Óbuda University

PhD thesis

Dynamic Execution of Scientific Workflows

by

Eszter Kail

Supervisors:

Miklós Kozlovszky

Péter Kacsuk

Applied Informatics Doctoral School

Budapest, 2016

Statement

I, Eszter Kail, hereby declare that I have written this PhD thesis myself, and have only used sources that have been explicitly cited herein. Every part that has been borrowed from external sources (either verbatim, or reworded but with essentially the same content) is unambiguously denoted as such, with a reference to the original source.

Contents

List of Figures

List of Tables

Abstract

With the increase in computational capacity more and more scientific experiments are conducted on parallel and distributed computing infrastructures. These in silico experiments, represented with scientific workflows, are long-running and often time constrained computations. To successfully terminate them within soft or hard deadlines dynamic execution environment is indispensable.

The first and second thesis group deals with the topic of one of the main aspect of dynamism, namely fault tolerance. This issue is long standing in focus due to the increasing number of in silico experiments, and the number of faults that can cause the workflow to fail or to successfully terminate only after the deadline.

In the first thesis group I have investigated this topic from a workflow structure perspective. Within this thesis group I have introduced the influenced zone of a failure concerning the workflow model, and based on this concept I have formulated the sensitivity index of a scientific workflow. According to this index I gave a classification of scientific workflow models.

In the second thesis group based on the results obtained from the first thesis group I have introduced a novel (Wsb) checkpointing algorithm, which can reduce the overhead of the checkpointing, compared to a method that was optimized concerning the execution time, without negatively affecting the total wallclock time of the workflow. I have also showed that this algorithm can be effectively used in a dynamically changing environment.

The third thesis group also considers a problem on a recently emerged topic: it investigates the possibility and requirements of provenance based adaptive execution and user-steering. In this thesis group I have introduced special control points (iPoints), where the system or the user can take over the control and based on provenance information the execution may deviate from the workflow model. I have specified these iPoints in IWIR which was targeted to promote interoperability between existing workflow representations.

Absztrakt

A számítási kapacitás növekedésével egyre több tudományos kisérlet végrehajtása történik párhuzamos és elosztott számítási erőforrásokon. Ezek az úgynevezett in silico kísérletek általában hosszú, de eredményeik érvényességét tekintve időben korlátozott futásidejű számítások. Tekintettel a komplex erőforrásokra és a gyakori, valamint széleskörű hibákra a határidőn belüli sikeres lefutás érdekében a dinamikus futási környezet biztosítása nélkülözhetetlen.

Az első és második téziscsoport a dinamizmus egyik fő területével, a hibatűrő mechanizmusokkal foglakozik. Ez a problémakör hosszú ideje a kutatások középpontjában áll köszönhetően az in silico kisérletek egyre szélesebb körű elterjedésének, valamint a gyakori és változatos hibák okozta sikertelen vagy határidőn túl befejeződő munkafolyamat futtatásoknak.

Az első téziscsoport a problémát a munkafolyamatokat leíró gráfok struktúrája felől vizsgálja. Bevezettem egy hiba hatáskörének fogalmát, majd a fogalomra alapozva kidolgoztam a munkafolyamatra jellemző, érzékenységi indexet. Az index értékei alapján osztályoztam a különböző munkafolyamat gráfokat.

A második téziscsoportban az első téziscsoport eredményeire támaszkodva kidolgoztam egy statikus (Wsb) ellenőrzőpont algoritmust, mely a futási időre optimalizált algoritmushoz képest csökkenti az ellenőrzőpontok készítésének költségét, anélkül, hogy a lefutási időt megnövelné. Munkám során megmutattam, hogy az algoritmus dinamikusan változó környezetben is hatékonyan működik.

A harmadik téziscsoport egy, az utóbbi időben jelentőssé vált problémával foglalkozik. A provenance alapú adaptív futás, illetve a felhasználó általi vezérlés lehetőségét és követelményeit vizsgálja. A téziscsoport keretein belül olyan vezérlési pontokat (iPoint) dolgoztam ki, ahol az irányítást átveheti a rendszer vagy a felhasználó és a provenance adatbázisban tátolt adatok alapján megváltoztathatja tervezett futását. A vezérlési pontokat egy munkafolyamat leíró köztes nyelven (IWIR) specifikáltam.

List of abbreviations

Abbreviation	Meaning
SWf	Scientific Workflow
SWfMS	Scientific Workflow Management System
HPC	High Performance Computing Infrastructures
OPM	Open Provenance Model
DA	Designator Action
XA	eXecutable Action
DAG	Directed Acyclic Graph
PD	Provenance Database
DFS	Depth-First Search
IWIR	Interoperable Workflow Intermediate Representation
W3C	World Wide Web Consortium
Wsb	Workflow structure based
AWsb	Adaptive Workflow structure based
RBE	Rule Based Engine
iPoint	intervention Point
SLA	Service-level Agreement
VM	Virtual Machine
WFLC	Workflow Life Cycle

1 Introduction

The increase of the computational capacity and also the widespread usage of computation as a service enabled complex scientific experiments conducted in laboratories to be transformed to in silico experiments executed on local and remote resources. In general these in silico experiments aim to test a hypothesis, to derive a summary, to search for patterns or simply to analyze the mutual effects of the various conditions. Scientific workflows are widely accepted tools in almost every research field (physics, astronomy, biology, earthquake science, etc.) to describe and to simplify the abstraction and to orchestrate the execution of these complex scientific experiments.

A scientific workflow is composed of computational steps that are executed in sequential order or parallel wise determined by some kind of dependency factors. We call these computational steps tasks or jobs, which can be data intensive and complex computations. A task may have input and output ports where the input ports consume data and the output ports produce data. Data produced by an output port is forwarded through outgoing edges to the input ports of subsequent tasks. Mostly we differentiate data flow or control flow oriented scientific workflows. While in the former one the data dependency determines the real execution path of the individual computational steps and data movement path, in the latter one there is an explicit task or job precedence defined.

Scientific workflows are in general data and compute intensive thus they usually require parallel and distributed High Performance Computing Infrastructures (HPC), such as clusters, grids, supercomputers and clouds to be executed. These infrastructures consist of numerous and heterogeneous resources. To hide the complexity of the underlying low-level, heterogeneous architecture Scientific Workfow Managements Systems (SWfMS) have emerged in the past two decays. SWfMs tend to manage the execution-specific hardware types, technologies and protocols whilst providing user-friendly, convenient interfaces to the various user types with different knowledge about the technical details. However, this user-friendly management system hides a complex thus, an error prone architecture, and a continuously changing environment for workflow execution.

As a consequence, when the environment is changing continuously, then a dynamically changing or adapting execution model should be provided. It means that the Scientific

Workflow Management System should provide means to adapt to the new environmental conditions, to recover from failures, to provide alternative executions and to guarantee successful termination of the workflow instances with a probability of p and lastly, but not finally to enable optimization support according to various needs such as time and energy usage.

We differentiated three different aspects of dynamism: Fault tolerance, which is the ability to continue the execution of the workflow in the presence of failures; Optimization, which enables optimized executions according to given parameters (i.e.: cost, time, resource usage, power,...); and Intervention and Adaptive execution, which enables the user, the scientist or the administrator to interfere with workflow execution during runtime and even that the system adaptively reacts to the unexpected situations.

The present dissertation deals with two of the above mentioned research areas: the fault tolerance and the adaptive and user-steered execution.

1.1 Motivation

The following subsections summarize the motivation of our research which was conducted during the past few years.

1.1.1 Workflow structure and fault tolerance

The different scientists' communities have developed their own SWfMs, with divergent representational capabilities, and different dynamic support. Although the workflow description language differs from SWfMS to SWfMS according to their scientific research and needs, it is widely acknowledged that Directed Acyclic Graphs (DAG) serve as a top-level abstraction representation tool. Thus, a Scientific workflow can be represented by a $G(V, \vec{E})$, where the nodes (V) represents the computational tasks and the edges ($\vec{E}$) denote the data dependency between them. Concerning graphs a wide range of scientific results have been achieved in order to provide to other scientific disciplines with a simple but easily analyzable model. Also in the context of scientific workflows it is a widely accepted tool to analyze problems in scheduling, workflow similarity analysis and also in workflow estimation problems. However, dynamic execution of scientific workflows is most generally based on external conditions for example on failure statistics about components of execution environments or network elements and provenance data from historical executions. Despite this fact, we think that the structure of the graph representing the scientific workflow holds valuable information that can be exploited in workflow scheduling, resource allocation, fault tolerance and optimization techniques.

The first two thesis groups addresses the following questions to answer:

How much information can be obtained from the structure of the scientific workflows to adjust fault tolerance parameters and to estimate the consequences of a failure occurring during one task concerning the total makespan of the workflow execution?
How can this information be built in a proactive fault tolerance method, in checkpointing?

1.1.2 Adaptive and user-steered execution

From the scientists' perspective workflow execution is like black boxes. The user submits the workflow and at the end he gets a notification about successful termination or failed execution. Concerning long executions and due to the complexity of scientific workflows it may not be sufficient. Moreover, due to the exploratory nature of scientific workflows the scientist or the user may intend to interfere with the execution and based on monitoring or debugging capabilities to carry out a modified execution on the workflow.

In the third thesis group we were looking for the answers for the questions:

How can scientists be supported to interfere with the workflow execution? How can provenance based user-steering be realized?

1.2 Objectives

Motivated by the problems outlined in the previous subsections the objectives of this thesis can be split into two major parts. The first and second thesis groups deal with problems connected to fault tolerance and the third thesis group concerns with adaptive and user steered workflow execution.

1.2.1 Workflow structure and fault tolerance

In the first thesis group I introduce the flexibility zone of a task concerning a certain time delay, and based on this definition I formulate the sensitivity index SI of a scientific workflow model, which gives information on the connectivity property of the workflow. I also introduce the time sensitivity of a workflow model, which gives information about how sensitive the makespan of a workflow to a failure. According to the time sensitivity TS parameter I give an upper and lower limit for the sensitivity index, and based on the sensitivity index I give a taxonomy of scientific workflows.

1.2.2 Adjusting the checkpointing interval

In the second thesis group I present a static Workflow structure based (Wsb) and an Adaptive workflow structure based (Awsb) algorithm which were targeted to decrease the checkpointing overhead compared to the optimal checkpointing intervals calculated by Young (Young 1974) and Di (Di et al. 2013) without effecting the total wallclock time of the workflow. The effectiveness of the algorithms are demonstrated through various simulations. At first I will show the connectivity between the sensitivity index of a workflow model and the effectiveness of the Wsb algorithm. Then I present five different execution scenarios to compare the improvements of each scenario and finally I show the results of simulations that were carried out on random graphs which properties were adjusted according to real workflow models, based on a survey on the myExperiment.org website.

1.2.3 Adaptive and user-steered execution

In the third thesis group I introduce iPoints, special intervention Points with the primary aim to help the scientist to interfere with the execution and according to provenance analysis to alter the workflow execution or to change the abstract model of the workflow. These iPoints are also capable to realize provenance based adaptive execution with the help of a so called Rule Based Engine that can be controled or updated by the scientist or with data mining support. In this thesis group I also give a specification of the above mentioned iPoints in IWIR (*Interoperable Workflow Intermediate Representation*) (Plankensteiner, Montagnat, and Prodan 2011) language, which was developed with the aim to enable interoperability between four existing SWfMSs (ASKALON, P-Grade, MOTEUR and Triana) within the framework of the SHIWA project.

1.3 Methodology

As a starting point of my research I thoroughly investigated the related work in the theme of faults, failures and dynamic execution. According to the reviewed literature I gave a taxonomy about most frequently arising failures during the workflow lifecycle (Bahsi 2008) (Gil et al. 2007) and about the existing solutions that were aimed to provide dynamic execution at a certain level.

This thesis employs two main methodologies to validate and evaluate the introduced formulas, ideas and algorithms. The first is an analytical approach. Taking into account that scientific workflows at the highest abstraction level are generally represented with

Directed Acyclic Graphs, our validation technique is based on investigating the structure of the interconnected tasks.

As graphs can range in size from a few tasks to thousands of tasks, and values assigned to the edges and tasks may diverse, I started with simplifying the workflows with a transformation that eliminates the values assigned to the edges and homogenize the tasks. As a next step I used simple graph models to demonstrate my hypothesis, and afterwards with use of algorithms and methods from the field of graph theory I demonstrated, validated and proved my results.

The second approach was to validate my results with simulations in Matlab, in a numerical computing environment by MathWorks. I have implemented algorithms for the invented formulas and for the checkpointing algorithms as well, and conducted numerous simulations based on special workflow patterns as well as on randomized workflows. For the randomized workflow patterns I took into account a survey on real-life workflows from the myExperiment.org website.

1.4 Dissertation Organization

The dissertation is organized as follows: In the next chapter (2) I summarize the state of the art in the topic of dynamic execution, which served as the background work to my research. In this chapter I give a brief overview about the most frequent failures that can arise during execution, about dynamic execution from a failure handling perspective, about the most popular Scientific Workflow Management Systems (SWfMS), and their capabilities concerning to the fault prevention or fault handling methods. In chapter (3) I present my work on workflow structure analysis. Chapter (4) details the Static workflow structure based (Wsb) and the Adaptive workflow structure based (Awsb) checkpointing methods as well as the simulation results. In chapter (5) I introduce a novel workflow control mechanism, which provides the user intervention points and also for the system an adaptive provenance based steering and control points. This chapter also contains the specification of these intervention points in the IWIR (*Interoperable Workflow Intermediate Representation language*) which was developed within the framework of the SHIWA project and was targeted to promote interoperability between four existing SWfSMs. At the end the conclusion summarize my scientific results.

2 Dynamic execution of Scientific Workflows

Scientific workflow systems are used to develop complex scientific applications by connecting different algorithms to each other. Such organization of huge computational and data intensive algorithms aim to provide user friendly, end-to-end solution for scientists. The various phases and steps associated with planning, executing, and analyzing scientific workflows comprise the scientific workflow life cycle (WFLC) (section 2.1) (Bahsi 2008) (Gil et al. 2007) (Deelman and Gil 2006) (Ludäscher, Altintas, Bowers, et al. 2009). These phases are largely supported by existing Scientific Workflow Management Systems (SWfMS) using a wide variety of approaches and techniques (Yu and Buyya 2005).

Scientific workflows being data and compute intensive, mostly require parallel and distributed infrastructures to be completed in a reasonable time. However, due to the complex nature of these High Performance Computing Infrastructures (clouds, grids and clusters) the execution environment of the workflows are prone to errors and performance variations. In an environment like this dynamic execution is needed, which means that the Scientific Workflow Management System should provide means to adapt to the new environmental conditions, to recover from failures, to provide alternative executions and to guarantee successful termination of the workflow instances with a probability of p.

In this chapter we aim to provide a comprehensive insight and taxonomy about dynamic execution, with special attention of the different faults, fault-tolerant methods and a taxonomy about SWfMSs concerning fault tolerant capabilities.

2.1 Scientific Workflow Life Cycle

- Hypothesis Generation (Modification):

  Development of a scientific workflow usually starts with hypothesis generation. Scientists working on a problem, gather information, data and requirements about the related issues to make assumptions about a scientific process and based on their work they build a specification, which can be modified later on.

- Workflow Design: At this abstraction level, the workflow developer builds a so called abstract workflow. In general this abstract workflow model is independent from the underlying infrastructure and deployed services it only contains the actual steps that are needed to perform the scientific experiment.

  Several workflow design language has been developed over the years, like AGWL (Fahringer, Qin, and Hainzer 2005), GWENDIA (Montagnat et al. 2009), SCUFL (Turi et al. 2007) and Triana Taskgraph (Taylor et al. 2003), since the different scientific communities have developed their own SWfMS according their individual requirements. However, at the highest abstraction level these scientific workflows can be represented by directed graphs $G(V, \vec{E})$, where the nodes or vertices $v_i \in V$ are the computational tasks or jobs an the edges between them represent the dependencies (data or control flow). Fig. 2.1 shows an example of a scientific workflow with 4 tasks (T_0, T_1, T_2, T_e), with T_0 being the entry task and Te being the end task. The numbers assigned to the tasks represent the execution time that is needed to successfully terminate the task and the numbers assigned to the edges represent the time that is needed to submit the successor task after the predecessor task has been terminated. This latter one can be the data transfer time, resource allocation time or communication time between the consecutive tasks.

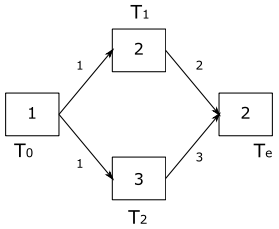

Figure 2.1: Simple workflow with four tasks

  In this phase also the configuration may take place. It means that besides the abstract workflow model a so called concrete workflow model is also generated. The concrete workflow also includes some execution-specific information, like the resource-type, resubmission tries, etc. Once it has been configured an instantiation phase began.

- Instantiation: In this phase the actual mapping takes place, i.e.: the resource allocation, scheduling, parameter and data binding functions.

- Execution: After the workflow instantiation the workflow can be executed.

- Result Analysis: After workflow execution the scientists analyze their results, debug

the workflows or follow the execution traces if the system supported provenance data capturing. Finally, due to the exploratory nature of the scientific workflows after the evaluation of the results the workflow lifecycle may begin again and again.

2.2 Definition of dynamism

Dynamism on one hand is the ability of a system to react or to handle unforeseen scenarios raised during the workflow enactment phase, in a way to avoid certain failures or to recover from specific situations automatically or with user intervention. The adaptation to new situations may range from resubmitting a workflow to even the modification of the whole workflow model. On the other hand the dynamism is the opportunity to change the abstract or concrete workflow model or to give faster execution and higher level performance according to the actual environmental conditions and intermediary results.

We distinguish several levels at the different lifecycle phases of a workflow, where dynamic behaviour can be realized. The system level concerns with those dynamic issues that are supported by the workflow management system. The composition level includes the language or the DAG support. With task level solutions large scale dynamism can be achieved if the system is able to handle tasks as separate units. Workflow level dynamism deals with problems which can only be interpreted in the context of a certain workflow, while user level gives the opportunity for user intervention.

2.3 Taxonomy of dynamism

The dynamism supported by the workflow management systems can be realized in three phases of the above mentioned time intervals of the workflow lifecycle. (i.e.: design, instantiation, execution).

1. **Design time** During design time, dynamism can be primarily supported by the modeling language at composition level. Several existing workflow managers have support for conditional structure in different levels. While some of them provide if, switch, and while structures that we are familiar with from high level languages, some of the workflow managers provide comparatively simple logic constructs. In the latter case, the responsibility of creating conditional structures is left to the users by combining those logic constructs with other existing ones (Wolstencroft et al. 2013).

Heinl et al. (Heinl et al. 1999), (Pesic 2008) gave a classification scheme for flexibility of workflow management systems. He defined two groups: flexibility by selection and flexibility by adaptation. Flexibility by selection techniques also should be implemented in the design time but of course they need some system level support. It can be achieved by advance modeling and late modeling.

The advance modeling technique means that the user can define multiple execution alternatives during the design or configuration phase and the completion or incompletion of the predefined condition decides the actual steps processed in run time. The late modeling technique means, that parts of a process model are not modeled before execution, i.e. they are left as 'black boxes' and the actual execution of these parts are selected only at the execution time.

During this phase the system may also support task level dynamism in a sense, that subworkflows, or tasks from existing workflows should be reusable in other workflows as well. The modular composition of workflows also enables the simple and quick composition of new workflows.

2. **Instantiation time**

   Static decision making involves the risk that decisions may be made on the basis of information about resource performance and availability that quickly becomes outdated (K. Lee, Paton, et al. 2009). As a result with system level support, benefits may appear either from incremental compilation, whereby resource allocation decisions are made for part of a workflow at a time (Deelman, G. Singh, et al. 2005), or by dynamically revising compilation decisions of a concrete workflow while it is executing (Heinis, Pautasso, and Alonso 2005), (Duan, Prodan, and Fahringer 2006), (J. Lee et al. 2007). In principle, any decision that was made statically during workflow compilation should be revisited at runtime (K. Lee, Sakellariou, et al. 2007).

   Another way to support dynamism at system level during instantiation is using breakpoints. To interact with the workflow for tracking and debugging, the developer can interleave breakpoints in the model. At these breakpoints the execution of job instances can be enabled or prohibited, or even it can be steered to another direction (Gottdank 2014).

   We also reckon multi instance activities among the above mentioned system level dynamic issues. Multi instantiation of activities gives flexibility to the execution of workflows. It means that during workflow enactment one of the tasks should be executed with multiple instances (i.e.: parallelism), but the number of instances is

not known before enactment. A way to allow flexibility in data management at system level is to support access to object stores using a variety of protocols and security mechanisms (Vahi, Rynge, et al. 2013).

A task level challenge for workflow management systems is to develop a flexible data management solution that allows for late binding of data. Tasks can discover input data at runtime, and possibly choose to stage the data from one of many locations. At workflow level using mapping adaptations depending on the environment, the abstract workflow to concrete workflow bindings can change. The authors in (K. Lee, Sakellariou, et al. 2007) deal with this issue in details. If the original workflow can be partitioned into subworkflows before mapping, then each sub-workflow can be mapped individually. The order and timing of the mapping is dictated by the dependencies between the sub-workflows. In some cases the sub-workflows can be mapped and executed also in parallel. The partitioning details are dictated by how fast the target execution resources are changing. In a dynamic environment, partitions with small numbers of tasks are preferable, so that only a small number of tasks are bounded to resources at any one time (Ludäscher, Altintas, Bowers, et al. 2009).

In scientific context the most important applications are parameter sweep applications over very large parameter spaces. Practically it means to submit a workflow with various data of the given parameter space. This kind of parallelization gives faster execution and high level flexibility in the execution environment. Scheduling algorithms can also be task based (task level) or workflow based (workflow level) and with system level support the performance and effectiveness of the algorithms can be improved with provenance based information.

3. **Execution time**

   In a dynamically changing environment, during workflow enactment unforeseen scenarios may result in various work item failure (due to faulty results, resource unavailability, etc.). Many of these failures could be avoided with workflow management systems that provide more dynamism and support certain level of adaptivity to these scenarios.

   We categorize the related issues into levels according to Table 2.1.

   The first level is made up from the failure of hardware, software or network component, associated with the work item or data resources unavailability. In these cases, exception handling may or should include mechanisms to detect and to recover from failures (for example restart the job or workflow or make some

other decision based on provenance data), even with provenance based support. In all these cases possible handling strategies should be tracking, monitoring and gathering provenance information in order to support users in coming to a decision.

The user level dynamism consists of scenarios where the system waits for user steering. Here we can rate the breakpoints, where workflow execution can be suspended and enabled again by the user. Also at this time happens the interpretation of the black boxes (late modeling technique). Suspending a workflow and then continue with a new task by deviating from the original workflow model also gives more flexibility to the system at workflow level. In Heinl's taxonomy (Heinl et al. 1999), (Pesic 2008) it is defined by flexibility by adaptation. In this case we distinguish adaptive systems and ad-hoc systems. While adaptive systems modify process model on instances leaving the process model unchanged, in ad-hoc systems the model migrates to a new state, to a new model (Pesic 2008).

According to the above described requirements and suggested solutions we have differentiated the different aspects of dynamism.

