Institut für Parallele und Verteilte Systeme
Abteilung Verteilte Systeme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3418

# Methods to coordinate the execution of workflow replicas in a distributed environment

Thomas Bach

| | |
|---|---|
| **Course of Study:** | Software Engineering |
| **Examiner:** | Prof. Dr. Dr. h. c. Kurt Rothermel |
| **Supervisor:** | MSc. Muhammad Adnan Tariq |
| **Commenced:** | October 4, 2012 |
| **Completed:** | April 5, 2013 |
| **CR-Classification:** | C.2.4, D.4.5 |

## Abstract

In many distributed systems robustness is a major concern since network nodes might fail spontaneously. Such failures can be a major problem when a workflow needs to be run on a set of unreliable and distributed nodes. Replication is a widely used architecture paradigm to increase system reliability. This thesis addresses robustness of workflow execution in distributed systems using replication.

A workflow is a plan that describes how a number of tasks needs to be executed and is defined using a workflow definition language. A task is the description of a single operation contained within such a workflow. The first part of this thesis gives an overview over available workflow definition languages focusing mainly on declarative ones. Task types present in this languages are identified and their impact on replication is evaluated.

A basic problem of workflow replication is that not all tasks within a workflow can be executed arbitrarily often. To solve this issue the execution of such tasks must be coordinated. The goal of this thesis is to propose and evaluate methods to coordinate the execution of such workflow replicas in a distributed environment.

The proposed replica coordination algorithms are implemented as a peer to peer protocol and simulated using the peer to peer simulator PeerSim. A synthetic workflow generator is used to provide a large number of workflows for evaluation to test the performance, scalability and robustness under different conditions. The evaluation is concluded with the replication of a real workflow to judge the significance of the synthetic tests to the real world.

# Contents

# List of Figures

# List of Algorithms

Chapter 1

# Introduction

## 1.1 Motivation

In the last decade the field of mobile computing has been steadily becoming more important and complex. Pervasive computing applications have made good progress evolving from laboratory scale prototypes to daily used market fit products. The diversity of available computing platforms increases rapidly and the task of running applications and services in such an heterogeneous, highly dynamic and unreliable environment remains challenging.

Workflows can provide a flexible and high level way to model the functionality and temporal order of complex tasks in many areas of information technology. Evolving from petri nets imperative workflow definition languages have become the tool of choice to model complex processes and applications. Acting on a high abstraction layer and allowing the user to focus on modeling functionality enabled their huge success.

Being petri net-based imperative languages need to precisely define the order and relation of all tasks. Especially the required definition of the task order might not always be needed and can make imperative workflows rigid and inflexible. Execution flexibility has to be explicitly modeled into a workflow which can easily lead to over specified and complex arrangements. Consider an example where three tasks can be executed in an arbitrary order. When using an imperative language every possible order would have to be explicitly defined.

In declarative languages however only necessary constraints have to be defined. If for example no order is defined the tasks can be executed any time. By not forcing the user to define possibly irrelevant details declarative languages can be much more flexible. Before however a workflow defined in a declarative language can be executed a concrete execution order has to be chosen.

While the user only needs to define execution order and other details when required by the process the execution system has more freedom to decide when or how the tasks are executed. This additional flexibility makes declarative languages a promising programing paradigm for distributed systems and pervasive computing applications. It allows workflow languages to bridge the gap between high level application definition and possibly heterogenous infrastructure.

The tasks in a workflow can have a limited number of executions. For example the booking of one airline ticket can only be done once. When replicating a workflow such tasks can thus only be executed on one replica and need to be coordinated. While one of the replicas executes such a task the others would be idle. Using the flexibility of declarative languages multiple execution orders can be generated and the number of idle replicas can be reduced. This could also speedup the execution of the workflow since multiple tasks can be executed in parallel.

## 1.2 Contribution to the field

As discussed in the related work section some efforts have been made to increase robustness of workflows using replication. While most of the work considers internet scale web service replication [SPJ11] [MSB08] et al. There is also some work regarding workflow replication in ad-hoc networks [DJ07].

This thesis considers the benefits of using declarative languages to define workflows intended for replication and how this translates into coordination methods.

In the first part of this thesis available declarative languages are researched. In need for a language capable of defining complex coherences and temporal constraints the Declare language is chosen. Since the original Declare language doesn't support temporal constraints an extension proposed by [LUW10] is used. Based on that language different task types are identified and their impact on replication is analyzed.

Regarding the identified task types the coordination algorithms are developed. While the execution of tasks that cannot be executed arbitrarily often needs to be coordinated globally the execution of others can be decided locally on every replica. This global coordination process must somehow be organized to ensure the correct execution of all tasks.

The previously described flexibility of declarative languages is used to further increase the efficiency of the workflow execution. By replicating a workflow in different orders it is possible to run several tasks in parallel and decrease the execution time of the workflow. The evaluation shows that this also decreases the message overhead and idle time of the replicas.

