allow a *partial commit* where effects of non-optional activities are made visible for other sub-processes. Failed activities within an optional section may be re-scheduled later; meanwhile results of those activities are treated as invalid. Formally, we adapt our failure handling mechanism as follows:

1.Step(σ-operator): Again, first all conflict pairs belonging to the corresponding cyclic serialization graph are inserted into the failure scope graph F_{SG}. To reduce the size F_{SG} and to minimize the time effort needed for compensation and re-execution, only failed and abort-dependent activities are included in F_{SG} that are not part of an optional block section.

2.Step(μ-operator): By reducing the size of the failure scope graph, we also reduce the number of variables of its corresponding constraint graph. Also, possible constraints are excluded thus eliminating edges in the constraint graph.

3.Step(ϕ-operator): The mapping or instantiation process respectively may now affect less constraint solvers due to the exclusion of variables. Since the variable, constraint and domain set is reduced, the overall solving procedure will speed up.

Example 4.3. In Figure 12, an additional block P_4 is added that has an abort-dependency with its parent block P_3, i.e. the abort-dependency $d_1^{P_4} \rightarrow_{fail} c_5^{P_3}$ exists. Given the observed schedule and abort-dependencies, assume now that $d_3^{P_4}$ fails forcing a compensation and re-execution of all other activities in block P_4 as well as the path activities $\left\langle c_4^{P_3}, c_5^{P_3} \right\rangle$ of block P_3. However, $\left\langle c_4^{P_3}, c_5^{P_3} \right\rangle$ belong to an optional path section (depicted within dotted rectangle) and therefore our recovery algorithm will exclude those from the failure scope. □

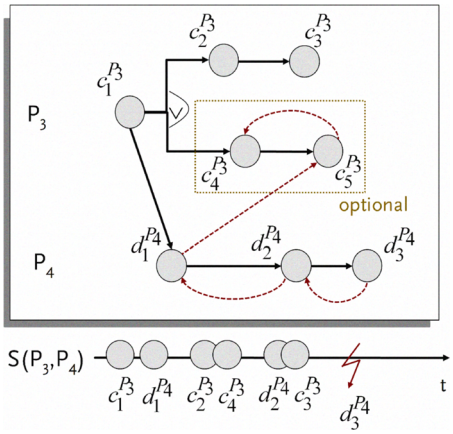

Fig. 12. Optional Block Section in P_3

V. Implementation and Evaluation Results

We have implemented and tested our framework in a Java-based prototype using a PC with 2GHz (dual core) and 4GB RAM. Our prototype uses multiple instances of an off-the-self constraint solver *Choco* [21] to solve constraint graphs of different sizes in a parallel fashion. Choco itself is a Java-based library for constraint satisfaction problems, constraint programming and explanation-based constraint solving. It supports a variety of constraints including basic constraints on *Integer Variables*, such as eq, neq, leq, geq and allows the user to define different search strategies.

Activities and resources are described as database tuples in a MySQL database each containing additional information, such as control/data flow dependencies (in case of activity tuples) and ID, current consumption, as well as cluster affiliation (in case of resource tuples). We also store the status of an activity, such as *ready, running, failed, committed, compensated*. Control and cost constraints are modeled as respective Choco constraints. As soon as the activity status changes to *failed*, our recovery algorithm implemented in Java identifies the failure scope, builds the corresponding constraint graph(s) and performs the allocation by starting multiple *Choco* threads.

As metric we have measured the *re-instantiation time* needed to map corresponding constraint graphs to the network graph. For our experiments, we considered the *number of activities* a, the *number of resources* r and the number of *constraint solvers* s as configurable parameters.

The summary of our findings are the following:

- For the re-instantiation time, the number of constraint solver participating during the re-scheduling step are more crucial than the number of activities to be re-scheduled. The use of more constraint solvers reduces the domain size to be traversed through by each single constraint solver thus improving the overall solving time.

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8306
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8306

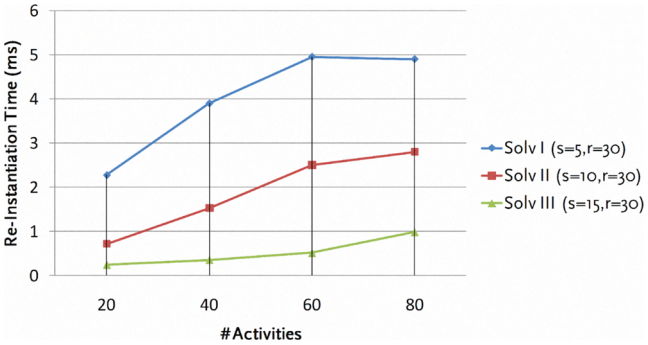

Fig. 13. Parallel Constraint Solving

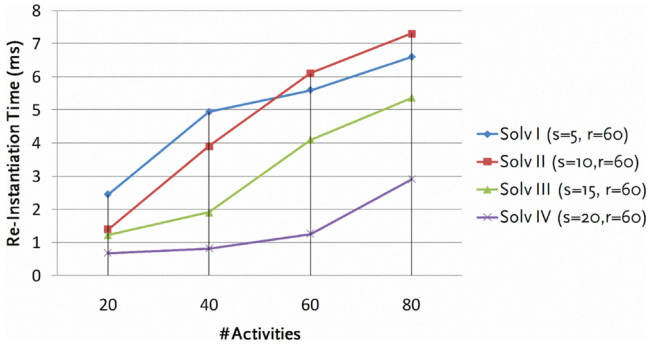

Fig. 14. Higher Number of Resources (Larger Clusters)

- The cluster size also greatly influences the re-instantiation time. On the one side, a bigger cluster may require more calculation time due to more variable assignments to be checked. On the other side, it may satisfy control and cost constraints more likely than smaller clusters. Hence, it must be always traded off between cluster size and constraint satisfaction.