2.4 Aspects of dynamism

Depending on the goal of dynamic support dynamic behavior can also be classified into another three categories: 1. The dynamic and adaptive execution of the workflow from the users' point of view. 2. The handling of the various problems and failures arising during execution that cannot be foreseen. 3. The optimizing purpose interventions of the system or the administrator. For example because of the effective or energy save usage of the system or the quick execution of a workflow [K-2].

2.5 Fault tolerance

Scientific workflows may range in size from a few tasks to thousands of tasks. For large workflows it is often required to execute them in a parallel and distributed manner in order to successfully complete the computations in a reasonable time or within soft or hard deadlines. One of the main challenges in workflow execution is the ability of documenting and dealing with failures (Wrzesińska et al. 2006). Failures can happen because resources go down, data becomes unavailable, networks go down, bugs in the system software or in the application components appear, and many other causes.

2.6 Faults, failures and Fault tolerance

Investigating the literature we come across the fault, error, failure expressions, all having very similar meanings for the first sight. To clarify the concepts above the following definition is used.

Fault is defined as a defect at the lowest level of abstraction. A change in a system state due to a fault is termed as an error. An error can lead to a failure, which is a deviation of the system from its specified behavior (Chandrashekar 2015) . To handle failures at first faults should be detected.

In order to detect occurrence of faults in any grid resource two approaches can be used: the push and the pull model. In the push model, grid components by periodically sending heartbeat messages to a failure detector, announce that they are alive. In the absence of this heartbeat messages, the fault detector can recognize and localize the presence of a failure. The system then takes the necessary steps as dictated by the predefined fault tolerance mechanism. Contrariwise, in the pull model the failure detector sends live-ness requests periodically to grid components (H. Lee et al. 2005).

During the different phases of the workflow lifecycle we have to face many types of failures, which lead unfinished task or workflow execution. In these cases the users, instead of getting the appropriate results of their experiment, the workflow process aborts and in general the scientist does not have knowledge about the cause of the failure. In the literature sevaral studies examine the failures occuring during the different phases of the workflow lifecycle from different perspectives (Plankensteiner, Prodan, Fahringer, et al. 2007), (Das and De Sarkar 2012), (Schroeder and G. A. Gibson 2007), (Schroeder and G. Gibson 2010), (Alsoghayer 2011), (X. Chen, Lu, and Pattabiraman 2014), (Samak et al. 2012), (Deelman and Gil 2006). Most of them base their analysis on data that was gathered from a nine-year long monitoring of the supercomputer of the Los Alamos National Labs (LANL). Mouallem (Mouallem 2011) during his research, also based on the data from the Los Alamos National Labs (LANL), revealed that (50%) of the failures is caused by hardware, (20%) by the user, (10%) stems from network or other environmental sources and (20%) of them is unknown.

Based on these studies we have summarized and classified the most frequent failures that can arise during execution time on parallel and distributed environment including grids [**networkshop**] and clouds [**doceis**] environments. The arising failures are examined at four abstract levels, namely the system level, task level, workflow level and user level (Table 2.1). The system level failure deals on the one hand with errors and problems related to the infrastructure (hardware or network failures), on the other hand with

problems related to configuration parameters, which manage the execution. At workflow level we mention those failures, that have impact on the whole workflow and can corrupt the whole execution of the workflow. The task level failures can influence the execution of only one task, and the impact of any failures does not cover the whole workflow. Also we differentiate user level faults during the design time, they are mostly bounded to programming errors (i.e: infinite loop).

After categorizing the potential failures, we show how dynamic behavior (investigated in [K-3]) and provenance support can give solutions for avoiding and preventing them or to recover from situations caused by failures and problems that cannot be foreseen or predicted. In the table after the possible failures there is a '$\Longrightarrow$' sign inserted and then the potential solutions that can be carried out by a dynamic system are presented.

2.7 Taxonomy of Fault Tolerant methods

In this section I present a brief overview about the most frequently used fault tolerant techniques.

Hwang et al. (Hwang and Kesselman 2003) divided the workflow failure handling techniques into two different levels, namely task-level and workflow-level. Task-level techniques handle the execution failure of tasks individually, while workflow-level techniques may alter the sequence of execution in order to address the failures (Garg and A. K. Singh 2011).

Another categorization of the faults can be done according to when the failure handling occurs. Fault tolerance policy can be reactive and proactive. While the aim of proactive techniques is to avoid situations caused by failures by predicting them and taking the necessary actions, reactive fault tolerance policies reduce the effect of failures on application execution when the failure effectively occurs.

According to this classification reactive techniques include: user defined exception handling, retry, resubmission, job migration, using alternative task; proactive techniques are replication, checkpointing.

- Retrying: This might be the simplest task-level failure recovery technique to use with the assumption that whatever caused the failure, it will not be encountered in subsequent retries (Gärtner 1999), (Sindrilaru, Costan, and Cristea 2010).

- Alternative tasks: A key idea behind this failure handling technique is that when a task has failed, an alternative task is to be performed to continue the execution, as opposed to the retrying technique where the same task is to be repeated over and

Table 2.1: Notation of the variables of the Wsb and AWsb algorithm

Design time	system level	task level	user level	
			infinite loop $\Longrightarrow$ **advanced language and modeling support**	
Instantiation time	system level	task level	workflow level	
	HW failures network failures file not found Network congestion task submission failure $\Longrightarrow$**checkpoint** authentication failed $\Longrightarrow$**user intervention** file staging Service not reachable	Incorrect output data Missing shared libraries	infinite loop Input data not available $\Longrightarrow$**data and file replication** Input error $\Longrightarrow$**data and file replication** Data movement failed $\Longrightarrow$**checkpoint**	
Execution time	system level	task level	workflow level	user level
	Hardware failure Network failure File not found Job hanging in the queue of the local resource manager $\Longrightarrow$**dynamic resource brokering** job lost before reaching the local resource manager $\Longrightarrow$**dynamic resource brokering, user intervention**	job crashed $\Longrightarrow$**user intervention,** $\Longrightarrow$**alternate task** $\Longrightarrow$**checkpoint** deadlock $\Longrightarrow$**dynamic resource allocation** $\Longrightarrow$**checkpoint** uncaught exception $\Longrightarrow$ **exception handling,** $\Longrightarrow$ **user intervention**	data movement failed $\Longrightarrow$**checkpoint**	user defined exception

over again which might never succeed. This technique might be desirable to apply in some cases where there are at least two different task implementations available for a certain computation however, has different execution characteristics. (Hwang and Kesselman 2003).

- User-defined Exception Handling: This technique allows users to give a special treatment to a specific failure of a particular task. This could be achieved by using the notion of the alternative task technique.

- Workflow-level redundancy: As opposed to the task-level replication technique

where same tasks are replicated, the key idea in this technique is to have multiple different tasks run in parallel for a certain computation.

- Job Migration: During failure of any task, it can be migrated to another computing resource. (Plankensteiner, Prodan, and Fahringer 2009)

- Task resubmission: It is the most widely used fault tolerance technique in current scientific workflow systems. Whenever a failed task is detected, it is resubmitted either on the same resource or to another one. In general the number of the resubmissions can be configured by the user.

- Replication: When using replication where critical system components are duplicated using additional hardware or with scientific workflows critical tasks are replicated and executed on more than one processor. The idea behind task replication is that replication size r can tolerate $r - 1$ faults while keeping the impact on the execution time minimal. We call r the replication size. While this technique is useful for time-critical tasks its downsides lies in the large resource consumption, so our attention is focused on mainly checkpointing methods in this work. We can differentiate active and passive replication. Passive replication means that only one primary processor is invoked in the execution of a task and in the case of a failure the backup ones take over the task processing. In the active form all the replicas are executed at the same time and in the case of a failure the replica can continue the execution without intervention (Plankensteiner 2013). We also differentiate static and dynamic replication. The static replication means that, when some replica fails, it is not replaced by a new one. The number of replicas of the original task is decided before execution. While in case of dynamic replication, new replicas can be generated during run time (Garg and A. K. Singh 2011)

- Checkpointing: When system state is captured form time to time and when a failure occurs, the last saved state is restored and the execution can be continued from that point on. A more detailed state-of-the art about checkpointing can be found in section 4.1.

2.8 SWfMS

In this section I give a brief overview about the most dominant Scientific Workflow Management Systems (SWfMS). After a short introduction of each SWfMS the focus is on their fault tolerance capabilities.

2.8.1 Askalon

ASKALON (Fahringer, Prodan, et al. 2007) serves as the main application development and computing environment for the Austrian Grid Infrastructure. In ASKALON, the user composes Grid workflow applications graphically using a UML based workflow composition and modeling service. Additionally, the user can programmatically describe workflows using the XML-based Abstract Grid Workflow Language (**AGWL**), designed at a high level of abstraction that does not comprise any Grid technology details. Askalon can detect and recover failures dynamically at various levels.

The Execution Engine provides fault tolerance at three levels of abstraction: (1) activity level, through retry and replication; (2) control-flow level, using lightweight workflow checkpointing and migration; and (3) workflow level, based on alternative task, workflow level redundancy and workflow-level checkpointing. The Execution Engine provides two types of checkpointing mechanisms, lightweight workflow checkpointing saves the workflow state and URL references to intermediate data at customizable execution time intervals and is typically used for immediate recovery during one workflow execution. Workflow-level checkpointing saves the workflow state and the intermediate data at the point when the checkpoint is taken, is saved into a checkpointing database thus it can be restored and resumed at any time and from any Grid location.

2.8.2 Pegasus

The Pegasus (which stands for Planning for Execution in Grids) Workflow Management System (Deelman, G. Singh, et al. 2005), first developed in 2001, was designed to manage workflow execution on distributed data and compute resources such as clusters, grids and clouds.

The abstract workflow description language (DAX, Directed Acyclic graph in XML) provides a resource-independent workflow description. Pegasus dynamically handles failures at multiple levels of the workflow management system building upon reliability features of DAGMan and HTCondor. Pegasus can handle failures dynamically at various levels building on the features of DAGMan and HTCondor. If a node in the workflow fails, then the corresponding job is automatically retried/resubmitted by HTCondor DAGMan. This is achieved by associating a job retry count with each job in the DAGMan file for the workflow. This automatic resubmit in case of failure allows us to automatically handle transient errors such as a job being scheduled on a faulty node in the remote cluster, or errors occurring because of job disconnects due to network errors. If the number of failures for a job exceeds the set number of retries, then the job is marked as a fatal

failure that leads the workflow to eventually fail. When a DAG fails, DAGMan writes out a rescue DAG that is similar to the original DAG but the nodes that succeeded are marked done. This allows the user to resubmit the workflow once the source of the original error has been resolved. The workflow will restart from the point of failure (Deelman, Vahi, et al. 2015). Pegasus has its own uniform, lightweight job monitoring capability: the pegasus-kickstart (Vockler et al. 2007), which helps in getting runtime provenance and performance information of the job.

2.8.3 gUSE/WS-PGRADE

The gUSE/WS-PGRADE Portal (Peter Kacsuk et al. 2012), developed by Laboratory of the Parallel and Distributed Systems at MTA SZTAKI, is a web portal of the grid and cloud User Support Environment (gUSE). It supports development and submission of distributed applications executed on the computational resources of various distributed computing infrastructures (DCIs) including clusters (LSF, PBS, MOAB, SGE), service grids (ARC, gLite, Globus, UNICORE), BOINC desktop grids as well as cloud resources: Google App Engine, CloudBroker-managed clouds as well as EC2-based clouds (Balasko, Farkas, and Peter Kacsuk 2013). It is the second generation P-GRADE portal (Farkas and Peter Kacsuk 2011) that introduces many advanced features both at the workflow and architecture level compared to the first generation P-GRADE portal which was based on Condor DAGMan as the workflow enactment engine.

WS-PGRADE (the graphical user interface service) provides a Workflow Developer UI through which all the required activities of developing workflows are supported the gUSE service set provides an Application (Workflow) Repository service in the gUSE tier.

WS-PGRADE uses its own XML-based workflow language with a number of features: advanced parameter study features through special workflow entities (generator and collector jobs, parametric files), diverse distributed computing infrastructure (DCI) support, condition-dependent workflow execution and workflow embedding support. From a fault tolerance perspective gUSE can detect various failures at hardware -, OS -, middleware, task -, and workflow level. Focusing on prevention and recovery, at Workflow level, redundancy can be created, moreover light-weight checkpointing and restarting of the workflow manager on failure is fully supported. At Task level, checkpointing at OS-level is supported by PGRADE. Retries and resubmissions are supported by task managers. The workflow interpretation permits a job instance granularity of checkpointing, in the case of the main workflow, i.e. a finished state job instance will not be resubmitted during an eventual resume command. However, the situation is a bit worse in the case of embedded workflows, as the resume of the main (caller) workflow can involve the total

resubmission of the eventual embedded workflows (Plankensteiner, Prodan, Fahringer, et al. 2007).

2.8.4 Triana

The Triana problem solving environment (Taylor et al. 2003) (Majithia et al. 2004) is an open source problem solving environment developed at Cardiff University that combines an intuitive visual interface with powerful data analysis tools. It was initially developed to help scientists in the flexible analysis of data sets, and therefore contains many of the core data analysis tools needed for one-dimensional data analysis, along with many other toolboxes that contain components or units for areas such as image processing and text processing. Triana may be classified as a graphical Grid Computing Environment and provides a user portal to enable the composition of scientific applications. Users compose an XML- based task graph by dragging programming components (called units or tools) from toolboxes, and drop them onto a scratch pad (or workspace). Connectivity between the units is achieved by drawing cables. Triana employed a passive approach by informing the user when a failure has occurred. The workflow could be debugged through examining the inbuilt provenance trace implementation and through a debug screen. During the execution, Triana could identify failures for components and provide feedback to the user if a component fails but it did not contain fail-safe mechanisms within the system for retrying a service for example (Deelman, Gannon, et al. 2009). A recent development in Triana at Workflow level light-weight checkpointing and the restart or selection of workflow management services are supported (Plankensteiner, Prodan, Fahringer, et al. 2007).

2.8.5 Kepler

Kepler (Altintas et al. 2004) is an open-source system and is built on the data-flow oriented PTOLEMY II framework. A scientific workflow in Kepler is viewed as a composition of independent components called actors. The individual and resusable actors represent data sources, sinks, data transformers, analytical steps, or arbitrary computational steps. Communication between actors happens through input and output ports that are connected to each other via channels.

A unique property of Ptolemy II is that the workflow is controlled by a special scheduler called Director. The director defines how actors are executed and how they communicate with one another. Consequently, the execution model is not only an emergent side-effect of the various interconnected actors and their (possibly ad-hoc) orchestration, but rather

a prescribed semantics (Ludäscher, Altintas, Berkley, et al. 2006). Kepler workflow management system can be divided into three distinct layers: the workflow layer, the middleware layer, and the OS/hardware layer. The workflow layer, or the control layer provides control, directs execution, and tracks the progression of the simulation. The framework that was proposed in (Mouallem 2011) has three complementary mechanisms: a forward recovery mechanism that offers retries and alternative versions at the workflow level, a checkpointing mechanism, also at the workflow layer, that resumes the execution in case of a failure at the last saved consistent state, and an error-state and failure handling mechanisms to address issues that occur outside the scope of the Workflow layer.

2.8.6 Taverna

The Taverna workflow tool (Oinn et al. 2004), (Wolstencroft et al. 2013) is designed to combine distributed Web Services and/or local tools into complex analysis pipelines. These pipelines can be executed on local desktop machines or through larger infrastructure (such as supercomputers, Grids or cloud environments), using the Taverna Server. The tool provides a graphical user interface for the composition of workflows. These workflows are written in a new language called the simple conceptual unified flow language (Scufl), where by each step within a workflow represents one atomic task. In bioinformatics, Taverna workflows are typically used in the areas of high-throughput omics analyses (for example, proteomics or transcriptomics), or for evidence gathering methods involving text mining or data mining. Through Taverna, scientists have access to several thousand different tools and resources that are freely available from a large range of life science institutions. Once constructed, the workflows are reusable, executable bioinformatics protocols that can be shared, reused and repurposed.

Taverna has breakpoint support, including the editing of intermediate data. Breakpoints can be placed during the construction of the model at which execution will automatically pause or by manually pausing the entire workflow. However, in Taverna the e-scientist cannot find a way to dynamically choose other services to be executed on the next workflow steps depending on the results.

2.9 Provenance

Data provenance refers to the origin and the history of the data and its derivatives (meta-data). It can be used to track evolution of the data, and to gain insights into the analysis performed on the data. Provenance of the processes, on the other hand, enables

scientist to obtain precise information about how, where and when different processes, transformations and operations were applied to the data during scientific experiments, how the data was transformed, where it was stored, etc. In general, provenance can be, and is being collected about various properties of computing resources, software, middleware stack, and workflows themselves (Mouallem 2011).

Concerning the volume of provenance data generated at runtime another challenging research area is provenance data analysis concerning runtime analysis and reusable workflows. Despite the efforts on building a standard Open Provenance Model (OPM) (Moreau, Plale, et al. 2008), (Moreau, Freire, et al. 2008) provenance is tightly coupled to SWfMS. Thus scientific workflow provenance concepts, representation and mechanisms are very heterogeneous, difficult to integrate and dependent on the SWfMS. To help comparing, integrating and analyzing scientific workflow provenance, authors in (Cruz, Campos, and Mattoso 2009) presents a taxonomy about provenance characteristics. PROV-man (Benabdelkader, Kampen, and Olabarriaga 2015) is an easily applicable implementation of the World Wide Web Consortium (W3C) standardized PROV. The PROV (Moreau and Missier 2013) was aimed to help interoperability between the various provenance based systems and gives recommendations on the data model and defines various aspects that are necessary to share provenance data between heterogeneous systems. The PROV-man framework consists of an optimized data model based on a relational database system (DBMS) and an API that can be adjusted to several systems.

When provenance is extended with performance execution data, it becomes an important asset to identify and analyze errors that occurred during the workflow execution (i.e. debugging).

3 Workflow Structure Analysis

Scientific workflows are targeted to model scientific experiments, which consists of data and compute intensive calculations and services which are invoked during the execution and also some kind of dependencies between the tasks (services). The dependency can be data-flow or control-flow oriented, which somehow determine the execution order of the tasks. Scientific workflows are mainly data-dependent, which means that the tasks share input and output data between each other. Thus a task cannot be started before all the input data is available. It gives a strict ordering between the tasks and therefore the structure of a scientific workflow stores valuable information for the developer, the user and also for the administrator or the scientific workflow manager system. Therefore workflow structure analysis is frequently used in different tasks, for example in workflow similarity analysis, scheduling algorithms and workflow execution time estimation problems.

In this chapter I am going to analyze workflows from a fault tolerance perspective. I am trying to answer the questions how flexible a workflow model is; how robust is the selected and applied fault tolerance mechanism; how can the fault tolerance method to a certain DCI , or to the actually available resource assortment fine-tuned.

Proactive fault tolerance mechanisms generally have some costs both in time and in space (network usage, storage). The time cost affects the total workflow execution time, which is one of the most critical constraints concerning scientific workflows, especially time-critical applications. Fault tolerance mechanisms are generally adjusted or fine tuned based on the reliability of the resources or on failures statistics gathered and approximated by the means of historical executions stored in Provenance Database (PD), for example expected number of failures. However, when the mechanism is based on these before mentioned statistical data, the question arises: what happens when more failures occur then it was expected?

With our workflow structure analysis we are trying to answer these questions.

3.1 Workflow structure investigations - State of the art

One of the most frequently used aspect of workflow structure analysis is makespan estimation. In their work (Pietri et al. 2014) the authors have divided tasks into levels based on the data dependencies between them so that tasks assigned to the same level are independent from each other. Then, for each level, its execution time (which is equal to the time required for the execution of the tasks in the level) can be calculated considering the overall runtime of the tasks of the level. With this model they have demonstrated that they can still get good insight into the number of slots to allocate in order to achieve a desired level of performance when running in cloud environments.

Another important aspect of workflow structure investigation is workflow similarity research. It is a very urgent and relevant topic, because workflow re-usability and sharing among the scientists' community has been widely adopted. Moreover, workflow repositories increase in size dramatically. Thus, new challenges arise for managing these collections of scientific workflows and for using the information collected in them as a source of expert supplied knowledge. Apart from workflow sharing and retrieval, the design of new workflows is a critical problem to users of workflow systems (Krinke 2001). It is both time-consuming and error-prone, as there is a great diversity of choices regarding services, parameters, and their interconnections. It requires the researcher to have specific knowledge in both his research area and in the use of the workflow system. Consequently, it would make the researcher's work easier when they do not have to start from scratch, but would be afforded some assistance in the creation of a new workflow.

The authors in (Starlinger, Cohen-Boulakia, et al. 2014) divided the whole workflow comparison process into two distinct level: the level of single modules and the level of whole workflow. First they carry out a comparison comparing the task-pairs individually and thereafter a topological comparison is applied. According to their research in the existing solutions (Starlinger, Brancotte, et al. 2014) regarding topological comparison, existing approaches can be classified as either a structure agnostic, i.e., based only on the sets of modules present in two workflows, or a structure based approach. The latter group makes similarity research on substructures of workflows, such as maximum common subgraphs (Krinke 2001), or using the full structure of the compared workflows as in (Xiang and Madey 2007), where authors use SUBDUE to carry out a complete topological comparing on graph structures by redefining isomorphism between graphs. It returns a cost value which is a measurement of the similarity.

In scheduling problems workflow structure investigations are also a popular form to optimize resource mapping problems. The paper (Shi, Jeannot, and Dongarra 2006)

addresses to solve a bi-objective matching and scheduling of DAG-structured application as both minimize the makespan and maximize the robustness in a heterogeneous computing system. In their work they prove that slack time is an effective metric to be used to adjust the robustness and it can be derived from workflow structure. The authors in (Sakellariou and H. Zhao 2004) introduce a low cost rescheduling policy, which considers rescheduling at a few, carefully selected points during the execution. They also use slack time (we use this term as flexibility parameter in our work), which is the minimum spare time on any path from this node to the exit node. Spare time is the maximal time that a predecessor task can be delayed without affecting the start time of its child or successor tasks. Before a new task is submitted it is considered whether any delay between the real and the expected start time of the task is greater than the slack or the min-spare time. In (Poola et al. 2014) authors present a robust scheduling algorithm with resource allocation policies that schedule workflow tasks on heterogeneous Cloud resources while trying to minimize the total elapsed time (makespan) and the cost. This algorithm decomposes the workflow into smaller groups of tasks, into Partial Critical Paths (PCP), which consist of the nodes that share high dependency between them, for those the slack time is minimal. They declared that PCPs of a workflow are mutually exclusive, thus a task can belong to only one PCP.

To the best of our knowledge workflow structure analysis from a fault tolerance perspective has not been carried out.

3.2 Fault sensitivity analysis

Scientific experiments are usually modeled by scientific workflows at the highest abstraction level, which are composed of tasks and edges and some simple programming structures (conditional structures, loops, etc.). Thus, these scientific workflows can be represented by graphs.