In the last part of the thesis the proposed algorithms are evaluated on a large number of synthetically generated workflows. The algorithms are implemented as a peer to peer protocol and simulated using the peer to peer simulator PeerSim. A workflow generator was added to provide a large number of test workflows.

The simulations showed that replicating a workflow in different orders can increase robustness while execution and idle time decrease. As declarative languages can provide this kind of flexibility they seem to be a good choice to model workflows which need to run in distributed environments.

## 1.3 Organization of the thesis

The rest of this thesis is organized as follows.

- Chapter two will present the work related to this thesis.
- Chapter three introduces the used terminology, the characteristics of the addressed distributed environment, and the system model.
- Chapter four will briefly introduce some workflow languages. The presented languages are evaluated regarding their fitness for use in this thesis. The Declare language with its extension regarding temporal constrains is presented and the graphical notation for temporal constraints is introduced. Focusing on the Declare language task types are identified.
- Chapter five develops workflow replication and coordination methods. An introductive example explains the issues that have to be considered. Replication is discussed regarding data handling, messaging, coordination, identified task types, execution order, failure detection, and error handling. Examples illustrate the proposed strategies and conclude the chapter.
- Chapter six evaluates and discusses the proposed coordination algorithms.
- A conclusion chapter wraps up the thesis and summarizes the results of this work.

Chapter 2

# Related Work

This section presents work related to the topic of this thesis. Different approaches to reliability are being presented. Most of the approaches try to make distributed workflow execution robust by switching to backups or for critical tasks (Sections 2.1, 2.2), or even running critical tasks multiple times (Section 2.3).

If a workflow is using an external service it might be beneficial to not only rely on one service but maintain several interchangeable web services. Methods to measure their level of independence are proposed in the work presented in Section 2.4.

A completely different approach is described in Sections 2.5 and 2.6 where transactions are used to increase robustness of a distributed workflow.

The work presented in Section 2.9 proposes methods to deal with failure on a workflow level.

The detection of failed components is a major issue in fully distributed systems and addressed by the work presented in Section 2.10.

## 2.1 Dynamic Web Service Replication in Ad-Hoc Networks

In [DJ07] Dustdar and Juszczyk describe a system for flexible replication and synchronization of web services in ad-hoc networks. In contrast to the work presented in this thesis their system deals with the dynamic placement, provisioning, and coordination of multiple web services. One web service is only run by one replica at a time (possibly the strongest) and the states of all replicas replicating this service are synchronized.

Basing on wireless ad-hoc networks the proposed system has to deal with a series of difficulties. Due to their mobility, unreliability, and possibly poor wireless connection the nodes can spontaneously disappear or relocate in the network. This makes the network topology highly dynamic and the behavior of single nodes unpredictable. To deal with this difficult environment the following strategies are used:

- **Dynamic Replica Placement:** Replica placement refers to assignment of services to nodes. When the network state changes this placement is reevaluated. It is checked if the deployed services run on a sufficient number of nodes. In addition the load between the different nodes is balanced. If a node is under constant heavy load services are moved away. If the failure of a node seems to be imminent (i.e. a depleted battery) the running services are reallocated.
- **Completely decentralized:** In ad-hoc networks the chance of failure is equally high thus it is important to avoid a single point of failure.
- **Monitor availability:** Due to the continuos change and high dynamic of the network infrastructure it is critical to continuously monitor the state of the network and act appropriately.
- **little traffic:** The devices participating in an ad-hoc network are usually battery relay ant. Since wireless communication is quite expensive regarding battery power the system tries to minimize network traffic.
- **network partitioning:** An interface is provided to specify algorithms for resynchronization after rejoining of partitioned networks. This might not always be possible.

## 2.2 Using Communities to Sustain High-Availability of Web Services

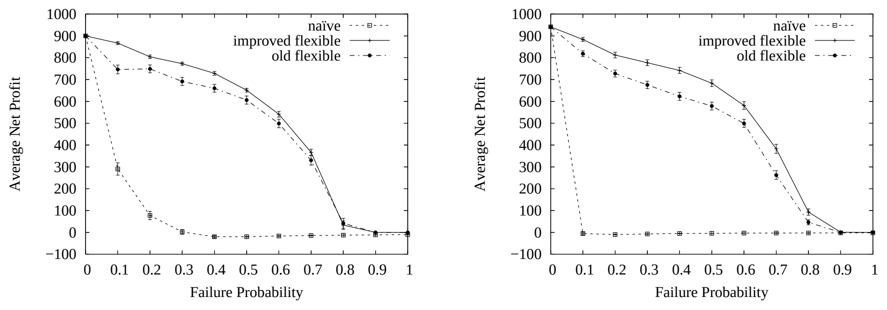
In [MSB08] Maamar, Sheng and Benslimane propose how the availability of web services can be increased by grouping them together in communities. Such a community combines several web services providing the same functionality. This web services can however have different non-functional properties and are combined independently of their origins.