VI. Conclusion and Future Work

In this paper, we have explored the issues in supporting a flexible failure handling for cooperative processes on top of distributed and resource constrained networks. In particular, we have focused on an efficient re-scheduling procedure that finds an optimal schedule with respect to limited resource capabilities in a distributed fashion. We allow a flexible failure handling for hierarchical processes by compensating and re-executing either just a block of the execution tree or along multiple blocks and levels of the execution tree. Based on our prototype implementation, we have shown that the number of participating constraint solvers has a bigger impact on the re-instantiation time than the number of activities.

Future work will include the development of a query engine part of our *Logging and Analysis Component* that can be used to efficiently retrieve huge amount of distributed process data through a SQL-like interface. Thereby, the query engine must cope with physical failures and resource constraints.

Acknowledgments.

This research is supported by the German Research Society (DFG grant no. NA 1324).

References

[1] P. Senkul, M. Kifer, and I. H. Toroslu, "A Logical Framework for Scheduling Workflows Under Resource Allocation Constraints," in *VLDB*, 2002, pp. 694–705.

[2] M. Reichert and P. Dadam, "Adept$_{flex}$-Supporting Dynamic Changes of Workflows Without Losing Control," *J. Intell. Inf. Syst.*, vol. 10, no. 2, pp. 93–129, 1998.

[3] F. Leymann and D. Roller, *Production Workflow: Concepts and Techniques*. Prentice Hall PTR, 2000.

[4] M. Gillmann, G. Weikum, and W. Wonner, "Workflow Management with Service Quality Guarantees," in *SIGMOD*, 2002, pp. 228–239.

[5] A. Avanes and J.-C. Freytag, "Adaptive Workflow Scheduling Under Resource Allocation Constraints and Network Dynamics," in *VLDB*, 2008, pp. 1631–1637.

[6] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem, "Modeling Long-Running Activities as Nested Sagas," *Data Eng.*, vol. 14, no. 1, pp. 14–18, 1991.

[7] A. K. Elmagarmid, Ed., *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992.

[8] Q. Chen and U. Dayal, "Failure Handling for Transaction Hierarchies," in *ICDE*, 1997, pp. 245–254.

[9] F. Leymann, "Supporting Business Transactions Via Partial Backward Recovery In Workflow Management Systems," in *BTW*, 1995, pp. 51–70.

[10] A. Guabtni, F. Charoy, and C. Godart, "Spheres of Isolation: Adaptation of Isolation Levels to Transactional Workflow," in *Business Process Management*, 2005, pp. 458–463.

[11] C. Hagen and G. Alonso, "Exception Handling in Workflow Management Systems," *IEEE Transactions on Software Engineering*, vol. 26, no. 10, pp. 943–958, 2000.

[12] H. Schuldt, G. Alonso, C. Beeri, and H.-J. Schek, "Atomicity and Isolation for Transactional Processes," *ACM Transactions on. Database Systems*, vol. 27, no. 1, 2002.

[13] A. Bernstein, C. Dellarocas, and M. Klein, "Towards Adaptive Workflow Systems: CSCW-98 Workshop Report," *SIGGROUP Bull.*, vol. 20, no. 2, pp. 54–56, 1999.

[14] P. Basu, W. Ke, and T. D. C. Little, "Dynamic Task-Based Anycasting in Mobile Ad Hoc Networks," *Mob. Netw. Appl.*, vol. 8, no. 5, pp. 593–612, 2003.

[15] L. Gürgen, C. Roncancio, C. Labbé, and V. Olive, "Transactional Issues in Sensor Data Management," in *VLDB Workshop DMSN*, 2006, pp. 27–32.

[16] R. Barga, S. Chen, and D. Lomet, "Improving Logging and Recovery Performance in Phoenix/App," in *ICDE*, 2004, p. 486.

[17] W. Vogels, D. Dumitriu, K. Birman, R. Gamache, M. Massa, R. Short, J. Vert, J. Barrera, and J. Gray, "The Design and Architecture of the Microsoft Cluster Service - A Practical Approach to High-Availability and Scalability," in *FTCS*, 1998, pp. 422–431.

[18] A. K. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz, "A Multidatabase Transaction Model for Interbase," in *VLDB*, 1990, pp. 507–518.

[19] P. Chan, K. Heus, and G.Weil, "Nurse Scheduling with Global Constraints in Chip: GYMNASTE," in *PACT*, 1998.

[20] K. Chow and M. Perett, "Airport Counter Allocation using Constraint Logic Programming," in *PACT*, 1997.

[21] F. Laburthe, "Choco: Implementing a CP Kernel," in *TRICS*, 2000, pp. 71–85.

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8306
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8306