Given the workflow model $G(V, \vec{E})$, where V is the set of nodes (tasks) and $\vec{E}$ is the set of edges representing the data dependency, formally $V = \{T_i | 1 \leq i \leq |V|\}$, $\vec{E} = \left\{ (T_i, T_j) \, | T_i, T_j \in V and \exists \, Ti \rightarrow T_j \right\}$. $|V| = n$ is the number of nodes (tasks in the workflow). Usually scientific workflows are represented with Directed Acyclic Graphs (DAGs), where the numbers associated to tasks specifies the time that is needed to execute the given task and the numbers associated to the edges represent the time needed to start the subsequent task. This latter one can involve data transfer time from the previous tasks, resource starting time, or time spent in the queue. All these values can be obtained from historical results, from a so called Provenance Database (PD) or it can

be estimated based on certain parameters for example on the number of instructions.

Definition 3.2.1. Let $G(V, \vec{E})$ be a DAG. V is the set of vertices, and $\vec{E}$ is the set of directed edges. $Parent(v)$ is the set of parent tasks of v and $Child(v)$ is the set of child tasks of v. Formally, $Parent(v) = \left\{u | u \to v \in \vec{E}\right\}$ and $Child(v) = \left\{u | v \to u \in \vec{E}\right\}$. ■

Definition 3.2.2. Let $G(V, \vec{E})$ be a DAG. V is the set of vertices, and $\vec{E}$ is the set of directed edges. $PRED(v)$ is the predecessor set of v and $SUCC(v)$ is the successor set of v. Formally $PRED(v) = \{u | u \to\to v\}$ and $SUCC(v) = \{u | v \to\to u\}$. Where $u \to\to v$ indicates that there exist a path from v to u in G. ■

In this work we only consider data-flow oriented scientific workflow models where their graph representations are DAGs (Directed Acyclic Graphs) with one entry task T_0 and one exit task T_e. If the original scientific workflow would have more entry tasks or more exit task, then we can introduce a T_{00} entry task which precedes all the original entry tasks and also an exit task T_{ee} which follows all the original exit tasks with parameters of 0 and they were connected to the entry tasks or exit tasks respectively with the 0 value assigned edges.

In such case the calculations are not affected, because path length are not increased due to the 0 parameters.

When a failure occurs during the execution of a task then the execution time of the given task is increased with the fault detection time and recovery time. The recovery time depends from the actually used fault tolerant method.

When the used fault tolerance is a checkpointing algorithm, then the recovery time is composed of the restoring time of the last saved state and the recalculation time from the last saved state. In the case of resubmission technique the recovery time consists of the recalculation time. In the case of a job migration technique the recovery time can be calculated as in the case of using the resubmission method increased by the restarting time of the new resource.

To investigate the effects of a failure we introduce the following definitions:

Definition 3.2.3. The local cost (3.1) of a failure on task T_i is the execution time overhead of the task when during its execution one failure occurs. ■

$$C_{local,i} = t(T_i) + T_r + T_f. \tag{3.1}$$

Definition 3.2.4. The global failure cost (3.2) of a task T_i is the execution time overhead of the whole workflow, when one failure occurs during task T_i. ■

$$C_{global,i} = T_r + T_f + rank(T_i) + brank(T_i), \tag{3.2}$$

where $t(T_i)$ is the expected or estimated execution time of task T_i, T_f and T_r are the fault detection and fault recovery time respectively, the rank() function (3.3) is a classic formula and is generally used in tasks scheduling problems (Topcuoglu, Hariri, and Wu 2002) (L. Zhao et al. 2010). Basically the rank() function calculates the critical path from task T_i to the last task, and can be computed recursively backward from the last task T_e. For simplicity we have introduced the $brank()$ (3.4) function, which is the backward $rank()$ value; from task T_i backward to the entry task T_0. It is the longest distance from the entry task to task T_i excluding the computation cost of the task itself. It can also be calculated recursively downward from task T_0.

$$rank(T_i) = t(T_i) + max_{T_j \in Child(T_i)} rank(T_j), \tag{3.3}$$

$$brank(T_i) = max_{T_j \in Parent(T_i)} (brank(T_j) + t(T_j)). \tag{3.4}$$

A simple definition of the critical path of a program is the longest, time-weighted sequence of events from the start of the program to its termination (Hollingsworth 1998). The critical path in a workflow schema is commonly defined as a path with the longest average execution time from the start activity to the end activity (Chang, Son, and Kim 2002).

Definition 3.2.5. The Critical Path between two tasks T_i and T_j of a workflow is the path in the workflow from task T_i to task T_j with the longest execution time of all the paths that exist from T_i to T_j. ■

Henceforward, we denote the length of the Critical Path between task T_0 and task T_e with CP.

Definition 3.2.6. The relative failure cost (3.5) of a task T_i is the ratio of the global failure cost of task T_i to the execution time of the critical path. ■

$$C_{relative,i} = \frac{T_r + T_f + rank(T_i) + brank(T_i)}{rank(T_0)}, \tag{3.5}$$

If the relative failure cost $C_{relative,i} < 1$ of a failure occurring during the execution of task T_i, then it means that it does not have global effects, because the failure-cost-increased path through task T_i is shorter then the critical path.

If a failure has local or global cost then the child tasks or some of its successor tasks may be started later than it was predestined.

If a failure does not have global effect on the workflow execution time, then we can define the scope of its effect, in other words the set of tasks which submission is postponed for a while due to this failure. To formulate the sensitivity of a workflow model we define the influenced zone of an individual task.

We introduce $T_i.start$ as the earliest possible start time for all $i \in V$ and $T_i.end$ which is the latest end time for all $i \in V$, without negatively affecting the total wallclock time of the workflow.

Definition 3.2.7. The influenced zone of an individual task I_i : is the set of tasks which submission time is affected because a failure is occurred during the execution of task T_i. Formally: $I_i = \left\{T_j \in SUCC(T_i) \mid T_j.start_{pred} = T_j.start + t, t > 0, t \leq C_{local,i}\right\}$ where $T_j.start_{pred}$ is the pre-estimated starting time of T_j. ■

Similarly, we can define the influenced zone for a delay d occurring during the data transmission time between two tasks:

Definition 3.2.8. The influenced zone of an edge between task T_i and T_j is the set of tasks which submission time is affected because a failure is occurred during the execution of task T_i. Formally: $I_{i,j} = \left\{T_k \in SUCC(T_j) \bigcup T_j \mid T_k.start_{pred} = T_k.start + t, t > 0, t \leq C_{local,i}\right\}$ where $T_k.start_{pred}$ is the pre-estimated starting time of T_k. ■

In other words the influenced zone is the set of tasks that constitute the scope of the failure. The influenced zone is always related to a certain delay parameter, in other words the cost of the failure.

To determine the effect of a failure during the execution of T_i on the whole workflow we define the sensitivity parameter of the Task T_i.

Definition 3.2.9. The sensitivity parameter (3.6) of a task T_i is the ratio of the size of the influenced zone I_i of T_i to the size of the remaining subgraph G_i induced by T_i and T_e. ■

$$S_i = \frac{|I_i|}{|G_i|} \tag{3.6}$$

A subgraph $G_i = G(V_i, \vec{E_i})$ is induced if it contains all the edges of the containing graph for which the endpoints are present in V_i. Formally, for all (x, y) vertex pairs of the

subgraph, $(x, y) \in \vec{E_i}$ if and only if $(x, y) \in \vec{E}$. Therefore, in order to specify an induced subgraph of a given graph $G(V_i, \vec{E_i})$, it is enough to give a subset $V_i \in V$ of vertices, as the edge set $\vec{E_i}$ will be determined by G.

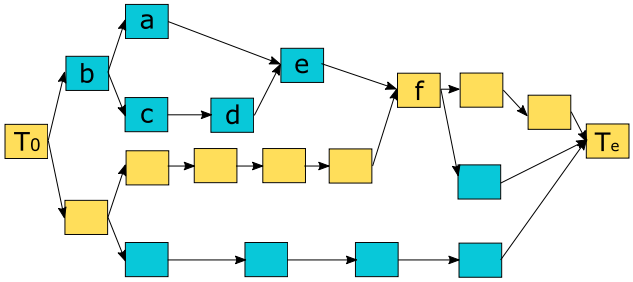

Figure 3.1: A sample workflow graph with homogeneous tasks

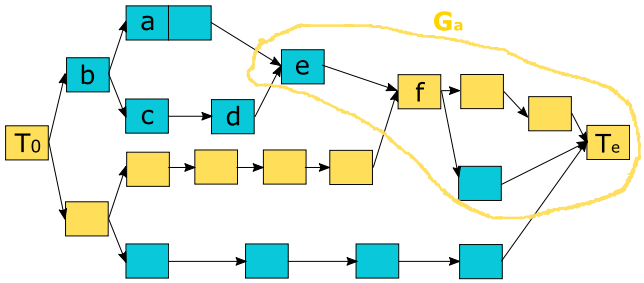

Figure 3.2: A 1-time-unit-long delay occurring during the execution of task a

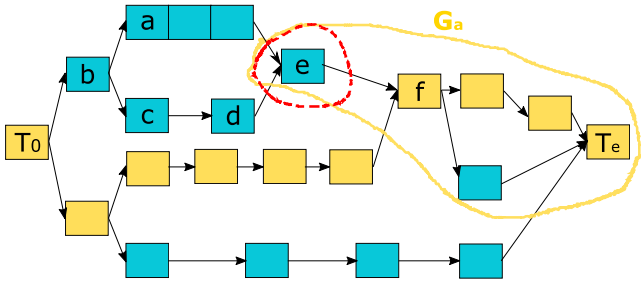

Figure 3.3: A 2-time-unit-long delay occurring during the execution of task a

Figures (3.1, 3.2, 3.3) are representing the meaning of the influenced zone and the sensitivity parameter of a given task a. In Figure (3.1) there is a simple workflow graph consisting of 1-time-unit-long tasks and edges with assigned values of 0. Figure (3.2) represents a 1-time-unit-long delay during the execution of task a and its effects, i.e.: this 1-time-unit-long delay has only local significance since task e cannot be started before all the data are ready from task d. In that case the sensitivity parameter of task a can

be calculated as follows: $SI_a = \frac{0}{G_a}$, where G_a consists of the solid line enclosed tasks. However, if the delay lasts for 2 time-units during the execution of task a, then it has an impact on task e's submission time too, but it still not influences task f. It means that the influenced zone consists of task e and the remaining subgraph is unchanged. Therefore the sensitivity parameter can be calculated as $SI_a = \frac{1}{G_a}$.

Based on the sensitivity parameters of the tasks constituting the workflow we can determine the sensitivity index of the whole workflow:

Definition 3.2.10. The Sensitivity Index (SI) (3.7) of the whole graph $G(V, \vec{E})$ is defined as the ratio ofthe size of the influenced zone to the size of the remaining subgraph summarized by all tasks, and averaged over all tasks ■

$$SI = \frac{\sum_{i=1}^{|V|-1} \frac{|I_i|}{|G_i|}}{|V|}. \tag{3.7}$$

3.3 Determining the influenced zones of a task

To calculate the fault sensitivity of a graph we should investigate the influenced zones of all tasks. To determine it for all tasks we will investigate the graphs representing the scientific workflows.

3.3.1 Calculating the sensitivity index and influenced zones of simple workflow graphs

In the next examples we also worked with simple workflow graphs that consist of homogeneous vertices i.e. all the tasks have uniform properties and all the edges are also uniform, therefore, the execution time for all tasks and the communication costs, the resource allocation costs and network costs are considered to be identical for all task-pairs. The values assigned to the tasks are 1 and the values assigned to the edges are 0. These assumptions are only necessary to simplify the examples and to facilitate the understanding and the proofs. The following calculations can be carried out with arbitrary parametrized workflows without any modifications. The only requirement is that the workflow is a DAG with one entry node T_0 and one exit task T_e.

1. The first example is a very simple graph model containing 3 tasks (T_0, T_1, T_e) in sequential order. We investigate the influenced zones and the sensitivity index for a delay $d < 1$. The influenced zones of the tasks are the remaining subgraph starting

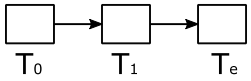

Figure 3.4: simple graph model containing 3 tasks

from the actual task. In this case the sensitivity index of this graph is calculated as follows: if a failure occurs during the execution of T_0 then it has effect on the whole subgraph induced by T_1 as a starting point, i.e. it has effect on all the tasks except T_0. If the failure occurs during the execution of task T_1 then it does not have effect on the predecessor task only on its successor, and so on. The sensitivity index for this first example is calculated by (3.8):

$$SI = \frac{\frac{2}{2} + \frac{1}{1}}{2} = 1. \tag{3.8}$$

2. The second example contains 5 tasks and 2 different paths from T_0 to T_e. It means that the graph includes one cycle if we neglect the orientation of the edges. We also calculate the influenced zones and the sensitivity index for a delay $d < 1$. In this case the sensitivity index of this graph is calculated as follows: If a failure occurs during the execution of T_0 then it has effect on the whole graph i.e. it has effect on all other tasks. It is generally true for all workflows with one entry task. If the failure occurs during the execution of task T_1 then it does not have effect on the predecessor task and on the tasks that are part of a path that does not include T_1 only on its successors, so the other path of the graph is not affected. The influenced zone for task T_3 can be calculated similarly. Due to the fact that a task cannot be submitted before all input data has not arrived from all of its predecessor tasks the influenced zone of task T_2 does not contain the exit task T_e, because the path through T_2 is shorter than the other one, so a $d < 1$ delay during the task T_2 does not cause the exit task to start later 3.9.

$$S = \frac{\frac{4}{4} + \frac{2}{2} + \frac{1}{1} + \frac{1}{2}}{4} = \frac{7}{8}, \tag{3.9}$$

In workflow models that are in focus of our investigations from the entry task T_0 to the exit task T_e may exist several different paths. If there exists only one path from T_0 to T_e so the graph execution model is sequential than the sensitivity of the graph is very high (in this case the sensitivity index is 1), so very strict fault tolerance method should be used, because all failures have global effects.

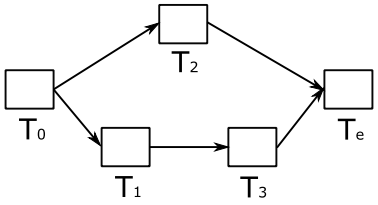

Figure 3.5: Simple graph model containing 2 different paths

If a task is part of all of the paths that exist from T_0 to T_e then it can be easily noticed that these tasks have high sensitivity, because occurring a failure during the execution of these tasks the overall makespan is under all circumstances increased with the local cost of these failure, i.e.: in this case the local cost is equal to the global cost of the failure. From that follows that in our graph model where only one entry task T_0 and one exit task T_e exists, the entry and exit task's sensitivity is high because they are part of all the paths.

Those tasks that are not part of all paths from T_0 to T_e may have smaller influenced zones, because there may exist longer paths parallel to this one, so the failure cost may not effect the global makespan.

It could be also noticed that the border of the influenced zone(s) is in general an edge before a task where at least two paths join together. So if we ignore the orientation of the edges we can conclude that influenced zones have some connection to the cycles in the workflow graph. It can be discovered very easily in simple graphs but for complex graphs with high numbers of vertices and paths is not so easy.

3.3.2 Calculating the Influenced Zones of complex graphs containing high number of vertices

It can be seen that on simple examples it is very easy to calculate the influenced zones and the sensitivity index, but in complex workflow structures with high number of vertices, it would need very long time to carry out an exhaustive search for all tasks. So these results only show us the theoretical possibility to take into account the workflow structures to adjust fault tolerant methods or scheduling tasks.

A naive algorithm to find all the influenced zones for each task for a local failure cost, would be to calculate the length for all the paths in the workflow graph. We denote the critical path between two nodes T_i and T_j $f(T_i, T_j)$. The influenced zone of task T_i is $I_i = \left\{T_j |\ f(T_0, T_j) - f(T_0, T_i) - f(T_i, T_j) < C_{local,i}\right\}$. Thus if we know the $f(T_i, T_j)$ values for all (T_i, T_j) pairs we can determine the influenced zones. The number of directed

paths in a DAG can be exponentially big, so we have to find an other solution which time complexity is a polynomial function of the number of nodes and edges.

Our algorithm is based on a DFS algorithm and consists of the following three steps:

1. Calculating the flexibility parameter for each task
2. Determining the influenced zones of each node
3. Determining the subgraph for each task which is induced by the node and all of its successors.

The DFS algorithm is an exhaustive search carried out in a graph to discover all the nodes starting from a selected (usually the root) one. The algorithm tries to systematically discover all nodes in the following way: starting from a selected node a new neighbor is discovered only when the subgraph connecting to the previous one has been completely discovered.

1. The first step of our algorithm is to carry out a Depth-First Search (DFS) on the workflow model; during the search, the following values must be stored to each node T_i, T_i.start is the earliest possible start time and T_i.end represents the latest possible end time of a node (task) T_i without affecting the total execution time of the workflow.

   By going through the workflow with DFS from the entry task T_0 to the exit task T_e, we calculate and store values $T_i.start$ in each step by summarizing the values $T_j.start$ for all $T_j \in Parent(T_i)$, and the time that is needed to start task T_i (values assigned to edge (T_i, T_j) for all $T_j \in Parent(T_i)$, and we only store the maximum of these values.

   The T_i.end time for all T_i can be calculated in a similar manner, recursively backwards from the last or exit task T_e.

   Definition 3.3.1. Given DAG $G(V, \vec{E})$, the flexibility parameter of $T_i \in V$ is $flex[T_i] = CP - T_i.end - T_i.start$. ■

   In other words, the flexibility parameter of task T_i (or slack time as in (Sakellariou and H. Zhao 2004)) gives the time flexibility of a task, in which the task execution can be freely managed. This is the available time for this task to be successfully completed, without negatively affecting the total wallclock time of the workflow.

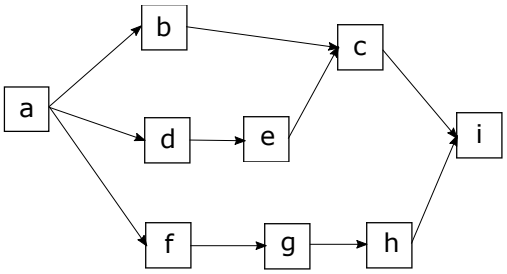

Figure 3.6: An example workflow with one critical path

If $flex[T_i] = t(T_i)$ for vertex T_i, this means that this node does not have any flexibility in time, where $t(T_i)$ is the calculation time of node T_i.

Since we investigate workflows here with one entry node T_0 and one exit task T_e, these two nodes are surely part of the critical path in all cases; so, for their flexibility parameter $flex[T_0] = t(T_0)$ and $flex[T_e] = t(T_e)$ stand.

It can be also generally declared that, if for task T_i $flex[T_i] = t(T_i)$, then this task must be part of at least one of the critical paths.

Corollary. *If $flex[v] = t(v)$ for a vertex v than this node is part of the critical path or of one of the critical paths.*

proof: $flex[v] = 0$ for a vertex v means that the earliest and latest starting time of this node v is the same. In other words the critical path length starting from the entry task T_0 to v plus the critical path length starting from node v to the entry task T_e is equal to the longest path of the workflow, since the v.start and v.end values store the minimum and the maximum of all respectively.

Corollary. *All the nodes v that are part of the critical path or of one of the critical paths have $flex[v] = t(v)$.*

This fact is the simple consequence of the definitions for the critical path and the flexibility parameter.

Fig. 3.6 shows a simple workflow model. For the sake of simplicity in this scenario, we also assume that the data transfer time is 0 (values assigned to edges are all 0), and all of the tasks need one time unit to be executed.

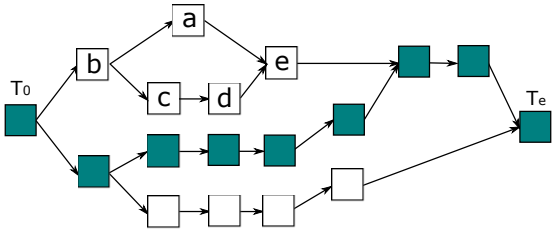

Figure 3.7: An example workflow with one critical path

Thus, there are two critical paths in the workflow: $a \to d \to e \to c \to i$ and $a \to f \to g \to h \to i$. From that follows that for all these tasks that are part of the critical paths $flex[a] = flex[d] = ... = flex[i] = 1$. There is only one task, b for which $flex[b] = 2$.

The time complexity of the algorithm to calculate the flexibility parameter for all tasks is $O(n + e)$.

2. If we have the flexibility parameters for all nodes, we have to determine the influenced zones.

   In the series of figs. 3.7, 3.8, 3.9 the change of the influenced zone of task T_i can be observed.

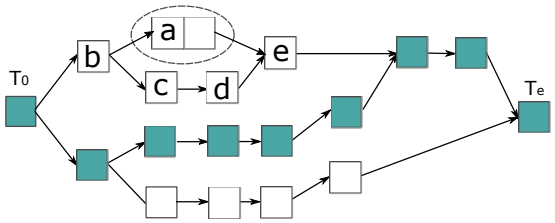

Figure 3.8: Effect of a one-time-unit delay during the execution of task a

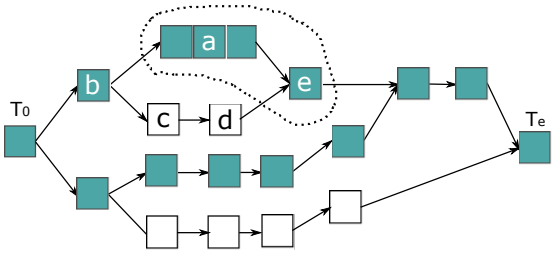

Figure 3.9: Effect of a two-time-unit delay during the execution of task a

In fig. 3.7 the critical path is built up from the blue tasks before submitting the

workflow. As a result, the flexibility parameters for all white tasks are two times the unit except for task a, where this value is three times the unit.

In figure 3.8 during the execution of task a, a 1-time-unit-long failure has occurred. Since $flex[a] = 3$ and $flex[e] = 2$, this 1-unit delay has only local significance. This means that this delay will not effect subsequent task e's submission time (it can also be determined from the alternative path through tasks c and d). So, the influenced zone of this task concerning to this failure is empty.

In Figure 3.9, the delay caused by the failure occurring during the execution of task a is two-times-the-unit long. In this case, the influenced zone is the set of tasks enclosed with the dotted line except the task a itself. This means that, due to this delay, task e should start later, but the successor tasks of task e are not influenced, so the workflow-execution time can still remain the originally estimated time.

However, this delay has another effect as well; namely, the path driving through task a also became a critical path in addition to the original one. As a consequence, if any further failure occurs during this path, the entire workflow execution lasts longer.

From the above example it can be clearly noticed that the border of an influence zone is always an edge that is directed to a certain task which indegree > 1. This is because tasks belonging to a simple path have some flexibility in time, if there exists at least one parallel path, which is longer. The nodes for which indeegre > 1 we call sink. The name can be conducted from the fact that these nodes may decrease or eliminate the flexibility parameter with a certain amount.

So according to our observations we determined an algorithm to find the influenced zone for each task. Starting from the entry task T_0 we carry out a DFS like search for all nodes covering all paths from the starting point until a sink node has not been found, which eliminates the flexibility parameter of the starting node. The pseudocode of determining the influenced zones can be seen in Listing 3.1.

The pseudocode of the influenced zone is a recursive formula, that starting from a given node steps along all paths stemming from that node, until a sink node on that path is found. The *for* loop in line 1 is responsible for determining the influenced zone for all tasks from T_0, except the last task, since it does not have an influenced zone. The second *for* loop in line 2 goes through on all paths from the actual node. Line 3 determines the paths stemming from the actual node. The simple conditions below from line 4 to line 10 determine whether the child task of the actual task is a sink task concerning the starting node. Line 13 helps to step

forward on the path.

Listing 3.1: Calculating the influenced zone

```
1  INFLUENCE(i)
2    for i=1:n-1
3      for j=i+1:n
4        if (i,j)∃ E→
5          if flex(i) > d
6            if    (flex(i) ≤ flex(j) then j ∈ I_i;
7              elseif flex(i) − flex(j) > d; (j = n + 1);
8              else j ∈ I_i; d = d − abs(flex(i) − flex(j));
9            end
10         elseif flex(i) == 0 then j ∈ I_i;
11         else    flex(i) < dI(i, j) = 1; d = d − (flex(i) − flex(j))
12       end
13       if (j < (n + 1))
14         I=influence(j);
15       end
16     end
17 end
```

The time complexity of this search is $O(n \cdot (n + e))$, since the worst case is when the starting node is the entry node T_0.

3. To determine the subgraph for each node which consists of the node and all of its successors we have to carry out a DFS like search again starting from the actual node. Its time complexity is $O(n \cdot (n + e))$ again, because the worst case is that the actual node is the entry node of the workflow.

   The time complexity of the whole algorithm is then $O(n^2)$.

3.4 Investigating the possible values of the Sensitivity Index and the Time Sensitivity of a workflow model

Robustness is considered as an important measurement for a good schedule, since higher robustness indicates that the schedule is likely to remain valid in a dynamic, changing, and non-deterministic environment. Similarly to robustness analysis of scheduling, we can define that a fault tolerance method is robust if any of the resources encounters at least one more failure then it was expected the workflow execution can still meet the calculated deadline.

Definition 3.4.1. The Time Senstivity (TS) (3.10) of the whole graph $G(V, \vec{E})$ is defined as the number of nodes in the graph for that is true, that a delay during the tasks's execution time would negatively effect the total wallclock execution time of the workflow, i.e. the workflow execution cannot be completed before deadline. ■

Formally:

$$TS_i = \begin{cases} 1, \ if flex(i) > t(T_i) + d \\ 0, \ if flex(i) < t(T_i) + d, \end{cases}$$

,

where TS_i is the time sensitivity of a task T_i regarding a time delay of d.

$$TS = \frac{\sum_{i=1}^{n-1} TS_i}{n-1} \ where \ n = |V|. \tag{3.10}$$

To simplify the understanding and the proof, we henceforward assume, that workflows consist of 1-time-unit-long tasks and edges with assigned value of 0.

As a consequence of the definitions (3.7 and 3.10), it can be easily noticed that for both parameters: $0 < SI < 1$ and $0 < TS < 1$ stands.

We are looking for the answer for the question how flexible those nodes are, which time sensitivity is not 0 concerning to a certain delay d. Thus we introduce the modified version of the sensitivity index:

Definition 3.4.2. The sensitivity index for flexible nodes SIF is defined as the ratio of the size of the influenced zone to the size of the remaining subgraph summarized by the tasks for which $TS_i > 0$, and averaged over the tasks for which $TS_i > 0$. ■

It can be calculated as follows:

$$SIF = \frac{\sum_{i=1}^{SF\cdot(n-1)} SI_i}{TS}, \ where \ TS_i > 0 \ and \ n = |V|. \tag{3.11}$$

As a consequence of the definition $SIF \geq 0$.

All the other nodes have time sensitivity 0, therefore these nodes should not be delayed if it is possible. We differentiate two cases to determine the possible values for the Sensitivity Index for flexible nodes (SIF) and the Time Sensitivity (TS).

1. In the first case we assume that the delay $d < 1$ for which we calculate the above mentioned two parameters (SIF, TS). In this case we can declare that those nodes that have $flex(i) = 1$ constitute the Critical Path or one of the Critical Paths.

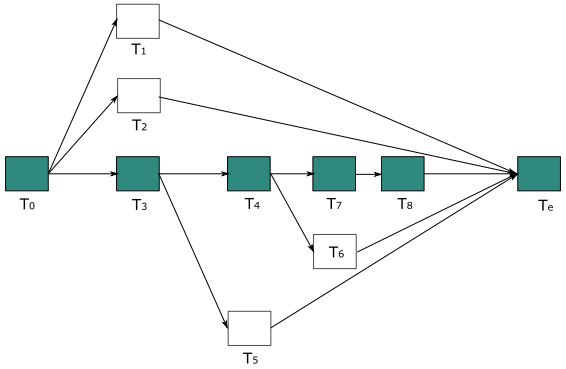

Figure 3.10: A most flexible workflow with a $TS = \frac{5}{9}$

In this case we can give the lower limit and upper limit for the Sensitivity Index concerning to the a given Time Sensitivity. It means, that if we have calculated the Time Sensitivity, then according to this value we can give the classification of the workflow, or can give some reference value, to determine how sensitive is the model. The most flexible is our workflow model when all the nodes that are not part of the critical path are completely independent from each other. In this case a delay during the execution of these tasks do not affect the execution of the other tasks. The equation for the this case is as follows 3.12:

$$SIF = \frac{\sum_{i=1}^{TS\cdot(n-1)} \frac{I_i}{G_i} = \frac{0}{G_i}}{TS}. \tag{3.12}$$

Because the most flexible a workflow model with a given TS value, when all the flexible nodes' influenced zone is 0 ($G_i > 0$ for all $i \in (1, n-1)$, because the subgraph G_i for all nodes T_i must contain at least the exit task T_e). The figure 3.10 presents a prime example for this case, where the dark nodes constitute the critical path, and the empty nodes are the flexible ones. As it can be noticed these nodes are not connected to each other, so their corresponding subgraphs contain only the exit task.

The most sensitive is a workflow concerning to a given TS, when all the flexible nodes are connected with almost all the other flexible nodes. The influenced zone of a flexible node T_i cannot contain any node T_j for which $TS_j = 0$, because it would mean that this node T_i has $TS_j = 0$ also, thus the node would not be flexible. In

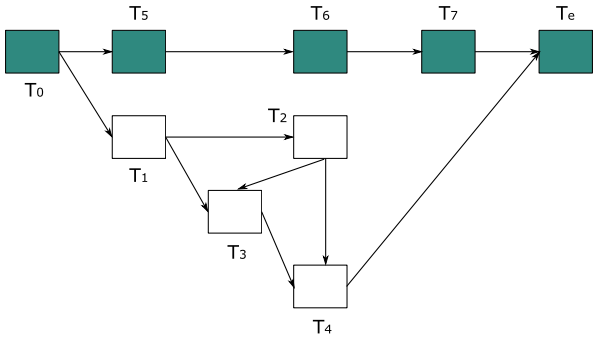

Figure 3.11: A most sensitive workflow with a $TS = \frac{5}{8}$

this case the upper limit of the flexibility can be calculated as follows:

$$SIF = \sum_{i=1}^{TS\cdot(n-1)} \frac{i-1}{i} \tag{3.13}$$

This is a simple consequence of having a fraction of $\frac{I_i}{G_i}$ as high as we can.

An example for a graph like this is illustrated in Fig. 3.11, where the influenced zones and the remaining subgraphs for each tasks are as follows:
$I_1 = \{T_2, T_3, T_4\}$
$I_2 = \{T_3, T_4\}$
$I_3 = \{T_4\}$
$I_4 = \{\}$

$G_1 = \{T_2, T_3, T_4, T_e\}$
$G_2 = \{T_3, T_4, T_e\}$
$G_3 = \{T_4, T_e\}$
$G_4 = \{T_e\}$

and thus the SIF value is calculated in Eq 3.14:

$$SIF = \frac{\overbrace{\frac{3}{4}}^{T_1} + \overbrace{\frac{2}{3}}^{T_2} + \overbrace{\frac{1}{2}}^{T_3} + \overbrace{\frac{0}{1}}^{T_4}}{4} = \frac{23}{48} \tag{3.14}$$

2. In the second case we assume that for the delay $d > 1$ stands. In this case it can be declared that not only the nodes building the Critical Path can have a $TS_i = 0$

value 3.12. There may exist tasks which flexibility value $flex(i) > 1$ but $TS_i = 0$, so their $flex(i) - 1 > d$. Thus for a node T_i that has $TS_i = 0$ the influenced zone I_i may contain a node T_j with the above mentioned properties (let us denote the number of these nodes with cs). As a consequence the upper limit of SIF_i in this case can be higher as in the first case:

$$SIF_i \leq \frac{TS \cdot (n-1) + cs - 1}{TS \cdot (n-1) + cs}, \tag{3.15}$$

$$SIF = \sum_{i=cs+1}^{TS\cdot(n-1)+cs} \frac{i-1}{i}. \tag{3.16}$$

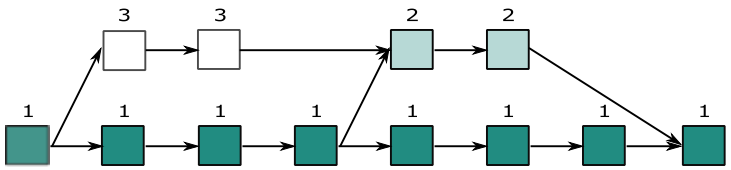

Figure 3.12: An example workflow for a most sensitive workflow with $TS = \frac{9}{11}$

3.5 Classification of the workflows concerning the sensitivity index and flexibility index

To highlighten the significance of a value from the fault tolerance perspective we can classify scientific workflows based on this value.

1. We talk about **totally rigid** workflows, when this sensitivity index for flexible nodes SIF cannot be calculated, because the time sensitivity $TS = n-1$. It means that all nodes are time sensitive, so none of them can tolerate a delay of d without affecting the total wallclock time of the workflow. In this case very strict fault tolerant method should be used.

2. The workflow is **most flexible** under a given TS value when $SIF = 0$. This is the case where fault tolerant can be fine-tuned, maybe other optimization factors can also be considered, for example network usage, storage capacity, etc.

3. The workflow is **most sensitive** under a given TS value when $SIF = \sum_{i=1}^{TS\cdot(n-1)} \frac{i-1}{i}$ or $TS = \sum_{i=cs+1}^{TS\cdot(n-1)+cs} \frac{i-1}{i}$ according to the two cases defined in the previous

subsection 3.4. With workflows belonging to this class one must carefully adjust the parameters of the fault tolerant method.

4. And in all the other cases we talk about **flexible workflow** models, where we get a value for the sensitivity index, and reference values as an upper and a lower limit under a given TS value. Therefore we or the SWfMS is able to decide whether it worth adjusting the fault tolerance parameters or the most strict fault tolerance mechanism should be followed. In the next chapter 4 we give an example how to use the flexibility of a workflow in a fault tolerance mechanism.

3.6 Conclusion

In this chapter I have analyzed the effects of a fault occurring during the execution of one task. I have introduced the influenced zone of a failure and I have investigated the connectivity property of a workflow graph from a fault tolerant perspective. Based on the influenced zone of a failure I have formulated the sensitivity index of a graph, which gives us information about the workflow flexibility, in other words how sensitive is a workflow concerning a failure. According to this value I have classified the workflow models as totally rigid, most flexible, most sensitive, or flexible workflow. The results of this thesis group can be used in fault tolerant methods. The next chapter gives a prime example for it, where the checkpointing interval can be adjusted based on the flexibility parameter.

3.7 New Scientific Results

Thesis group 1.: Fault sensitivity analysis
Thesis 1.1:

> ***Thesis 1.1***
> **I have defined the influenced zone of a task in a workflow represented with DAG, concerning to a certain time delay. Based on the influenced zones of the tasks I have defined the workflow sensitivity index which can help in fine-tuning the actually used fault tolerant method.**

Thesis 1.2:

Thesis 1.2

I have developed an algorithm to calculate the influenced zone of a task and sensitivity index for complex graphs consisting of a high number of tasks and data dependencies. The time requirement of this algorithm is a polynomial function of the number of tasks and edges.

Thesis 1.3:

Thesis 1.3

I gave a classification for the workflows based on their workflow structure analysis.

Relevant own publications pertaining to this thesis group: [K-7; K-9; K-6; K-5; K-3; K-1]

4 Adjusting checkpointing interval to flexibility parameter

Real time users typically want to know an estimation about the execution time of their application before deciding to have it executed. In many cases this estimation can be considered to be a soft deadline that shall be satisfied with some probability without serious consequences. Moreover, time critical scientific workflows to be successfully terminated before hard deadlines imposes many challenges. Hard deadline means that the results are only meaningful before the hard deadline, if any of the results are late then the whole computational workflow and its executions are a waste of time and energy. Many research field face time constraints and soft or hard deadlines to task execution.

Furthermore, scientific workflows are mainly enacted on distributed and parallel computing infrastructures such as grids, supercomputers and clouds. As a result, a wide variety of failures can arise during execution. Scientific workflow management systems should deal with the failures and should provide some kind of fault tolerant behavior. There are a wide variety of existing fault tolerant methods, but one of the most frequently used proactive fault tolerant method is checkpointing, where system state is captured from time to time and in the case of a failure the last saved and consistent state is restored.

However, capturing checkpoints generates costs both in time and space. On one hand the time overhead of the checkpointing can have great impact on the total processing time of the workflow execution and on the other hand the needed disk size and network bandwidth usage can also be significant. By dynamically assigning the checkpointing frequency we can eliminate unnecessary checkpoints or where the danger of a failure is considered to be severe we can introduce extra state savings. Checkpoints also have impact on network usage, when the aim is to save the states on a non-volatile storage and also have impact on storage capacity. Considering all these facts one can conclude that checkpointing can be very expensive. So one must consider taking checkpoints, while taking into account the Pros and Cons.

In this chapter I would like to introduce our novel static (Wsb) and adaptive (AWsb) checkpointing methods for scientific workflows based on not communicating, but parallel

executable jobs, that is primarily based on workflow structure analysis introduced in chapter 3. The proposed algorithms try to utilize the above introduced flexibility values in order to decrease the checkpointing cost in time, without affecting the total wallclock execution time of the workflow. I also show the way this method can be used adaptively in a dynamically changing environment. Additionally, the adaptive algorithm creates the possibility for the scientist to get feedback about the remaining execution time during enactment and the possibility to meet a predefined soft or hard deadline.

4.1 Related work

Concerning dynamic workflow execution fault tolerance is a long standing issue and checkpointing is the most widely used method to achieve fault tolerant behavior.

The checkpoint scheme consists of saving intermediate states of the task in a reliable storage and, upon a detection of a fault, restoring the previously stored state. Hence, checkpointing enables to reduce the time to recover from a fault, while minimizing loss of the processing time.

The checkpoint can be stored on temporary as well as stable storage (Oliner et al. 2005). Lightweight workflow checkpointing saves the workflow state and URL references to intermediate data at adjustable execution time intervals. The lightweight checkpoint is very fast because it does not backup the intermediate data. The disadvantage is that the intermediate data remain stored on possibly unsecured and volatile file systems. Lightweight workflow checkpointing is typically used for immediate recovery during one workflow execution.

Workflow-level checkpointing saves the workflow state and the intermediate data at the point when the checkpoint is taken. The advantage of the workflow-level checkpointing is that it saves backup copies of the intermediate data into a reliable storage so that the execution can be restored and resumed at any time and from any location. The disadvantage is that the checkpointing overhead grows significantly for large intermediate data.

According to the level, where the checkpointing occurs we differentiate: application level checkpointing, library level checkpointing and system level checkpointing methods. Application level checkpointing means that the application itself contains the checkpointing code. The main advantage of this solution lies in the fact, that it does not depend on auxiliary components, however, it requires a significant programming effort to be implemented while library level checkpointing is transparent for the programmer. Library level solution requires a special library linked to the application that can perform the

checkpoint and restart procedure. This approach generally requires no changes in the application code, however, explicit linking is required with user level library, which is also responsible for recovery from failure (Garg and A. K. Singh 2011). System level solution can be implemented by a dedicated service layer that hides the implementation details from the application developers but still give the opportunity to specify and apply the desired level of fault tolerance (Jhawar, Piuri, and Santambrogio 2013).

Checkpointing schemes can also be categorized to be full or incremental checkpoints. A full checkpoint is a traditional checkpoint mechanism which occasionally saves the total state of the application to a local storage. However, the time consumed in taking checkpoint and the storage required to save it is very large (Agarwal et al. 2004). Incremental checkpoint mechanism was introduced to reduce the checkpoint overhead by saving the pages that have been changed instead of saving the whole process state. The performance tradeoff between periodical and incremental checkpointing was investigated in (Palaniswamy and Wilsey 1993).

From another perspective we can differentiate coordinated and uncoordinated methods. With coordinated checkpointing (synchronous) the processes will synchronize to take checkpoints in a manner to ensure that the resulting global state is consistent. This solution is considered to be domino-effect free. With uncoordinated checkpointing (independent) the checkpoints at each process are taken independently without any synchronization among the processes. Because of the absence of synchronization there is no guarantee that a set of local checkpoints results in having a consistent set of checkpoints and thus a consistent state for recovery. It may lead to the initial state due to domino-effect. Meroufel and Belalem (Meroufel and Belalem 2014) proposed an adaptive time-based coordinated checkpointing technique without clock synchronization on cloud infrastructure. Between the different Virtual Machines (VMs) jobs can communicate with each other through a message passing interface. One VM is selected as initiator and based on timing it estimates the possible time interval where orphan and transit messages can be created. There are several solutions to deal with orphan and transit messages, but most of them solve the problem by blocking the communication between the jobs during this time interval. However, blocking the communication increases the response time and thus the total execution time of the workflow, which can lead to SLA (Service-level Agreement) violation. In Meroufel's work they avoid blocking the communication by piggybacking the messages with some extra data so during the estimated time intervals it can be decided when to take checkpoint or logging the messages can resolve the transit messages problem. The initiator selection is also investigated in Meroufel and Belalem's another work (Meroufel and Ghalem 2014) and they found that the impact of initiator

choice is significant in term of performance. They also propose a simple and efficient strategy to select the best initiator.

The efficiency of the used checkpointing mechanism is strongly dependent on the length of the checkpointing interval. Frequent checkpointing may increase the overhead, while rarely made checkpoints may lead to loss of computation. Hence, the decision about the size of the checkpointing interval and the checkpointing technique is a complicated task and should be based upon the knowledge specific to the application as well as the system. Therefore, various types of checkpointing optimization have been considered by the researchers.

Young in (Young 1974) has defined the formula for the optimum periodic checkpoint interval, which is based on the checkpointing cost and the mean time between failures (MTBF) with the assumption that failure intervals follow an exponential distribution. Di et al. in (Di et al. 2013) has also derived a formula to compute the optimal number of checkpoints for jobs executed in the cloud. His formula is generic in a sense that it does not use any assumption on the failure probability distribution.

Optimal checkpointing is often investigated with different conditions. In (Kwak and Yang 2012) authors try to determine the static optimal checkpointing period that can be applied to multiple real-time tasks with different deadlines. There are also optimization investigations when more different checkpoints are used. In (Nakagawa, Fukumoto, and Ishii 2003) authors use double modular redundancy, in which a task is executed on two processors. They use three types of checkpoints: compare-and-store checkpoints, store-checkpoints and compare checkpoints and analytically computed optimal checkpointing frequency as well.

The drawback of these static solutions lies in the fact that the checkpointing cost can change during the execution if the memory footprint of the job changes, network issues arise or when the failure distribution changes. Thus static intervals may not lead to the optimal solution. By dynamically assigning checkpoint frequency we can eliminate unnecessary checkpoints or where the danger of a failure is considered to be severe extra state savings can be introduced.

Also adaptive checkpointing shemes have been developed in (Z. Li and H. Chen n.d.) where compare checkpoints and store checkpoints have placed between compare and store checkpoints according to two different adaptive schemes. Di et al. also proposed another adaptive algorithm to optimize the impact of checkpointing and restarting cost (Di et al. 2013). Theresa et al in their work (Lidya et al. 2010) propose two dynamic checkpoint strategies: Last Failure time based Checkpoint Adaptation (LFCA) and Mean Failure time based Checkpoint Adaptation (MFCA), which takes into account the stability of

the system and the probability of failure concerning the individual resources.

In this work the determination of the checkpointing interval, besides some failure statistics is primarily based on workflow characteristics which is a key difference from existing solutions. To the best of our knowledge our work is unique in this aspect. We demonstrate that we can still get good insight into the number of checkpoints during a job execution in order to achieve a desired level of performance with minimum overhead of the used fault tolerant technique.

4.2 The model

4.2.1 General notation

Given a workflow model $G(V, \vec{E})$, V is the set of nodes (tasks in the workflow) and $\vec{E}$ is the set of edges representing data dependency. There are $|V| = n$ tasks and m resources in the system. The execution time of a task without any fault tolerant behavior and without any failures (i.e., the calculation time of task T_i on resource j) is $t(T_i)_j$. This $t(T_i)_j$ value can be obtained from a provenance database or can be estimated based on the number of instructions the code contains. Table 4.1 summarizes the notation for the variables of our model.

Table 4.1: Notation of the variables of the Wsb and AWsb algorithm

$t(T_i)_j$	Calculation time of task T_i on resource j
$t_{f,j}$	Fault detection time on resource j
$t_{s,j}$	Restart time on resource j
$C_j(t)$	Checkpointing cost on resource j
C	Checkpointing cost (considered constant)
$T_{C,j}$	Checkpointing interval on resource j
T_C	The checkpointing interval
$R_{i,j}$	Recomputation time of task T_i on resource j
T_{opt}	The optimal checkpointing interval
X_i	Optimal number of checkpoints during the execution of a task T_i
T_f	Mean time between failures (MTBF)
$E(Y)$	Expected number of failures during the execution of a task
T_l	Loading time, to restore the last saved checkpoint state
T_0	First or entry task of the workflow
T_e	Last or exit task of the workflow

4.2.2 Environmental Conditions

The following assumptions are used in our algorithms:

- The system resources are monitored and failures can be detected as soon as possible, therefore the fault detection time (t_f) does not add high latency to the overall makespan of the workflow execution ($t_f = 0$ considered during our research).
- Task T_j cannot be started before it has received the output from all its predecessors and the results of a Task T_i can only be sent to its successor tasks after the task has been finished. Concerning a simple workflow as in Fig. 3.5 task T_e can only be started after the successful termination of both tasks T_3 and T_2.
- There is an ideal case so that tasks can be executed as soon as all the results from the predecessor tasks are ready and available. The system resources are inexhaustible in number, so the system can allocate the required number of resources to execute all the tasks parallel that are independent from each other.
- The system supports the collection of provenance data, therefore the intermediary results generated by the individual tasks are saved and in case of a failure they can be easily retrieved. Thus, there is no need to take checkpoints at the end of the tasks, and there is no need to take global checkpoints, since in the case of a failure only the effected task should be rolled back.
- The system also supports provenance data about failure statistics, so the probability of failures for a certain period of time is available for each resource component taking into account the aging factor as well.

4.3 Static Wsb algorithm

For our first order model, let us assume that the checkpointing cost does not change during execution and does not depends on the type of resource, so we denote it with C. We also assume that the fault-detection time is negligible, so $t_{f,j} = 0$ for all j, and we have only one type of resource. So, from now on, we omit the notations $t(T_i)_j$, $t_{f,j}$, $t_{s,j}$, $T_{C,j}$, $R_{i,j}$; we only use $t(T_i)$, t_f, t_s, T_C, R_i, respectively.

We also use the simplification, that when a failure occurs during checkpointing interval T_C, the rework time that is needed to recalculate the lost values is, on average, $\frac{T_C}{2}$. From this, it follows that the expected rework time that is needed to successfully terminate

the given task T_i can be expressed by:

$$E(R_i) = \sum_{j=1}^{\infty} P(Y = j) \cdot j \cdot \left(\frac{T_c}{2} + t_s \right), \tag{4.1}$$

where $P(Y = j)$ denotes the probability of having j failures during the execution of task T_i. With these assumptions, we can calculate the expected wallclock (total processing) time of a task T_i as:

$$E(W_i) = t(T_i) + \overbrace{\left(\frac{t(T_i)}{T_C} - 1 \right) \cdot C}^{checkpointing\ cost} + \overbrace{\sum_{j=1}^{\infty} P(Y = j) \cdot j \cdot \left(\frac{T_c}{2} + t_s \right)}^{Rework\ time}. \tag{4.2}$$

Thus, if critical errors (failures that do not allow for the further execution of a job) and program failures do not occur during the execution, then the expected execution time can be calculated using the above equation. According to the definition of the expected value for a discrete random variable, we get $E(Y) = \sum_{j=1}^{\infty} P(Y = j) \cdot j$. From the above equation, authors in (Di et al. 2013) derived the optimal number of checkpointing intervals (X_{opt}) for a given task:

$$X_{opt} = \sqrt{\left(t(T_i) \cdot \frac{E(Y)}{2C} \right)}. \tag{4.3}$$

If we assume that the failure events follow an exponential distribution, then we get that the optimal checkpointing interval during the execution of task T_i can be expressed by:

$$T_{copt} = \sqrt{(2CT_f)}, \tag{4.4}$$

where T_f is the mean time between failures. This equation was derived by Young (Young 1974).

We will use equation (4.2) as a starting point to calculate the checkpointing interval, in order to minimize the checkpointing overhead without affecting the total wallclock execution time of the whole workflow. In equation 4.2, the unknown parameter is the checkpointing interval; for W_i, we have an upper bound from the flexibility parameter of task T_i.

4.3.1 Large flexibility parameter

If flexibility parameter $flex[T_i] >> t(T_i)$, then this means that we have ample time to successfully terminate the task. Maybe the task could be successfully executed even more times. In this case, it is not worth pausing the execution to take checkpoints, but trying to execute it without any checkpoints. If failure occurs, we still have time to re-execute it. When there has already been more than one trial and no successful completion, then we should check the remaining time to execute the task without negatively affecting the total wallclock execution time. We would like to ensure that the task execution time does not affect the total execution time of the workflow (or only has an effect with probability p).

4.3.2 Adjusting the checkpointing interval

When the failure distribution is not known but we have a provenance database which contains the timestamps of the occurrences of failures for a given resource, then calculating the time that is needed to execute a task in the presence of failures with probability p is as follows:

If the mean time between failures is T_f, and we also have the deviance from provenance, then, with Chebyshev's inequality (4.5), we can determine the minimum size interval between the failures with probability p. This means that, with probability p, the failures do not happen within shorter time intervals

$$P\left(\left|\xi - T_f\right| \geq \epsilon\right) \leq \frac{D^2\xi}{\epsilon^2}. \tag{4.5}$$

We should find a valid ϵ for that $P\left(\left|\xi - T_f\right| \geq \epsilon\right) \leq 1 - p$ stands. If we have this ϵ, then we can calculate $T_m = T_f - \epsilon$ as the minimum failure interval with a probability greater than p. From this follows that, with probability p, there will not be more than $k = \frac{t(T_i)}{Tm}$ failures during the execution time of $T_{i,j}$. If we substitute this k into equation (4.2), we get an upper bound for the total wallclock execution time of the given task with k failures:

$$W_i = t(T_i) + (\frac{t(T_i)}{T_c} - 1) \cdot C + k \cdot \left(\frac{T_c}{2} + t_s\right). \tag{4.6}$$

If we use the optimal checkpointing for given task T_i with T_f mean time between failures (MTBF) and the deviance from this MTBF is ξ, then T_p gives the upper bound of the

wallclock execution time with probability p:

$$T_p = t(T_i) + \left(\frac{t(T_i)}{T_{copt}} - 1 \right) \cdot C + k \cdot \left(\frac{T_{copt}}{2} + t_s \right). \tag{4.7}$$

We henceforth assume that the failures do not occur during checkpointing and recovery (restarting and restoring the last-saved state) time, only during calculations.

If the flexibility parameter still permits some flexibility (i.e., $flex[T_i] > T_p$), then we can increase the checkpointing interval and so decrease the checkpointing overhead.

To calculate the checkpointing interval according to the flexibility parameter, we should substitute $flex[T_i]$ into W_i:

$$flex[T_i] > t(T_i) + \left(\frac{t(T_i)}{T_{cflex}} - 1 \right) \cdot C + \hat{k} \cdot \left(\frac{T_{cflex}}{2} + t_s \right). \tag{4.8}$$

We should fine T_{cflex} value for that (4.8) and $T_{cflex} > T_{copt}$ stands. However, we should also take into consideration, that in this case the expected number of failures k may be higher, so we denote it with $\hat{k}$.

From these inequalities, the actual T_{cflex} can be calculated easily.

If $W_i - T_p = 0$, the flexibility only allows us to guarantee successful completion with probability p.

However, if the flexibility parameter does not permit any flexibility (moreover, if $W_i < T_p$), then maybe the soft deadline cannot be guaranteed with probability p.

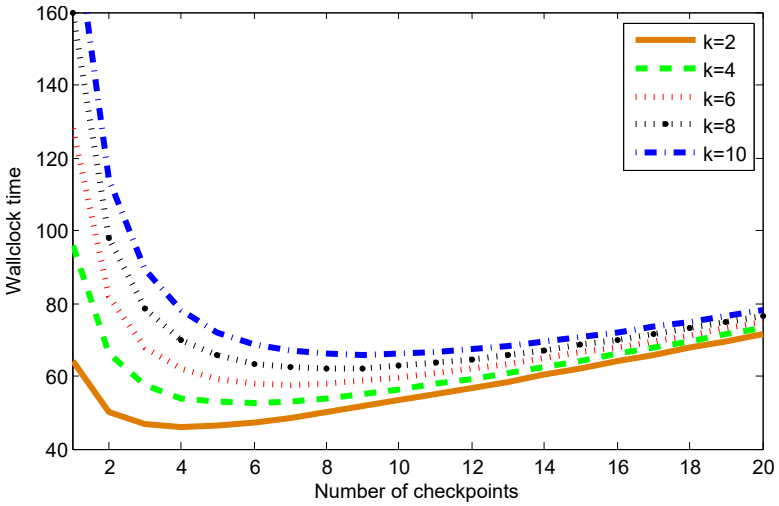

Figure 4.1: Total process time as a function of the number of checkpoints

4.3.3 Proof of the usability of the algorithm

According to (4.8), it is also numerically proven that the total execution time is a function of checkpointing interval T_c; or as it is indicated, a function of the number of checkpoints $n = \frac{t(T_i)}{T_c}$. As seen in Fig. 4.1, the dependency is quadratic. Fig. 4.1 shows five parabolas with a different number of failures (k values). All of the parabolas have minimum points, where the wallclock time of a task is minimal with an appropriate number of checkpoints. As k increases, the minimum points are shifted to the right. The dashed green line represents the curve with $k = 4$, where checkpointing cost $C = 2$ and calculation time $t(T_i) = 32$. This curve has its minimum points at four checkpoints $n = 4$. However, if we have time flexibility according to the curves in Fig. 4.1, we have the possibility of decreasing the number of checkpoints. In the case of the dashed green line, if we have four checkpoints, then the wallclock time reaches its minimum, while having only two checkpoints increases the total wallclock time. According to the flexibility parameter, an appropriate number of checkpoints can be determined, of course it should not decrease below the theoretical minimum, i.e.: $MTBF > T_{cflex}$ thus, it is possible to minimize the checkpointing overhead without increasing the total wallclock execution time of the workflow.

4.3.4 The operation of the Wsb algorithm

Our Static Wsb algorithm works as follows: before submitting the workflow, at first the optimal checkpointing interval should be calculated for each task based on some failure statistics of the resource(s) (expected value of the failures that can arise during execution) and the estimated (or retrieved from provenance database) execution time of the task. Concerning those tasks that are part of the critical path or one of the critical paths of the workflow the checkpointing interval should remain the optimal value. Than the adjusted checkpointing intervals for all the other tasks can be calculated. Wsb algorithm was planned to be a fair algorithm, it tries to share the flexibility parameter of the tasks equivalently. Thus starting from a flexible node it tries to decrease the number of the checkpoints equivalently between all nodes.

This algorithm is executed only once before submitting the workflow and after that the checkpointing intervals are not modified. It gives a workflow level, static solution for decreasing the number of checkpoints.

The exact operation of the Wsb algorithm can be seen on the flowchart diagram (Fig. 4.2):
As a first step the optimal checkpointing intervals (T_c) and the number of checkpoints

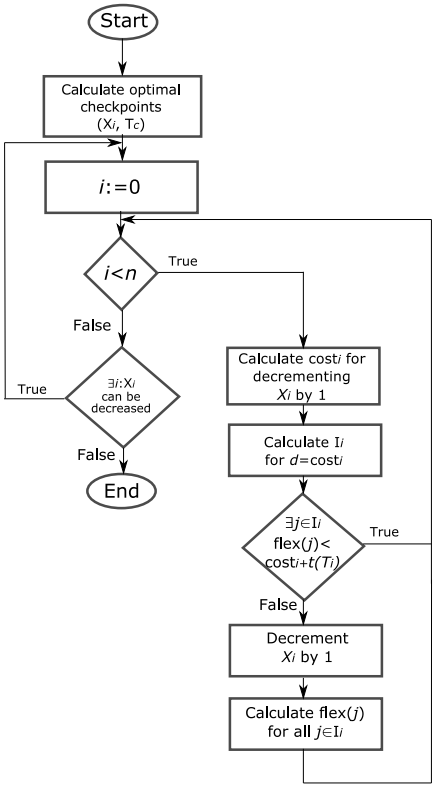

Figure 4.2: Chartflow diagram of the Wsb static algorithm

(X_i) are calculated for all tasks. Afterwards, while exists at least one task T_i for which the number of checkpoints can be decreased without negatively effecting the makespan of the workflow the algorithm evaluates for all nodes the possibility of decrementing.

It means the algorithm has to check whether the flexibility parameter of the task is higher than the predestined execution time plus the delay, and that all the nodes that are part of the influenced zone of this task T_i with this $cost_i$ as a delay, can absorb this delay, i.e.: the flexibility parameter of all these nodes is still higher than this $cost_i$. If yes, than the number of checkpoints is decreased for this task T_i and the flexibility parameter of the affected tasks are adjusted.

The algorithm proceeds until there exist at least one task, where the number of checkpoints can be decreased.

4.4 Adaptive Wsb algorithm

We talk about adaptive workflows when a workflow model can change during execution according to the dynamically changing conditions.

In chapter 3, we made calculations on the graphs that are based on prior knowledge obtained from previous enactments or estimations for runtime, communication, and data-transfer-time requirements. However, if the system supports provenance data storage and runtime provenance analysis, then we can base our calculations on realistic and up-to-date data. For example, if the precise timing of the task submissions that are under enactment and the precise completion time is known for all tasks that are already terminated, then the accurate flexibility parameter of the running tasks can be calculated, and a more-precise estimation of the influenced zones and the flexibility parameters of the successor tasks can be made available. Moreover, the estimated and the real values are generally not the same, so it would provide a more accurate timing. These calculations are always updated with newer and newer timing data but include less and less subgraphs with the advance of the execution steps. So, the remaining steps and calculations are getting simpler. Thus, if before workflow submission we calculate the flexibility parameters for the whole workflow, and we also store the estimated starting time of the individual execution times relative to each other, then before executing a task, its starting time should be updated to the new situation caused by the failures. Of course, depending on the delay, the flexibility parameters of all of the nodes belonging to the influenced zone of this task should be adjusted.

Based on these calculations, it is also possible to give a scientist more feedback about its workflow execution during enactment. For example, the researcher may get feedback on the probability of meeting soft or hard deadlines or whether the results will be outdated when the workflow execution terminates. So, it can be decided to stop the workflow, to modify the workflow, or to take other actions that are supported by the scientific workflow management system.

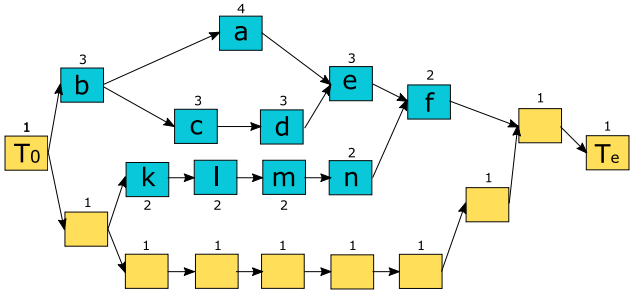

Figure 4.3: A sample workflow with homogeneous tasks

For the sake of simplicity, let us assume again that data transfer time is negligibly small in our examples (there are not any values assigned to the edges) and task execution time is 1 time unit for all tasks in Fig. 4.3. The critical path is built up from the yellow tasks

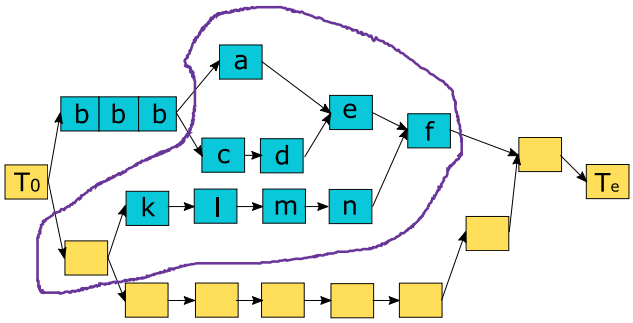

Figure 4.4: A two time-unit-long delay during execution of task b

before submitting the workflow. The flexibility parameters for all tasks are indicated in the figure above each task. two times the unit except for task a, where this value is three times the unit.

In Figure 4.4 during the execution of task b, a 2-time-unit-long failure has occurred. Since $flex[b] = 3$, $flex[a] = 4$ and $flex[c] = flex[d] = flex[e] = 3$, this 2-unit delay has not only local significance. This means that this delay will effect subsequent task a's, b's, c's, d's and f's submission time.

However, in that case when the failure is detected very soon, then maybe tasks k, l, m, and n do not have to be executed in a strict manner. The checkpointing interval should be recalculated for these tasks. The scope of this recalculation is the flexibility zone. In Figure 4.4 the flexibility zone is the set of enclosed tasks.

Definition 4.4.1. The flexibility zone in a workflow is a subworkflow of the original workflow, where flexibility parameters are changed due to a failure or time delay. ■

In other words the flexibility zone is a subgraph where changes in timing parameters can happen without affecting the total wallclock time of the workflow. The flexibility zone is always related to an influenced zone, thus, it is based on a certain delay interval. The border of a flexibility zone is always a sink task of the influenced zone and the tasks that belongs to the flexibility zone are on the paths that lead to this sink node. To determine the beginning of a flexibility zone is not straightforward.

The operation of the adaptive algorithm can be seen in Figure 4.5. The algorithm starts before each task submission. At first it evaluates whether for task T_i the difference $\delta_i$between the predestined and the real submission time is greater than a predefined threshold value ϵ_0. If yes, than it calculates the influenced zone I_i and the flexibility zone FZ_i concerning this delay, and determines the new flexibility parameters and number of checkpoints for all tasks T_j that are part of the flexibility zone FZ_i and not yet submitted.

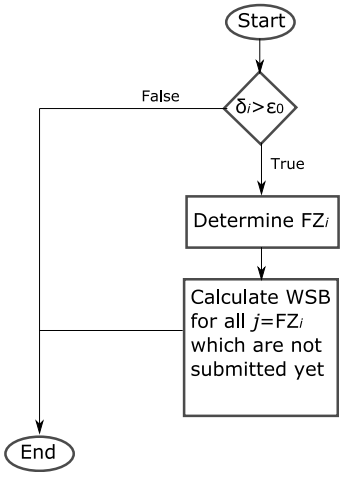

Figure 4.5: Chartflow diagram of the AWsb adaptive algorithm

4.4.1 Calculating the flexibility zone for complex graphs with high number of vertices and edges

It can be realized that flexibility zones are connected to cycles in the workflow graph when ignoring the orientation of the edges (regarding DAGs, we can only talk about cycles when we omit the orientation of the edges). More precisely, this is the case with subgraphs that contain several cycles interconnected with each other. To calculate the flexibility zones of a workflow model, we use the base of the algorithms published by Li et al. in (W.-N. Li, Xiao, and Beavers 2005). In this paper, the authors calculated the number of all topological orderings of a Directed Acyclic Graph.

DAGs are used to indicate a precedence relationship or relative ordering among the vertices. Given DAG $G(V, \overrightarrow{E})$, a topological ordering of G is a linear order of all vertices, which respects the precedence relation; i.e., if G contains edge (T_i, T_j), or with another notation $T_i \rightarrow T_j$, then T_i appears before T_j in the topological ordering. Concerning the graph demonstrated in Fig. 3.6, a possible topological ordering would be $\{a, b, d, e, c, f, g, h, i\}$, but the series $\{a, d, e, b, c, f, g, h, i\}$ also gives a valid ordering. As can be seen from the example, many topological orders may exist for a given DAG.

Lemma 4.4.1. *A G graph is a DAG if and only if it has a topological ordering.*

As a consequence of lemma, we know that every DAG has topological orderings.

The topological order of a DAG can be computed in many ways, but maybe the most-frequently-used method is applying a Depth-First Search (DFS).

For this purpose, the authors in (W.-N. Li, Xiao, and Beavers 2005) introduced the following concepts (which we also need in our calculations):

Definition 4.4.2. A static vertex is vertex T_i for which $|PRED(T_i)| + |SUCC(T_i)| = |V| - 1$ for given DAG $G(V, \vec{E})$ ■

The placement of a static vertex is deterministic, so it is the same in all existing topological orders.

Definition 4.4.3. Static vertex set $S \in V$ is a vertex set for which $|PRED(S)| + |SUCC(S)| = |V| - |S|$ for given DAG $G(V, \vec{E})$ and is minimal; that is, no proper subset of S has the same property. ■

In Li's work, the authors proved that these static vertex sets are disjoint.

According to these static vertex sets, a graph can be partitioned into disjoint static vertices and vertex sets.

Since the static vertex set means that the nodes or subset of these nodes can be in almost arbitrary order to each other, we may divide the vertex set into disjoint parallel threads of tasks. Thus, if a subgraph resulting from the algorithm is not simple enough, we can further use these algorithms after dividing the subgraphs into disjoint parallel threads. So, our algorithm can be recursively adapted until the desired depth.

As a result, the minimal flexibility zones of a workflow will be the union of those static vertex sets that cannot be further partitioned and consist the task where the failure occured and the border tasks of its effect. Sometimes these static vertex sets are not simple enough to determine the flexibility zone for a task. In this case we have to determine it, by going along all paths originating from all border task of the flexibility zone backwards until the tasks that are not submitted yet. This kind of searching algorithm has only O(n+e) time complexity. But the authors in (W.-N. Li, Xiao, and Beavers 2005) has also proved that for series-parallel digraphs a complete partitioning is possible. Thus this algorithm is efficient for small graphs or for series-parallel digraphs.

4.5 Results

For validation purposes, we have implemented both of our checkpointing algorithms in Matlab, a numerical computing environment by MathWorks.

4.5.1 Theoretical results

As our first simulation we validate our results from chapter 3 and show the relationship between the sensitivity property of a workflow and the improvement in the checkpointing cost. We have carried out simulation for two special graphs, namely for a most sensitive

Table 4.2: Simulation results for max. rigid and max flex. workflows

	$t(T_i)$	x_{opt}	x_{max_rigid}	x_{max_flex}
T_1	18	2	2	2
T_2	26	4	2	1
T_3	7	1	0	0
T_4	33	4	2	1
T_5	42	6	6	2
T_6	23	3	3	3
T_7	46	7	7	7
T_8	9	1	1	1
T_9	2	0	0	0
T_{10}	28	4	4	4
T_{11}	18	2	2	2

(fig. 4.7) and for a most flexible (fig. 4.6) one. The settings can be seen in table 4.2. The second column displays the calculation time for all tasks T_i for both scenarios. The third column includes the optimal number of checkpoints according to (Di et al. 2013), for the case when the expected number of failures $E(Y) = 2$ for an average of 18 time-unit-long task, and it is proportionally adjusted to other tasks. The checkpointing cost was set to $C = 2$. The fourth and fifth column list the decreased number of checkpoints based on our Wsb algorithm. It can be noticed that in the case of the maximal rigid workflow, this decrease is less (concerning tasks T_2, T_4, T_5). This fact is a simple consequence of the graph structure, because the time sensitivity parameter is $TS = \frac{4}{10}$ for both workflows, but for the maximal rigid case the sensitivity index for flexible nodes value $SIF = 0,333$, while it is $SIF = 0$ for the maximal flexible workflow. Thus in the former case the total improvement for the whole workflow is $\frac{5}{34}$, while in the latter one this value is $\frac{11}{34}$.

4.5.2 Comparing the Wsb and AWsb algorithms to the optimal checkpointing

To clarify the benefits of our static (Wsb) and adaptive (AWsb) algorithms we carried out simulations on a sample workflow model, (shown in Fig. 4.8) $G_{sample}\left(V, \vec{E}\right)$, where $V = \{T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8\}$ and
$\vec{E}= \{(T_1, T_2), (T_1, T_3), (T_3, T_4), (T_1, T_5), (T_5, T_6), (T_6, T_7), (T_2, T_8), (T_4, T_8), (T_7, T_8)\}$,

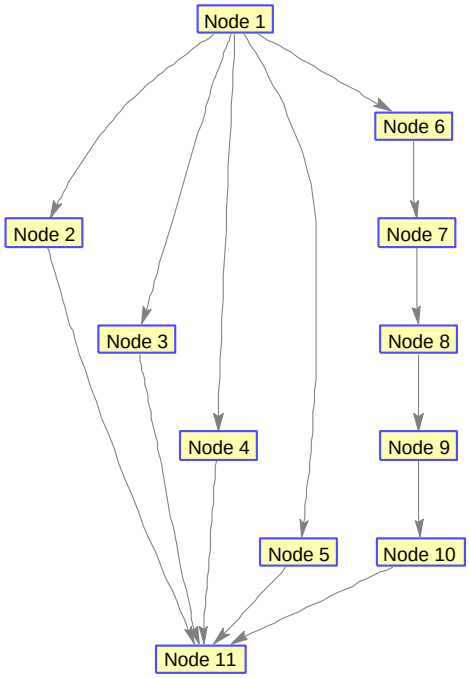

Figure 4.6: Most flexible workflow

running in a distributed environment, consisting of three resources: R_1, R_2, and R_3. For the sake of simplicity, the resources are identical and have identical failure distribution. We use $E(Y) = 2$ as the expected number of failures for an 18-time-unit-long task, and when changes occur during execution, this value is proportionally calculated to the changes. We also take advantage of the simplification that the data transfer times are negligibly small (they are all zeros) and the checkpointing cost has a constant value of $C = 2$. The workflow makespan (total wallclock time) is the longest path from T_0-T_e. We have simulated five scenarios with the same input parameters for our sample workflow:

1. optimal static case: Optimal checkpointing is used (Di et al. 2013) (T_{copt} is the optimal checkpointing interval, X_{opt} is the number of checkpoints, W_{orig} is the total execution time).

2. static execution with our static Wsb algorithm: In this case, the Wsb algorithm is executed once before workflow submission, which calculates the number of checkpoints based on the workflow structure ($X_{stat-wsb}$ is the number of checkpoints, $W_{stat-wsb}$ is the total execution time).

3. dynamic execution with optimal checkpointing: In this case, the execution time of a task is changed, but the execution is based on the original optimal checkpointing interval. (Optimal checkpointing interval T_{copt} is used, $W_{dyn-opt}$ is the total execution time).

4. dynamic execution with our static (Wsb) algorithm: In this scenario, the execution time of a task is changed, but the execution is based on static Wsb algorithm that was carried out before workflow submission; thus, before the change (the checkpointing interval is the same as in the static execution with the Wsb algorithm, $W_{dyn-wsb}$ the total execution time).

5. dynamic execution with our adaptive (AWsb) algorithm: In this case, the execution time of a task is changed, and the adaptive AWsb algorithm recalculated the checkpointing intervals after the change ($X_{dyn-awsb}$ is the number of checkpoints, $W_{dyn-awsb}$ the total execution time).

In the above-defined dynamic scenarios, there is only one task during each individual

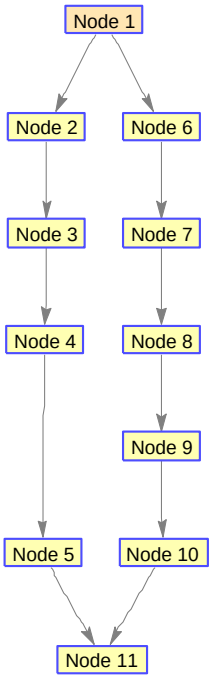

Figure 4.7: Most sensitive workflow

execution of the workflow; namely, T_3, for which the execution time is changed compared to the predestined values.

The simulation was carried out with $t(T_i) = 18$ and based on this value $T_{copt} = 6$, $X_{opt} = 3$ and thus $W_{i-orig} = 28$ was calculated for all tasks T_i, where W_{i-orig} is the total processing time of T_i when optimal checkpointing interval (T_{copt}) is used.

Table 4.3 shows the actual parameters for all tasks of the workflow for the static and adaptive cases. Table 4.4 compares the number of checkpoints and the total wallclock time for the whole worklfow for the five scenarios.

As the results show, our static algorithm reduces the checkpointing overhead by 18,75%, as the number of checkpoints were decreased from 16 to 13 with our algorithm, and the total wallclock time of the workflow did not change. We can also notice that, in a dynamically changing environment where the execution time for the tasks can change unpredictably, our adaptive algorithm may further increase the number of checkpoints but decrease the total wallclock time compared to dynamic execution with the static-algorithm scenario.

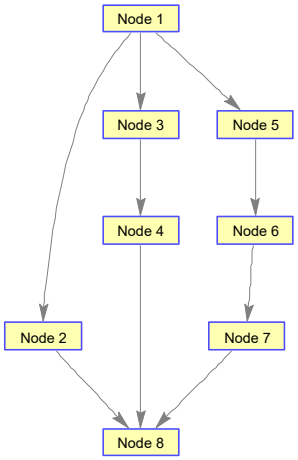

Figure 4.8: Sample workflow with 8 tasks.

4.5.3 Tests with random workflows

We have also carried out simulations with randomly formed DAGs. In these cases, the number of tasks has moved between 10 and 60 nodes, and calculation time $t(T_i)$ was randomly generated within the interval of (10,100). The checkpointing cost was increased during the simulations (Fig. 4.9), from $C = 2$ to $C = 14$, the expected number of faults was $E(Y) = 2$ for an average of a 45 time-unit-long task, and it was proportionally

Table 4.3: Simulation results for sample workflow (Fig. 4.8)

	$X_{stat-wsb}$	$t(T_i)_{dyn}$	$(W_i)_{dyn-awsb}$	$X_{dyn-awsb}$
$t(T_1)$	2	18	28	2
$t(T_2)$	1	18	36	1
$t(T_3)$	1	36	72	1
$t(T_4)$	1	18	28	2
$t(T_5)$	2	18	28	2
$t(T_6)$	2	18	36	1
$t(T_7)$	2	18	36	1
$t(T_8)$	2	18	28	2

Table 4.4: Comparison of number of checkpoints (X) and the total wallclock time (W) in the five scenarios

X_{opt}	$X_{stat-wsb}$	$X_{dyn-awsb}$	W_{orig}	$W_{stat-wsb}$	$W_{dyn-stat}$	$W_{dyn-awsb}$	$W_{dyn-opt}$
16	13	12	140	140	148	144	140

adjusted to the tasks according to their calculation time. Each point of the curve was averaged over 50 executions.

The results in 4.9 show a mild divergence, but they also show a significant decrease as a function of the checkpointing cost. It can be explained by the fact, that according to the optimal checkpointing interval (Di et al. 2013) when the checkpointing cost is higher, than system originally takes less checkpoints, thus the improvement in this case is lower. On the other hand we can see a significant improvement as the number of nodes constituting the workflow increases. This phenomenon can be explained by the consequences from chapter 3, because with higher number of tasks, a workflow can be less time sensitive and also more flexible. The sensitivity values from this scenario were validated by the simulations.

Our AWsb adaptive algorithm has also been tested with random graphs similar to the static case. As a consequence of the randomly generated workflows, the average difference between the total wallclock time of the dynamic execution with our Wsb algorithm case compared to dynamic execution with the AWsb scenario spread over a range of 0 % and 10 % improvement, and the number of checkpoints also shows a significant decrease in the latter case. So, we can conclude that the AWsb algorithm may decrease the checkpointing overhead to a further extent than the static Wsb algorithm while keeping the total processing time at its necessary minimum.

4.5.4 Remarks on our work

In our simulations, we have simplified the calculations by using constant values as checkpointing cost C by neglecting the data-transfer and task-submission times during the executions (or by assuming identical resources). Nevertheless, these assumptions can be easily resolved by substituting actual functions instead of using constant or simplified parameters.

The calculation time for complex graphs can be lengthy; but after a brief study at the myExperiment.org website, we have concluded that the mean size of the uploaded

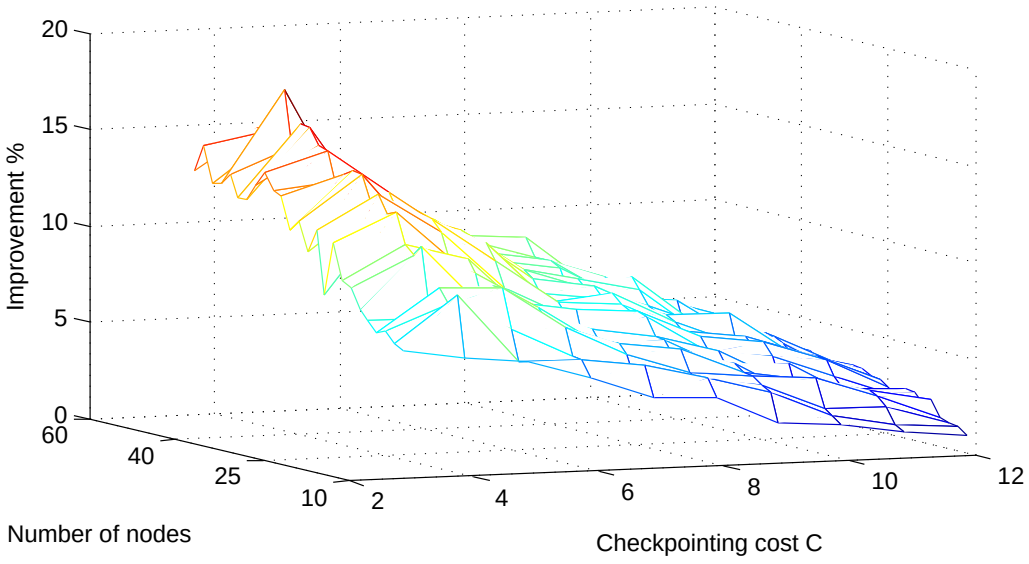

Figure 4.9: Results of our static algorithm

workflows moves between 30 and 50 nodes with manageable complexity. This revelation led us to develop the adaptive algorithm for which the recalculation time can be measured in hundreds of milliseconds.

Our algorithm cannot be used for an arbitrary type of failure or fault. It was intended to develop a mechanism against crash faults, or network outage. Of course, the proposed checkpointing method does not solve programming failures, byzantine failures, etc. in itself, as is the case with the optimal checkpointing strategy developed by Young (Young 1974) and Di (Di et al. 2013).

A further limitation of the algorithms lies in the fact that they depend highly on historical execution data or on estimated data about execution time and failure distribution. Data about historical executions can be stored in a provenance database; but today, there are only limited capabilities for runtime provenance analysis, and of course the estimations lack precision.

4.6 Conclusion and future work

In this chapter I have introduced my static (Wsb) and adaptive (AWsb) algorithms, which besides some failure statistics about resources are primarily based on the information that can be obtained from the workflow structure. With the help of these checkpointing methods the checkpointing overhead can be decreased while continually keeping the performance at a predefined level; namely, without negatively affecting the total wallclock time of the workflow. I also showed that the static Wsb algorithm can be adapted to a dynamically changing environment by updating the results of the workflow structure analysis. The simulation results showed that the checkpointing overhead can be decreased by as much as 20% with our static Wsb algorithm, and the adaptive AWsb algorithm may further decrease this overhead while keeping the total wallclock time at its necessary minimum. I have also showed the relationship between our worfklow structure analysis and the effectiveness of the checkpointing algorithms.

In the future this algorithm can be further developed to ensure the successful termination of the workflow with a probability of p. It is also integrated into our future plans to implement this algorithm in the gUSE/WS-PGRADE system.

4.6.1 New Scientific Results

Thesis group 2.: Workflow structure based checkpointing algorithm
Thesis 2.1:

Thesis 2.1

I have developed a workflow-level, periodic (Wsb) checkpointing algorithms for DAG based scientific workflows, which can be used with known, constant checkpointing costs and known failure rate. The algorithms decreases the checkpointing overhead compared to the checkpointing algorithm optimized for the execution time, without affecting the total wallclock time of the workflow.

Thesis 2.2:

Thesis 2.2

I have developed the adaptive version of the proposed Wsb algorithm, which may further decrease the checkpointing overhead in the case when the real execution and data transmission time encounter some difference compared to the estimated ones. In this case the algorithm may also decrease the execution time of the workflow compared to the static (Wsb) algorithm.

Relevant own publications pertaining to this thesis group: [K-5; K-7; K-9]

5 Provenance based adaptive execution and user-steering

From the scientist's perspective the workflow execution is like black boxes. The scientist submits the workflow and at the end, the result or a notification about failed completion is returned. Concerning long running experiments or when workflows are in experimental phase it may not be acceptable. Scientist need some feedback about the actual status of the execution, about failures and about intermediary results in order to save energy and time and to make accurate decisions about the continuation. Thus scientists need to monitor the experiment during its execution in order to fine-tune their experiments or to analyze provenance data and dynamically interfere with the execution of the scientific experiment. Mattoso et al. summarized the state-of-the-art and possible future directions and challenges (Mattoso et al. 2013). They found that lack of support in user steering is one of the most critical issues that the scientific community has to face with.

In this chapter we introduce iPoints, special intervention points, where the scientist or the user can interfere with the workflow execution, he or she can alter the workflow execution or change some parameter or filtering criteria. With these intervention points our aim was to enable already in the design phase of the workflow lifecycle to plan and insert intervention points. We also specified these iPoints in a language that was targeted to enable interoperability between four existing SWfMSs.

5.1 Related Work

5.1.1 Interoperability

In the past decay a lot of Scientific Workflow Management Systems have been developed that were designed to execute scientific workflows. SWfMSs are mostly bounded to one or more scientific discipline, thus they have their own scientific community and they all have their own language for workflow definition. While the business workflow community has developed its standardized, control-flow oriented language WS-BPEL Web Services Business Process Execution Language (WS-BPEL) (Jordan et al. 2007), the scientists

community has not accepted it due to the data and computation intensive nature of scientific workflows. Their definition languages are therefore mostly targeted at modeling the data-flow between the individual workflow tasks with a strong focus on efficient data processing and scheduling (Plankensteiner thesis), so languages like AGWL (Fahringer, Qin, and Hainzer 2005), GWENDIA (Montagnat et al. 2009), gUSE (Peter Kacsuk 2011), SCUFL (Turi et al. 2007) and Triana Taskgraph (Taylor et al. 2003) all belong to the data-flow oriented category.

Because of the different requirement that were addressed by the various scientific communities, it is widely acknowledged that the creation of a single standard language for all users of scientific workflow systems is a difficult undertaking that will probably not succeed in being adopted by all communities given the heterogeneous nature of their fields and problems to solve.

The SHIWA project (2010-2012) (Team 2011) was targeted to promote interoperability between different workflow systems by applying both coarse- and fine-grained strategy. The coarse-grained strategy (Terstyanszky et al. 2014) treats each workflow engine as distributed black boxes, where data being sent to preexisting enactment engines and results are returned. One workflow system is able to invoke another workflow engine through the use of the SHIWA interface, and the Shiwa Portal facilitates the publishing and sharing of reusable workflows. The fine-grained approach (Plankensteiner, Prodan, Janetschek, et al. 2013) deals with language interoperability by defining and Interoperable Workflow Intermediate Representation (IWIR) language (Plankensteiner, Montagnat, and Prodan 2011) for translating workflows (ASKALON, P-Grade, MOTEUR and Triana) from one DCI to another, thus creating a cross-compiler for workflows. The aim of the fine grained interoperability was to realize interoperability at at two abstraction levels (abstract and concrete) between four European scientific workflow management systems (MOTEUR developed by the French National Center for Scientific Research (CNRS), ASKALON (Fahringer, Prodan, et al. 2007) from the University of Innsbruck, WS-PGRADE (Peter Kacsuk et al. 2012) from the Computer and Automation Research Institute, Hungarian Academy of Sciences MTA SZTAKI), and Triana (Taylor et al. 2003) from Cardiff University.

5.1.2 User-steering

As it was already mentioned in section 2 in the literature there exist several solutions to support dynamism at different granularity (dynamic resource allocation, advance and late modeling techniques, incremental compilation techniques, etc.). Obviously, most of them relates on monitoring the workflow execution or the state of the computing resources.

However, monitoring from the scientist's perspective is also very important, moreover, data analysis and dynamic intervention is also an emerging need concerning nowadays scientific workflows (Ailamaki 2011). Due to their exploratory nature they need control and intervention from the scientist to conserve energy and time.

There are several systems that support dynamic intervention such as stopping, or re-executing jobs or even the whole workflow but there is an increasing need to have more sophisticated manipulation possibilities. Vahi et al. (Vahi, Harvey, et al. 2012) introduced Stampede, a monitoring infrastructure that was integrated in Pegasus and Triana and which main target was to provide generic real-time monitoring across multiple SWfMSs. The results proved that Stampede was able to monitor workflow executions across heterogeneous SWfMSs but it required the scientists to follow the execution from a workstation. This solution may be tiring concerning long-term executions. To tackle this, it is possible to pre-program triggers, such as proposed by Missier et al. in (Missier et al. 2010), to check for problems in the workflow and to alert the scientist. In an other paper by Pintas et al. (Pintas et al. 2013) worked out sciLightning, a system that is able to notify the scientist upon completion of certain, predefined events. In their work (Dias et al. 2011) authors managed to implement dynamic parameter sweep workflow execution where the user has the possibility to interfere with the execution and change the parameter of some filtering criteria without stopping the workflow.

Oliveira et al. in (Oliveira et al. 2014) considered three types of unexpected execution behavior; one is related to execution performance, the second is aware of the workflow stage of execution and the third is related to data-flow generation, including domain data analysis.

Execution performance analysis during runtime is already integrated in several solutions:

In their paper (K. Lee, Paton, et al. 2009) the authors describe an extension to Pegasus whereby resource allocation decisions are revised during workflow evaluation, in the light of feedback on the performance of jobs at runtime. Their adaptive solution is structured around the MAPE (K. Lee, Sakellariou, et al. 2007) functional decomposition which is a useful framework for systematic development of adaptive systems, and can be applied in a wide range of applications, including different forms to workflow adaptation. Domain data analysis during execution time may prevent generating anomaly when for example unexpected data was consumed by a job thus producing unexpected results (Oliveira et al. 2014).

5.1.3 Provenance based debugging, steering, adaptive execution

Besides monitoring, debugging is essential for workflows that execute in parallel in large-scale distributed environments since the incidence of errors in this type of execution is high and difficult to track. By debugging at runtime, scientists can identify errors and take the necessary actions, while the workflow is still running. Costa et al. in their paper (Costa et al. 2013) investigated the usefulness of runtime generated provenance data. They found that provenance data can be useful for failure handling, adaptive scheduling and workflow monitoring. Based on PROV recommendation they created their own data modeling structure.

Concerning the volume of provenance data generated at runtime another challenging research area is provenance data analysis concerning runtime analysis.

Authors in (Dias et al. 2011) analysed the importance and effectiveness of provenance based debugging during executions and showed that debugging is essential to support the exploratory nature of science, and that large-scale experiments can benefit from it from a time and financial cost saving perspective.

However, most of the existing solutions for dynamism provide limited range of changes, that have to be scheduled a-priori and they do not solve on-the-fly modification of parameter sets, data sets or the model itself. On the other hand adapting the workflow execution to runtime provenance data analyses still remained a challenge.

5.2 iPoints

To implement workflow manipulation we created and defined a new, dynamic workflow control mechanism based on Intervention Points (iPoints) to enable provenance based adaptive and user-steered workflow execution, which is able to modify the execution according to provenance data or intermediary results and to adapt it to environmental changes. With the use of iPoints during enactment the user can take over the control for a while and has the opportunity to restart and stop the workflow execution, to insert time management functions, or based on provenance and runtime intermediary data the user can change certain parameters, filtering criteria or the input dataset. Furthermore with the insertion of a checkpoint the user can also change the execution model of the running workflow.

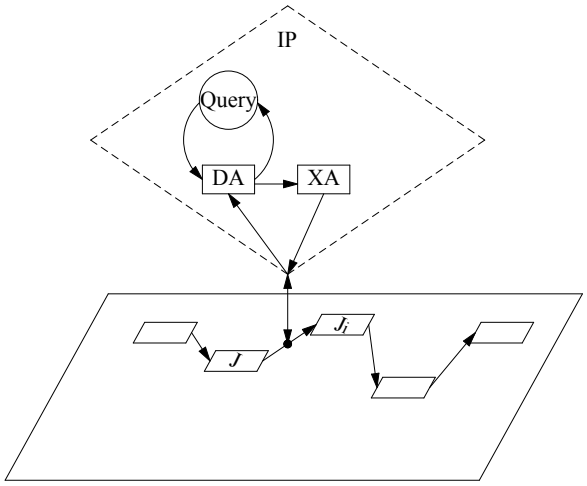

Figure 5.1: An iPoint

5.2.1 Structure and Functionality of an iPoint

An iPoint is somewhat similar to a meta-workflow or sub-workflow. It is located out of the plane of the workflow. In 5.1 the big rectangle with solid line represents the workflow plane, where the small rectangles on it represent the jobs and the sequential or parallel mesh of the user defined jobs form the scientific workflow. The iPoint which can be imagined or handled as a special job jumps out from this plane. It contains series of steps that are not part of the computational tasks (bordered with dashed line in 5.1) defined by the scientist rather which can affect the real execution (analysis of data or controlling functions, etc.).

The iPoint consists of a Designator Action (DA), a decision mechanism, which can be a predefined Rule Based Engine (RBE) or the scientistś on the fly decision and an eXecutable Action (XA). During the execution of an iPoint first a designator action (DA) is performed. The output of this DA designates or determines what changes are necessary during the execution. The DA can be one of the following actions: intermediary data query, provenance data query, provenance data creation and time management functions.

After an input reply is returned and based on this reply an eXecutable Action (XA) is performed and the process of iPoint terminates.

This XA can be one of the following possibilities: modifying the workflow model with checkpoint request, restarting or stopping the workflow execution, changing certain

parameters, filtering criteria or input dataset or requesting a checkpoint. Of course, the scientist has the possibility to perform more queries.

5.2.2 Designator Actions (DA)

1. Provenance data query: During the execution after termination of a job or sub-workflow, the scientist might need to analyze provenance data or partial results and make changes in the data sets to be processed, changes in parameter values or filtering criteria (for example iterative experiments). Interrupting the execution of activities or even stopping the complete or partial workflow execution is also a necessary action in several cases.

2. Provenance data creation: In certain cases the scientists may customize the provenance data captured during execution time. There exist some scientific workflow management system that support provenance capturing at various granularity levels. In these systems the scientist can decide and select the appropriate level during configuration phase.

3. Time management: At given points of the workflow execution the user might need to use some time management functions, for example to start, to stop, to check, to reset a timer or to set an alarm.

5.2.3 eXecutable Actions (XA)

To determine all the possible XAs that our framework should support at first we have investigated the requirements of the different scientific communities and we summarized it into the following list.

Management commands:

1. Delete a workflow, a subworkflow, a job, a link or an input or output port.

2. Create a link (Only links can be created between existing ports or jobs).

3. Stop a workflow, a subworkflow or a job.

4. Start a workflow, a subworkflow or a job.

5. Carry out a Modified Start for a workflow, a subworkflow or a job

6. Insert an iPoint (The iPoint can be inserted before or after a job, and after an input or output port).

Time Management commands: start, stop, check, reset a timer and to set an alarm.

5.2.4 Types of iPoints

According its operation we have differentiated four types of iPoints:

1. Closed iPoint: When the conditions and the changes are also known before execution, then the user can design the location and the function of the iPoints during the composition phase. He can also define the proper provenance queries, or the time management functions and depending on the results the action (XA) that should be carried out. So in this case the user do not have to interfere, all the actions are clearly defined, that is why we call it closed iPoint. The only difference from using a simple if-then-else structure in the composition phase is that with the closed iPoint the user can obtain provenance information from the Provenance Database during execution.

2. Open iPoint: When the conditions or the changes are not known before execution, because they can only be specified depending on the provenance query results, then the user should only determine the places where the intervention should take place. With these iPoints the user can interrupt the workflow execution for a while. During this predefined time interval the user can decide what DA to take, and depending on the results what XA to perform next. If the user does not interfere within the predefined time interval, then the control is given back to the original execution.

3. Dynamic iPoint: When the user do not have the opportunity to use closed IP (because he or she cannot give a definite, unambiguous correlation between the condition and the actions to take), and do not have the possibility to interfere during execution, then he or she can define a Rule Based Engine (RBE). The Rule Based Engine determines that depending on the outcome of the DA what XA should be performed and with what priority. During enactment phase, when the workflow execution arrives at the dynamic IP point, after the DA the system investigates the RBE and determines the next XA to take according to the defined priorities The Rule Based Engine should be defined before enactment. It can be specified by the scientist, for example if one task is consuming more time to execute than expected (e.g. more than average execution time), there is an indication of alert, and an alternative execution possibility should be selected from provenance

database. However, the RBE may include task execution features connected to scientific content. When data mining support exist it can be updated by the SWfMS adaptively, thus, realizing a provenance based adaptive execution. The workflow developer or the scientist should determine the priority of each eXecutable Action depending on the outcome of the provenance query.

4. Ad-hoc iPoint: When the demand for interfering arises only during execution upon some kind of external effect (for example computational or other failure, or because from an input sensor unwanted data arrives or simply because the user need some intervention) which cannot be foreseen before execution. In this case by using a special signal the system could stop the workflow execution at the nearest possible place, it should perform a checkpoint (if necessary) and give the control to the user for a while. At this point the user (like in the previous cases) may perform some query for intermediary results and depending on the outcome of the query certain XA can be carried out.

5.2.5 The Placement of iPoints

The iPoints can be scheduled upon four various events: after data arrival at input port, after data arriving at output port fig. 5.2, before job starting, after job completion fig. 5.3. However, a trigger event of an iPoint also can be: timer expiration, external effect, failure message arrival, timed alarm. In this case the placement happens in an ad-hoc manner at the nearest possible moment during the execution.

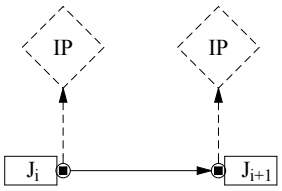

Figure 5.2: iPoint placement before data arrival or after data producement

5.2.6 iPoint Language Support

A dynamic system that supports user intervention must provide the user with language tools to define intervention points along with XA actions. In this case the iPoint is

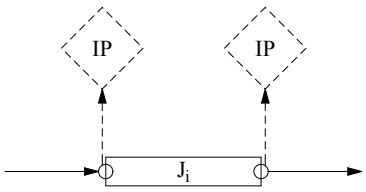

Figure 5.3: iPoint placement before submission, or after completion

an if Query = X then „Action1 else Action2" statement or time management function. At abstract workflow level the iPoint is a special job, which can be visualized with a pentagon or hexagon (fig. 5.4 and 5.5) The figures show the course of the execution steps involving an iPoint.

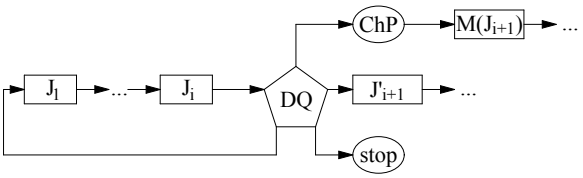

Figure 5.4: Process of Provenance Query

In fig. 5.4 the iPoint performs some kind of provenance query or partial data analyses, and depending on the result the workflow can be stopped, restarted, the execution model can deviate from the original model or even a checkpoint can be performed. In fig. 5.5 time management functions are inserted into the model.

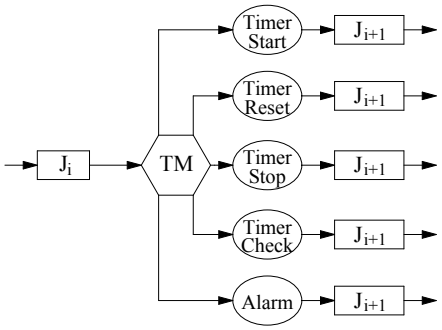

Figure 5.5: Process of Time Management

5.2.7 Benefits of using iPoints

Debugging mechanisms in HPC scientific workflows are essential to support the exploratory nature of science and the dynamic process involved in scientific analysis. Large-scale experiments can benefit from debugging features to reduce the incidence of errors, decrease the total execution time and sometimes reduce the financial cost involved. A prime example for this was proved in (Oliveira et al. 2014), where unexpected program errors were discovered. The user received an unexpected error message about a problem where users were unable to determine the specific details of investigating a receptor structure a-priori. With the help of provenance queries during runtime the problem could be detected and solved. Authors also reported 3% of time saving and the succesful re-execution of failed workflows caused by this error. Debugging without provenance would be more time consuming. When all these relationship could be discovered automatically with a strong data mining tool and stored in the RBE, the changes in the execution would be carried out adaptively based on the RBE.

Also provenance based adaptive execution would result in less failed execution and time savings when using task execution time information, from historic provenance, system would be able to identify performance variations that may indicate execution anomalies. If one task is consuming more time to execute than expected (e.g. more than average execution time), then the system would change the settings and task like this would be submitted to a more reliable one.

The implementation of the iPoint can be realized with a Scientific Workflow Manager

independent module that handles the actions taking place during the interventions. This module takes over the control of the workflow while the actions defined in the given iPoints are executed. This module can be an extension of an existing workflow management system, or a completely new system. In this latter case there is no need to change the existing SWfMS, only an interface should be specified and implemented.

5.3 IWIR

The above introduced iPoints were planned to be an extension of the IWIR language (Interoperable Workflow Intermediate Representation) (Plankensteiner, Montagnat, and Prodan 2011), which was developed within the framework of the SHIWA (Team 2011) project.

5.3.1 IWIR Introduction

The IWIR language is a representation that was targeted to be a common bridge for translating workflows between different languages independently from the underlying Distributed Computing Infrastructure (DCI). The figure 5.6 displays the architecture of the fine-grained interoperability realized in the Shiwa project.

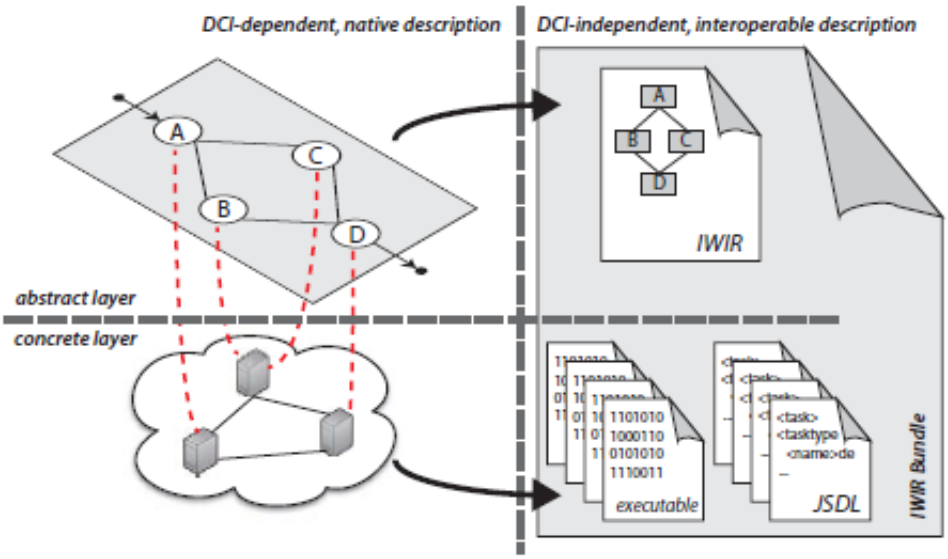

Figure 5.6: Abstract and concrete layers in the fine-grained interoperability framework architecture. (Plankensteiner 2013)

The abstract level defines the abstract input/output functionality of each workflow task

(the task signature) and the workflow-based orchestration of the computational tasks, defining the precedence relations in terms of data-flow (and control-flow) dependencies.

The concrete part of a workflow application contains low-level information about its computational tasks' implementation technologies. For example how to execute a certain application on a certain resource, where and how to call a certain web service, or even an executable binary file, representing the computational task itself. The type and form of information contained in the concrete part of the workflow is often specific to a certain workflow system and DCI.

IWIR is an XML- and graph based representation enriched with sequential and parallel control structures already known from programming languages. Due to its original objective to enable portability of workflows across different specification languages, workflow systems and DCIs, the IWIR language decouples itself from the concrete level by separating the computational entities from specific implementations or installations details through a concept called Task Type. It does not define ways to manipulate data, instead in an abstract level it only provides means to effectively distribute data to the computational tasks, that do the data manipulation (Plankensteiner, Montagnat, and Prodan 2011).

5.3.2 Basic building blocks of IWIR

An IWIR workflow has a hierarchical structure; it consists of exactly one top-level task, which may contain an arbitrary number of other tasks as well as data- and control-flow links. This top-level task forms the data entry and exit point of a workflow application. An IWIR document structure can be seen in listing 5.1.

Listing 5.1: IWIR document structure

```
1 <IWIR version =" version " wfname =" name ">
2 <task ... >
3 </IWIR >
```

The *IWIR version* is the actually used version of the IWIR language specification. *IWIR wfname* is the IWIR workflow name which serves as the identification of the workflow.

A task can either be an *atomic* task, which is a single executable computational entity or a compound task which consists of a set of atomic or other *compound* tasks with their data dependencies. A Task type is composed of a *type name* and a set of *input* and *output port*s with corresponding data types. The source of input data and the storage of output data being workflow management system specific is not defined in IWIR. Between the

tasks links can be created by defining the from *task/port* and the to *task/port* attributes (listing 5.2).

Listing 5.2: Link

```
1 <links >
2          <link from =''from''  to =''to'' >
3 </links >
```

The *from* attribute of a link defines the source of the data flow connection. The *to* attribute of a link defines the destination of the data flow connection. In IWIR, this attribute is specified in the form *of task/port*, where task is the name of the task and port is the name of the data port consuming the data. The data type of the data port specified in the *from* attribute has to match the data type of the port referred to in the *to* attribute.

In IWIR it is also possible to define control flow dependencies without any data dependency. It can be expressed by giving the appropriate task names without the input and output ports names.

An *Atomic Task* is a task which is implemented by a single computational entity and can be seen in listing 5.3. It may have several *input* and *output* ports.

Listing 5.3: Task

```
<task name =" name" tasktype =" tasktype">
       <inputPorts >
            <inputPort name =" name" type =" type "/>*
            ...
       </inputPorts >
       <outputPorts >
            <outputPort name ="name" type =" type "/>*
            ...
       </outputPorts >
</task >
```

IWIR defines its built in data types as *integer, string, double, boolean,* a *file* and a *collection type* which can be a multidimensional ordered, indexed list. IWIR have two types of predefined *compound tasks*: *Basic Compound Tasks: blockScope, if, while, for, forEach* and *Parallel Compound Tasks: ParallelFor, parallelForEach.* These latter one was targeted to express loops whose iterations can be executed concurrently.

5.4 Specifications of iPoints in IWIR

There are several solutions in already existing SWfMS to support the modification of the workflow execution by the use of breakpoints. For example, in gUSE (Gottdank 2014) at the workflow configuration phase users can insert breakpoints into workflow executions. These breakpoints are very similar to that are used in programming languages. The execution is paused at these points and the user can stop, restart or alter his workflow execution. However, these modifications are only done at concrete workflow level, so the original workflow model is not changed accordingly. This problem is the workflow evolution problem. To solve this problem when user or administrator interferes with the workflow execution their changes modify also the original IWIR file and map a new workflow version number to this file, which serves as an identification of the actually used version of the workflow, and as a support to track workflow evolution. So according to our first extension to IWIR is to append a *wf_version* to the IWIR document.

Stemming from the above described workflow evolution problem we make even more strict the hierarchical structure of an IWIR document. In order to make it easier to follow the changes and to determine the border of its scope it is required from an IWIR workflow to be built up from subworkflows (compound tasks). iPoints can be inserted only at the border of this subworkflows. So the changes described in the iPoint can only refer to a given subworkflow of it.

5.4.1 Provenance Query

As a further extension we introduce some atomic task description into IWIR, namely the Designator Action (DA) of the iPoint. As it was mentioned above a Designator Action can be a Provenance Query, a Provenance entry creation and a Time management function. The *provenance query* atomic task can be seen in listing 5.4. The only difference from a simple atomic tasks, is the input port type is string where an SQL query (SELECT ... FROM ...) is received and then the task frowards it to the provenance database. The *provenance entry creation* can be similarly specified.

Listing 5.4: Task

```
1 <task name =" name " tasktype =" Prov_query ">
2   <inputPorts >
3     <inputPort name =" query " type =" string"/ >*
4     ...
5   </ inputPorts >
7   <outputPorts >
7     <outputPort name =" query_res " type =" type "/ >*
```

```
9    </ outputPorts >
10 </task >
```

5.4.2 Time management Functions

In order to provide *time management* functions such as start, stop, check or reset a timer, and set an alarm the language should support a time-like data type. So we extend the predefined list of datatypes with a *date type*. The time management functions should be defined also as predefined *atomic task*, the planned IWIR specification of a *timer check task* is shown in listing 5.5.

Listing 5.5: Timer check

```
<task name =" name" tasktype =" timer check">
  <inputPorts >
    <inputPort name ="timer_ID" type =" integer "/>
    ...
  </inputPorts >
  <outputPorts >
      <outputPort name ="time_elapsed" type =" integer "/>
      ...
  </outputPorts >
</task >
```

5.4.3 eXecutable Actions

In our specification the eXecutable Actions can also be realized as special *atomic tasks*. As an example the specification of the *Delete atomic task* can be seen in listing 5.6. The *Delete atomic task* can delete what is determined at its input port. As its input port any object can be addressed that has a unique ID in a subworkflow and is involved in the remaining subworkflow. For example a task, a link between tasks, a port (with a corresponding link) or even a whole subworkflow. Also the workflow name should be specified as an input parameters that involves this object and a new version should be specified for the resulting workflow.

Listing 5.6: Delete task

```
1 <task name =" name " tasktype =" delete ">
2   <inputPorts >
3     <inputPort name =" ID " type =" integer"/ >
4     <inputPort name =" wf_name " type =" string"/ >
5     <inputPort name =" wf_version " type =" integer"/ >
```

```
6       ...
7    </ inputPorts >
8    <outputPorts >
9      <outputPort name =" name " type =" type "/ >*
10     ...
11   </ outputPorts >
12 </task >
```

5.4.4 The iPoint compund tasks

The IWIR specification of iPoints are compound tasks which can consist of DAs, XAs, and if conditionals. The *closed iPoint* is presented in Listing 5.7. At least one input port must be defined, where the provenance_query as a string should be specified. The body consists of a DA (a provenance query) and an 'if' task.

There is only a little difference between closed and dynamic iPoints, namely that in dynamic iPoint after the provenance query another query takes place before the 'if' structure, which queries the Rule Based Engine. The open iPoints are similar to the breakpoints, so in this case the IWIR specification can be an atomic task, which causes workflow execution to pause, and whatever the user or administrator does is then inserted into the original workflow model and saved with unique ID for future execution tracking.

Listing 5.7: Closed iPoint

```
1 <task name =" name " tasktype =" closed_iPoint ">
2    <inputPorts >
3      <inputPort name =" name" type =" string "/ >*
4    </ inputPorts >
5  <body>
6   <task name=âĂİprov_queryâĂİ tasktype=âĂİprov_queryâĂİ>
7      <inputPorts >
8        <inputPort name =" prov_query" type =" string "/ >*
9      </ inputPorts >
10     <outputPorts >
11       <outputPort name =" query_res" type =" type "/ >*
12     </ outputPorts >
13   </task>
14   <if name =" name">
15      <inputPorts >
16         <inputPort name =" name" type =" type "/>*
17      </inputPorts >
18     <condition > condition </condition >
19     <then >
20       <task name=" XA1"  tasktype=" delete" >
```

```
21          <inputPorts >
22            <inputPort name =" name" type =" type "/ >*
23          </ inputPorts >
24          <outputPorts >
25            <outputPort name =" query_res" type =" type "/ >*
26          </ outputPorts >
27       </task>
28      </then >
29      <else >
30       < task name=" XA2"  tasktype=" modification" >
31          <inputPorts >
32            <inputPort name =" prov_query" type =" string "/ >*
33          </ inputPorts >
34          <outputPorts >
35            <outputPort name =" query_res" type =" type "/ >*
36          </ outputPorts >
37       </task>
38      </else >
39     <outputPorts >
40       <outputPort name =" name" type =" type "/>*
41     </outputPorts >
42     <links >
43       <link from =" from" to="to" />*
44     </links >
45    </if >
46  </body>
47      <outputPorts >
48        <outputPort name =" name" type =" string "/ >*
49      </ outputPorts >
50      <links >
51        <link from =" prov_query / query_res " to ="if/ prov_res " />*
52      </links >
53 </task >
```

5.5 Conclusion and future directions

In this chapter I have proposed a new dynamic workflow control mechanism based on Intervention Points (iPoints). With the help of the introduced intervention points and system monitoring capabilities adaptive and user steered execution can be realized at different level. Furthermore, when the system supports (runtime) provenance analysis, with the help of these iPoints provenance based, adaptive execution can be realized. Originally, the iPoints were planned to solve the problem of user-steering, but the

introduction of dynamic iPoints with a Rule Based Engine enables provenance based adaptive execution. The Rule Based engine may define special anomalies or coexisting features that require the modification of the execution. Data mining can also support the RBE based control. The administrator can also insert them to realize provenance based adaptive fault recovery or even system optimization tasks.

I also gave a specification for these iPoints in IWIR language that was targeted to solve interoperability between four existing SWfMSs. With this specification I created the possibility to plan and to insert these intervention points already in the design phase of the workflow lifecycle. Furthermore, the selected language promotes the widespread usage of this iPoint because among the IWIR enabled SWfMs it is enough that only one SWfMS is capable of executing iPoints.

In the future we intend to implement these iPoints into the gUSE/WS-PGrade system, where an gUse-IWIR interpreter is already exists.

5.5.1 New Scientific Results

Thesis group 3.: Provenance based adaptive and user-steered execution
Thesis 3.1:

> ***Thesis 3.1***
> **I have defined special control points, (iPoints), with the help of which and based on provenance analysis real-time adaptive execution of scientific workflows can be realized.**

Thesis 3.2:

> ***Thesis 3.2***
> **I have defined special control points, (iPoints), with the help of which real-time user-steered execution of scientific workflows can be realized.**

Thesis 3.3:

> ***Thesis 3.3***
> **I have specified the control points introduced in *thesis 3.1 and 3.2* in an Interoperable Workflow Intermediate Representation (IWIR) language.**

Relevant own publications pertaining to this thesis group: [K-2; K-4; K-7; K-8]

6 Conclusion

Scientific workflows are widely accepted tools to model and to orchestrate in silico scientific experiments. Due to the data and compute intensive nature of scientific workflows they require High Performance Computing Infrastructures to be executed in order to successfully terminate in a reasonable time. These computational resources are highly error-prone thus dynamic execution environment is indispensable.

In my Phd research work I have investigated the different aspects of dynamism. I have studied the dynamic support the Scientific Workflow Management Systems provide and concluded that fault tolerance is an ever green research field concerning scientific workflow execution. In the field of fault tolerance I came across the most widely used proactive fault tolerant mechanisms, such as replication and checkpointing and I have noticed, that while in scheduling and time estimation problems workflow structure is often involved in heuristics, fault tolerance is generally based on the properties of the computing resources and failure statistics.

Focusing on the aim to fill this gap in my first thesis group I investigated the workflow structure from a fault tolerant perspective. I have introduced the influenced zone of a failure, and based on this concept I have formulated the sensitivity index of a workflow model. Investigating the possible values of this index I have classified the workflow models.

Based on the results obtained from the first thesis group, in the second thesis group I have developed a novel, static Wsb checkpointing algorithm, which decreases the overhead of the checkpointing compared to a solution that was optimized for the execution time when the checkpointing cost and the expected number of failures is known, without increasing the total wallclock time of the workflow. With simulation results I have pointed at the relationship between the sensitivity index and the performance of the Wsb checkpointing algorithm. I have also shown that this algorithm can be effectively used in dynamically changing environment.

In the third thesis group I have turned my attention to a recently emerged issue of dynamism, namely the provenance based adaptive execution and user steering. I have introduced special control points to enable adaptive execution and user intervention

based on runtime provanance analysis. I also gave the specification of these control points in an Interoperable Workflow Intermediate Representation (IWIR) language. With this specification I have further promoted workflow interoperability, because among the IWIR enabled workflows it is only enough to have one SWfMS that is capable of handling these iPoints.

Pertaining to this thesis several open challenges remained that should be addressed. First of all the further development of our checkpointing algorithms into a task level adaptive one should be considered. Upon monitoring the failures during task execution, if too many errors have already encountered then it may necessary to change the frequency of the checkpointing according to the time constraint derived from the workflow structure. Furthermore, the implementation of the proposed schemes are planned into the gUSE/WS-PGRADE system. Also in the provenance based adaptive execution and user steering topic has left several open challenges, which should be preceded by prototyping the solution into a real system.

Bibliography

References

Agarwal, Saurabh, Rahul Garg, Meeta S Gupta, and Jose E Moreira (2004). "Adaptive incremental checkpointing for massively parallel systems". In: *Proceedings of the 18th annual international conference on Supercomputing*. ACM, pp. 277–286.

Ailamaki, Anastasia (2011). "Managing scientific data: lessons, challenges, and opportunities". In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, pp. 1045–1046.

Alsoghayer, Raid Abdullah (2011). *Risk assessment models for resource failure in grid computing*. University of Leeds.

Altintas, Ilkay, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludascher, and Steve Mock (2004). "Kepler: an extensible system for design and execution of scientific workflows". In: *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*. IEEE, pp. 423–424.

Bahsi, Emir Mahmut (2008). "Dynamic Workflow Management For Large Scale Scientific Applications". PhD thesis. Citeseer.

Balasko, Akos, Zoltan Farkas, and Peter Kacsuk (2013). "Building science gateways by utilizing the generic WS-PGRADE/gUSE workflow system". In: *Computer Science* 14.2), pp. 307–325.

Benabdelkader, Ammar, Antoine AHC van Kampen, and Silvia D Olabarriaga (2015). *PROV-man: A PROV-compliant toolkit for provenance management*. Tech. rep. PeerJ PrePrints.

Chandrashekar, Deepak Poola (2015). "Robust and Fault-Tolerant Scheduling for Scientific Workflows in Cloud Computing Environments". PhD thesis. Melbourne, Australia: THE UNIVERSITY OF MELBOURNE.

Chang, Duk-Ho, Jin Hyun Son, and Myoung Ho Kim (2002). "Critical path identification in the context of a workflow". In: *Information and software Technology* 44.7, pp. 405–417.

Chen, Xin, Charng-Da Lu, and Karthik Pattabiraman (2014). "Failure analysis of jobs in compute clouds: A google cluster case study". In: *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, pp. 167–177.

Costa, Flavio, Vítor Silva, Daniel De Oliveira, Kary Ocaña, Eduardo Ogasawara, Jonas Dias, and Marta Mattoso (2013). "Capturing and querying workflow runtime provenance with PROV: a practical approach". In: *Proceedings of the Joint EDBT/ICDT 2013 Workshops*. ACM, pp. 282–289.

Cruz, Sérgio Manuel Serra da, Maria Luiza M Campos, and Marta Mattoso (2009). "Towards a taxonomy of provenance in scientific workflow management systems". In: *2009 Congress on Services-I*. IEEE, pp. 259–266.

Das, Arindam and Ajanta De Sarkar (2012). "On fault tolerance of resources in computational grids". In: *International Journal of Grid Computing & Applications* 3.3, p. 1.

Deelman, Ewa, Dennis Gannon, Matthew Shields, and Ian Taylor (2009). "Workflows and e-Science: An overview of workflow system features and capabilities". In: *Future Generation Computer Systems* 25.5, pp. 528–540.

Deelman, Ewa and Yolanda Gil (2006). "Managing Large-Scale Scientific Workflows in Distributed Environments: Experiences and Challenges." In: *e-Science*, p. 144.

Deelman, Ewa, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G Bruce Berriman, John Good, et al. (2005). "Pegasus: A framework for mapping complex scientific workflows onto distributed systems". In: *Scientific Programming* 13.3, pp. 219–237.

Deelman, Ewa, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, et al. (2015). "Pegasus, a workflow management system for science automation". In: *Future Generation Computer Systems* 46, pp. 17–35.

Di, Sheng, Yves Robert, Frédéric Vivien, Derrick Kondo, Cho-Li Wang, and Franck Cappello (2013). "Optimization of cloud task processing with checkpoint-restart mechanism". In: *2013 SC-International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, pp. 1–12.

Dias, Jonas, Eduardo Ogasawara, Daniel de Oliveira, Fabio Porto, Alvaro LGA Coutinho, and Marta Mattoso (2011). "Supporting dynamic parameter sweep in adaptive and user-steered workflow". In: *Proceedings of the 6th workshop on Workflows in support of large-scale science*. ACM, pp. 31–36.

Duan, Rubing, Radu Prodan, and Thomas Fahringer (2006). "Run-time optimisation of grid workflow applications". In: *2006 7th IEEE/ACM International Conference on Grid Computing*. IEEE, pp. 33–40.

Fahringer, Thomas, Radu Prodan, Rubing Duan, Jüurgen Hofer, Farrukh Nadeem, Francesco Nerieri, Stefan Podlipnig, Jun Qin, Mumtaz Siddiqui, Hong-Linh Truong, et al. (2007). "Askalon: A development and grid computing environment for scientific workflows". In: *Workflows for e-Science*. Springer, pp. 450–471.

Fahringer, Thomas, Jun Qin, and Stefan Hainzer (2005). "Specification of grid workflow applications with AGWL: an Abstract Grid Workflow Language". In: *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.* Vol. 2. IEEE, pp. 676–685.

Farkas, Zoltan and Peter Kacsuk (2011). "P-GRADE portal: a generic workflow system to support user communities". In: *Future Generation Computer Systems* 27.5, pp. 454–465.

Garg, Ritu and A Kumar Singh (2011). "Fault tolerance in grid computing: state of the art and open issues". In: *International Journal of Computer Science & Engineering Survey (IJCSES)* 2.1, pp. 88–97.

Gärtner, Felix C (1999). "Fundamentals of fault-tolerant distributed computing in asynchronous environments". In: *ACM Computing Surveys (CSUR)* 31.1, pp. 1–26.

Gil, Yolanda, Ewa Deelman, Mark Ellisman, Thomas Fahringer, Geoffrey Fox, Dennis Gannon, Carole Goble, Miron Livny, Luc Moreau, and Jim Myers (2007). "Examining the challenges of scientific workflows". In: *Ieee computer* 40.12, pp. 26–34.

Gottdank, Tibor (2014). "Introduction to the ws-pgrade/guse science gateway framework". In: *Science Gateways for Distributed Computing Infrastructures*. Springer, pp. 19–32.

Heinis, Thomas, Cesare Pautasso, and Gustavo Alonso (2005). "Design and evaluation of an autonomic workflow engine". In: *Second International Conference on Autonomic Computing (ICAC'05)*. IEEE, pp. 27–38.

Heinl, Petra, Stefan Horn, Stefan Jablonski, Jens Neeb, Katrin Stein, and Michael Teschke (1999). "A comprehensive approach to flexibility in workflow management systems". In: *ACM SIGSOFT Software Engineering Notes*. Vol. 24. 2. ACM, pp. 79–88.

Hollingsworth, Jeffrey K (1998). "Critical path profiling of message passing and shared-memory programs". In: *IEEE Transactions on Parallel and Distributed Systems* 9.10, pp. 1029–1040.

Hwang, Soonwook and Carl Kesselman (2003). "Grid workflow: a flexible failure handling framework for the grid". In: *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on*. IEEE, pp. 126–137.

Jhawar, Ravi, Vincenzo Piuri, and Marco Santambrogio (2013). "Fault tolerance management in cloud computing: A system-level perspective". In: *IEEE Systems Journal* 7.2, pp. 288–297.

Jordan, Diane, John Evdemon, Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, et al. (2007). "Web services business process execution language version 2.0". In: *OASIS standard* 11.120, p. 5.

Kacsuk, Peter (2011). "P-GRADE portal family for grid infrastructures". In: *Concurrency and Computation: Practice and Experience* 23.3, pp. 235–245.

Kacsuk, Peter, Zoltan Farkas, Miklos Kozlovszky, Gabor Hermann, Akos Balasko, Krisztian Karoczkai, and Istvan Marton (2012). "WS-PGRADE/gUSE generic DCI gateway framework for a large variety of user communities". In: *Journal of Grid Computing* 10.4, pp. 601–630.

Krinke, Jens (2001). "Identifying similar code with program dependence graphs". In: *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*. IEEE, pp. 301–309.

Kwak, Seong Woo and Jung-Min Yang (2012). "Optimal checkpoint placement on real-time tasks with harmonic periods". In: *Journal of Computer Science and Technology* 27.1, pp. 105–112.

Lee, HwaMin, KwangSik Chung, SungHo Chin, JongHyuk Lee, DaeWon Lee, Seongbin Park, and HeonChang Yu (2005). "A resource management and fault tolerance services in grid computing". In: *Journal of Parallel and Distributed Computing* 65.11, pp. 1305–1317.

Lee, JongHyuk, SungHo Chin, HwaMin Lee, TaeMyoung Yoon, KwangSik Chung, and HeonChang Yu (2007). "Adaptive workflow scheduling strategy in service-based grids". In: *International Conference on Grid and Pervasive Computing*. Springer, pp. 298–309.

Lee, Kevin, Norman W Paton, Rizos Sakellariou, Ewa Deelman, Alvaro AA Fernandes, and Gaurang Mehta (2009). "Adaptive workflow processing and execution in pegasus". In: *Concurrency and Computation: Practice and Experience* 21.16, pp. 1965–1981.

Lee, Kevin, Rizos Sakellariou, Norman W Paton, and Alvaro AA Fernandes (2007). "Workflow adaptation as an autonomic computing problem". In: *Proceedings of the 2nd workshop on Workflows in support of large-scale science*. ACM, pp. 29–34.

Li, Wing-Ning, Zhichun Xiao, and Gordon Beavers (2005). "On computing the number of topological orderings of a directed acyclic graph". In: *Congressus Numerantium* 174, pp. 143–159.

Li, Zhongwen and Hong Chen. “Adaptive Checkpointing Schemes for Fault Tolerance in Real-Time Systems with Task Duplication”. In:

Lidya, A, S Therasa, G Sumathi, and S Antony Dalya (2010). “Dynamic adaptation of checkpoints and rescheduling in grid computing”. In: *International Journal of Computer Applications (0975–8887)* 2.3.

Ludäscher, Bertram, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A Lee, Jing Tao, and Yang Zhao (2006). “Scientific workflow management and the Kepler system”. In: *Concurrency and Computation: Practice and Experience* 18.10, pp. 1039–1065.

Ludäscher, Bertram, Ilkay Altintas, Shawn Bowers, Julian Cummings, Terence Critchlow, Ewa Deelman, David D Roure, Juliana Freire, Carole Goble, Matthew Jones, et al. (2009). “Scientific process automation and workflow management”. In: *Scientific Data Management: Challenges, Existing Technology, and Deployment, Computational Science Series* 230, pp. 476–508.

Majithia, Shalil, Matthew Shields, Ian Taylor, and Ian Wang (2004). “Triana: A graphical web service composition and execution toolkit”. In: *Web Services, 2004. Proceedings. IEEE International Conference on*. IEEE, pp. 514–521.

Mattoso, Marta, Kary Ocaña, Felipe Horta, Jonas Dias, Eduardo Ogasawara, Vitor Silva, Daniel de Oliveira, Flavio Costa, and Igor Araújo (2013). “User-steering of HPC workflows: state-of-the-art and future directions”. In: *Proceedings of the 2nd ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*. ACM, p. 4.

Meroufel, Bakhta and Ghalem Belalem (2014). “Adaptive time-based coordinated checkpointing for cloud computing workfl ows”. In: *Scalable Computing: Practice and Experience* 15.2, pp. 153–168.

Meroufel, Bakhta and B Ghalem (2014). *Policy Driven Initiator in Coordination Checkpointing Strategies*.

Missier, Paolo, Stian Soiland-Reyes, Stuart Owen, Wei Tan, Alexandra Nenadic, Ian Dunlop, Alan Williams, Tom Oinn, and Carole Goble (2010). “Taverna, reloaded”. In: *International conference on scientific and statistical database management*. Springer, pp. 471–481.

Montagnat, Johan, Benjamin Isnard, Tristan Glatard, Ketan Maheshwari, and Mireille Blay Fornarino (2009). “A data-driven workflow language for grids based on array programming principles”. In: *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*. ACM, p. 7.

Moreau, Luc, Juliana Freire, Joe Futrelle, Robert E McGrath, Jim Myers, and Patrick Paulson (2008). "The open provenance model: An overview". In: *International Provenance and Annotation Workshop*. Springer, pp. 323–326.

Moreau, Luc and Paolo Missier (2013). "Prov-dm: The prov data model". In:

Moreau, Luc, Beth Plale, Simon Miles, Carole Goble, Paolo Missier, Roger Barga, Yogesh Simmhan, Joe Futrelle, Robert E McGrath, Jim Myers, et al. (2008). *The open provenance model (v1. 01)*.

Mouallem, Pierre A. (2011). "A Fault Tolerance Framework for Kepler-based Distributed Scientific Workflows". AAI3479572. PhD thesis. ISBN: 978-1-124-92319-2.

Nakagawa, Sayori, Satoshi Fukumoto, and Naohiro Ishii (2003). "Optimal checkpointing intervals of three error detection schemes by a double modular redundancy". In: *Mathematical and computer modelling* 38.11, pp. 1357–1363.

Oinn, Tom, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R Pocock, Anil Wipat, et al. (2004). "Taverna: a tool for the composition and enactment of bioinformatics workflows". In: *Bioinformatics* 20.17, pp. 3045–3054.

Oliner, Adam J, Ramendra K Sahoo, José E Moreira, and Manish Gupta (2005). "Performance implications of periodic checkpointing on large-scale cluster systems". In: *19th IEEE International Parallel and Distributed Processing Symposium*. IEEE, 8–pp.

Oliveira, Daniel de, Flavio Costa, Vítor Silva, Kary Ocaña, and Marta Mattoso (2014). "Debugging Scientific Workflows with Provenance: Achievements and Lessons Learned". In: *29th SBBD–SBBD Proceedings, Curitiba, PR, Brazil*.

Palaniswamy, Avinash C and Philip A Wilsey (1993). "An analytical comparison of periodic checkpointing and incremental state saving". In: *ACM SIGSIM Simulation Digest*. Vol. 23. 1. ACM, pp. 127–134.

Pesic, Maja (2008). "Constraint-based workflow management systems: shifting control to users". In:

Pietri, Ilia, Gideon Juve, Ewa Deelman, and Rizos Sakellariou (2014). "A performance model to estimate execution time of scientific workflows on the cloud". In: *Workflows in Support of Large-Scale Science (WORKS), 2014 9th Workshop on*. IEEE, pp. 11–19.

Pintas, Julliano Trindade, Daniel de Oliveira, Kary ACS Ocaña, Eduardo Ogasawara, and Marta Mattoso (2013). "SciLightning: a cloud provenance-based event notification for parallel workflows". In: *International Conference on Service-Oriented Computing*. Springer, pp. 352–365.

Plankensteiner, Kassian (2013). "Scientific Workflow Management on Distributed Computing Infrastructures". PhD thesis. Innsbruck, Austria: Universit´at Innsbruck.

Plankensteiner, Kassian, Johan Montagnat, and Radu Prodan (2011). "IWIR: a language enabling portability across grid workflow systems". In: *Proceedings of the 6th workshop on Workflows in support of large-scale science*. ACM, pp. 97–106.

Plankensteiner, Kassian, Radu Prodan, and Thomas Fahringer (2009). "A new fault tolerance heuristic for scientific workflows in highly distributed environments based on resubmission impact". In: *e-Science, 2009. e-Science'09. Fifth IEEE International Conference on*. IEEE, pp. 313–320.

Plankensteiner, Kassian, Radu Prodan, Thomas Fahringer, Attila Kertesz, and Peter K Kacsuk (2007). "Fault-tolerant behavior in state-of-the-art grid workflow management systems". In:

Plankensteiner, Kassian, Radu Prodan, Matthias Janetschek, Thomas Fahringer, Johan Montagnat, David Rogers, Ian Harvey, Ian Taylor, Ákos Balaskó, and Péter Kacsuk (2013). "Fine-grain interoperability of scientific workflows in distributed computing infrastructures". In: *Journal of grid computing* 11.3, pp. 429–455.

Poola, Deepak, Saurabh Kumar Garg, Rajkumar Buyya, Yun Yang, and Kotagiri Ramamohanarao (2014). "Robust scheduling of scientific workflows with deadline and budget constraints in clouds". In: *2014 IEEE 28th International Conference on Advanced Information Networking and Applications*. IEEE, pp. 858–865.

Sakellariou, Rizos and Henan Zhao (2004). "A low-cost rescheduling policy for efficient mapping of workflows on grid systems". In: *Scientific Programming* 12.4, pp. 253–262.

Samak, Taghrid, Dan Gunter, Monte Goode, Ewa Deelman, Gideon Juve, Fabio Silva, and Karan Vahi (2012). "Failure analysis of distributed scientific workflows executing in the cloud". In: *2012 8th international conference on network and service management (cnsm) and 2012 workshop on systems virtualiztion management (svm)*. IEEE, pp. 46–54.

Schroeder, Bianca and Garth Gibson (2010). "A large-scale study of failures in high-performance computing systems". In: *IEEE Transactions on Dependable and Secure Computing* 7.4, pp. 337–350.

Schroeder, Bianca and Garth A Gibson (2007). "Understanding failures in petascale computers". In: *Journal of Physics: Conference Series*. Vol. 78. 1. IOP Publishing, p. 012022.

Shi, Zhiao, Emmanuel Jeannot, and Jack J Dongarra (2006). "Robust task scheduling in non-deterministic heterogeneous computing systems". In: *2006 IEEE International Conference on Cluster Computing*. IEEE, pp. 1–10.

Sindrilaru, Elvin, Alexandru Costan, and Valentin Cristea (2010). "Fault tolerance and recovery in grid workflow management systems". In: *Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on*. IEEE, pp. 475–480.

Starlinger, Johannes, Bryan Brancotte, Sarah Cohen-Boulakia, and Ulf Leser (2014). "Similarity search for scientific workflows". In: *Proceedings of the VLDB Endowment* 7.12, pp. 1143–1154.

Starlinger, Johannes, Sarah Cohen-Boulakia, Sanjeev Khanna, Susan B Davidson, and Ulf Leser (2014). "Layer decomposition: An effective structure-based approach for scientific workflow similarity". In: *e-Science (e-Science), 2014 IEEE 10th International Conference on*. Vol. 1. IEEE, pp. 169–176.

Taylor, Ian, Matthew Shields, Ian Wang, and Omer Rana (2003). "Triana applications within grid computing and peer to peer environments". In: *Journal of Grid Computing* 1.2, pp. 199–217.

Team, SHIWA et al. (2011). *SHIWA: SHaring Interoperable Workflows for Large-Scale Scientific Simulation on Available DCIs*.

Terstyanszky, Gabor, Tamas Kukla, Tamas Kiss, Peter Kacsuk, Ákos Balaskó, and Zoltan Farkas (2014). "Enabling scientific workflow sharing through coarse-grained interoperability". In: *Future Generation Computer Systems* 37, pp. 46–59.

Topcuoglu, H., S. Hariri, and M. Wu (2002). "Performance-effective and low-complexity task scheduling for heterogeneous comput- ing". In: *IEEE Transactions on Parallel and Distributed Systems* 13.3, pp. 260–274.

Turi, Daniele, Paolo Missier, Carole Goble, David De Roure, and Tom Oinn (2007). "Taverna workflows: Syntax and semantics". In: *e-Science and Grid Computing, IEEE International Conference on*. IEEE, pp. 441–448.

Vahi, Karan, Ian Harvey, Taghrid Samak, Daniel Gunter, Kieran Evans, Dave Rogers, Ian Taylor, Monte Goode, Fabio Silva, Eddie Al-Shkarchi, et al. (2012). "A general approach to real-time workflow monitoring". In: *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:* IEEE, pp. 108–118.

Vahi, Karan, Mats Rynge, Gideon Juve, Rajiv Mayani, and Ewa Deelman (2013). "Rethinking data management for big data scientific workflows". In: *Big Data, 2013 IEEE International Conference on*. IEEE, pp. 27–35.

Vockler, Jens S, Gaurang Mehta, Yong Zhao, Ewa Deelman, and Mike Wilde (2007). "Kickstarting remote applications". In: *International Workshop on Grid Computing Environments*.

Wolstencroft, Katherine, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher,

et al. (2013). “The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud”. In: *Nucleic acids research*, gkt328.

Wrzesińska, Gosia, Rob V Van Nieuwpoort, Jason Maassen, Thilo Kielmann, and Henri E Bal (2006). “Fault-tolerant scheduling of fine-grained tasks in grid environments”. In: *International Journal of High Performance Computing Applications* 20.1, pp. 103–114.

Xiang, Xiaorong and Gregory Madey (2007). “Improving the Reuse of ScientificWorkflows and Their By-products”. In: *IEEE International Conference on Web Services (ICWS 2007)*. IEEE, pp. 792–799.

Young, John W (1974). “A first order approximation to the optimum checkpoint interval”. In: *Communications of the ACM* 17.9, pp. 530–531.

Yu, Jia and Rajkumar Buyya (2005). “A taxonomy of scientific workflow systems for grid computing”. In: *ACM Sigmod Record* 34.3, pp. 44–49.

Zhao, L., Y. Ren, Y. Xiang, and K. Sakurai (2010). “Fault-tolerant scheduling with dynamic number of replicas in heterogeneous systems”. In: *12th IEEE International Conference on High Performance Computing and Communications (HPCC)*, pp. 434–441.

Own Publications Pertaining to Theses

K-1 Bánáti, Anna, Eszter Kail, Péter Kacsuk, and Miklós Kozlovszky (2015). “Usability of Scientific Workflow in Dynamically Changing Environment”. In: *Technological Innovation for Cloud-Based Engineering Systems: 6th IFIP WG 5.5/SOCOLNET Doctoral Conference on Computing, Electrical and Industrial Systems, DoCEIS 2015, Costa de Caparica, Portugal, April 13-15, 2015, Proceedings*. Ed. by M. Luis Camarinha-Matos, A. Thais Baldissera, Giovanni Di Orio, and Francisco Marques. Springer International Publishing, pp. 129–136. ISBN: 978-3-319-16766-4. DOI: 10.1007/978-3-319-16766-4_14. URL: http://dx.doi.org/10.1007/978-3-319-16766-4_14.

K-2 Kail, E, A Bánáti, P Kacsuk, and M Kozlovszky. “Provenance based adaptive and dynamic workflows”. In: *15th IEEE International Symposium on Computational Intelligence and Informatics*, pp. 215–219.

K-3 Kail, Eszter, Anna Bánáti, Krisztián Karóczkai, Péter Kacsuk, and Miklós Kozlovszky (2014). “Dynamic workflow support in gUSE”. In: *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2014 37th International Convention on*. IEEE, pp. 354–359.

K-4 Kail, Eszter, Péter Kacsuk, and Miklós Kozlovszky (2015a). “A novel approach to user-steering in scientific workflows”. In: *Applied Computational Intelligence and Informatics (SACI), 2015 IEEE 10th Jubilee International Symposium on*. IEEE, pp. 233–236.

K-5 — (2015b). “Achieving dynamic workflow management system by applying provenance based checkpointing method”. In: *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015 38th International Convention on*. IEEE, pp. 250–253.

K-6 — (2015c). “New aspect of investigating fault sensitivity of scientific workflows”. In: *Intelligent Engineering Systems (INES), 2015 IEEE 19th International Conference on*. IEEE, pp. 185–188.

K-7 — (2016a). “A Novel Adaptive Checkpointing Method Based on Information Obtained from Workflow Structure”. In: *Transaction on Automating Control and Computer Science* 3.to be appear.

K-8 — (2016b). “Specification of user and provenance based adaptive control points at workflow composition level”. In: *International Symposium on Intelligent Sytems and Informatics (SISY), 2015 IEEE 14th*. IEEE.

K-9 Kail, Eszter, Krisztián Karóczkai, Péter Kacsuk, and Miklós Kozlovszky (2016). “Provenance Based Checkpointing Method for Dynamic Health Care Smart System”. In: *Scalable Computing: Practice and Experience* 17.2, pp. 143–153.

Own Publications Not Pertaining to Theses

Kx-1 Kail, E, S Khoor, K Fugedi, I Kovacs, B Khoor, B Kail, P Kecskeméthy, N Balogh, E Domijan, and M Domijan (2005). “Expert system for phonocardiographic monitoring of heart failure patients based onwavelet analysis”. In: *Computers in Cardiology, 2005*. IEEE, pp. 833–836.

Kx-2 Kail, E, S Khoor, B Kail, K Fugedi, and F Balazs (2004). “Internet digital phonocardiography in clinical settings and in population screening”. In: *Computers in Cardiology, 2004*. IEEE, pp. 501–504.

Kx-3 Kail, E, S Khoor, and J Nieberl (2005). “Ambulatory wireless Internet electrocardiography: new concepts & maths”. In: *2nd International Conference on Broadband Networks, 2005*. IEEE, pp. 1001–1006.

Kx-4 Kail, Eszter, Gábor Németh, and Zoltán Richárd Turányi (2001a). “The effect of the transmission range on the capacity of ideal ad hoc networks”. In: *rN* 2, p. 3.

Kx-5 Kail, Eszter, Gábor Németh, and Zoltán Richárd Turányi (2001b). "Throughput of ideally routed wireless ad hoc networks". In: *Proceedings of the 2nd ACM international symposium on Mobile ad hoc networking & computing*. ACM, pp. 271–274.

Kx-6 Khoór, S, J Nieberl, K Fugedi, and E Kail (2003). "Internet-based, GPRS, long-term ECG monitoring and non-linear heart-rate analysis for cardiovascular telemedicine management". In: *Computers in Cardiology, 2003*. IEEE, pp. 209–212.

Kx-7 Khoor, S, K Nieberl, K Fugedi, and E Kail (2001). "Telemedicine ECG-telemetry with Bluetooth technology". In: *Computers in Cardiology 2001*. IEEE, pp. 585–588.

Kx-8 Rónai, Miklós Aurél and Eszter Kail (2003). "A simple neighbour discovery procedure for Bluetooth ad hoc networks". In: *Global Telecommunications Conference, 2003. GLOBECOM'03. IEEE*. Vol. 2. IEEE, pp. 1028–1032.

Kx-9 Turányi, Zoltán R, Csanád Szabó, Eszter Kail, and András G Valkó (2000). "Global internet roaming with ROAMIP". In: *ACM SIGMOBILE Mobile Computing and Communications Review* 4.3, pp. 58–68.