A designer specifies the function of a new community i.e. "book a flight" and deploys a master web service. This master web service will lead the group. It is his responsibility to manage the community, attraction of new slave web services with rewards, and retaining existing web services in the community. He also identifies web services that will participate in composite web services.

When a service is needed by a client the master web service acts as an abstraction layer that provides the requested service by utilizing one of the slave replicas. If one of the slave replicas fails another one can be used as a backup. If the functionality provided by the master is not requested anymore the community is dismantled.

## 2.3 Flexible Provisioning of Service Workflows

Stein, Payne, and Jennings are researching methods to flexible provision services [SPJ11]. In service oriented computing different service provider offer the same kind of service to their customers at different levels of quality and price.



(a) Small workflows (10 tasks). (b) Large workflows (50 tasks).

**Figure 2.1:** Advantage of flexible Service Provisioning. Source [SPJ07]

These services are dynamically procured by customers which use them in their complex business processes. Since these workflows consist of several interdependent tasks the failure of a single dependency can have a highly detrimental knock off effect on other related services.

Unfortunately the behavior of such services is beyond the control of the consumer. They may fail or take a long time to execute. This makes them a critical issue. When the consumer is bound to time constrains possibly involving penalties and fees this becomes even more important.

Some approaches deal with this issues by applying quality-of-service measures or use utility theory to provision the single best qualified service provider [SPJ07]. In [SPJ07] Stein, Payne, and Jennings state that provisioning only one provider would lead to brittle workflows that are highly vulnerable to single failures. To solve this they propose to provision several service providers for particularly failure prone tasks.

Building on previous work [SJP06] they propose an improved strategy for flexible provisioning of multiple services [SPJ07] that shows an 8.4 % improvement compared to their old approach. Compared to the naive approach this is an improvement of 385 %. See Figure 2.1.

## 2.4 Deploying Reliable Web Services using Independent Replicas

A web service typically consists of one or more different components. If one of the components fails the web service would fail, too. To increase robustness several of those web services providing the same "service" could be used as backup. Obviously it would be unless to chose two different web services that rely on the same components because if one of those components is not available both web services would fail.

For example if a consumer wants to book a flight he can use several different websites to do that. When those websites however rely on the same booking system their availability is bound to the availability of the same booking service. When selecting backup websites to chose from it would be nice to know witch of them are totally independent to maximize fault tolerance.

This thesis proposes choosing different orders of a workflow for replication which have to be generated prior to execution from a workflow model. Depending on the workflow model in this process different workflows orders containing even different components are possible. If some of this components rely on external web services it would maximize fault tolerance if it would be possible to chose those workflow orders that have the least components in common.

In [TLHL09] Tang, Li, Hua, and Liu propose how to judge the independence of such web services, especially when the service provider is not willing to share implementation details about the web service.

## 2.5 Pervasive Workflow Execution Utilizing Transactions

In [MMTG08] Montagut, Molva and Golega propose an adaptive transactional protocol for a pervasive workflow model developed in previous work to support pervasive workflow execution.

The tasks of a workflow are distributed on different pervasive nodes. Critical tasks are only issued to reliable nodes. Another core idea is to introduce critical zones within workflows. In this zones transactions are used to either ensure the correct execution of a workflow or be able to roll back. Prior to the execution transaction rules are negotiated. It is defined what is to be done when a task of the critical zone fails and the already executed tasks need to be rolled back. To minimize such rollbacks the tasks contained in a critical zone are dispatched to nodes that have been proven to be reliable.

Another paper advocating the use of transactional models to increase robustness of workflows is presented by Alonso et al. [AAEA+96]. They however state that traditional transactions are to rigid for flexible workflows. They suggest the usage of flexible transactions utilizing subtransactions. In case of a failure a different sub transaction could be submitted to finish the transaction successfully.

## 2.6 Speres of Isolation for Transaction based workflows

Guabtni, Charoy, and Godart study the needs of Workflow systems for isolation properties [GCG05]. They propose the creation of isolation spheres in workflows for customized isolation constraints in transactional workflows. The goal is to allow the designer to decide the degree of isolation for a group of tasks. Such a group forms a so called atomic sphere to apply transactional constraints to.

Isolation spheres can be created by identifying several tasks using the same data. Within a sphere all tasks operate on the same data and can be sure that it is not changed from outside of the sphere. If the execution of the sphere fails no critical data (outside of the sphere) has been changed.

The use of spheres could be useful for executing several related tasks in a distributed workflow. Suppose a workflow is used to organize the booking of a flight, a hotel room and a conference ticket. All the booking procedures are executed on a different node in parallel. Before the execution is started a isolation sphere is created in cooperation with the service provider. When one of the booking tasks fails (for example there is no flight available) the other tasks can be aborted. When all tasks are executed successfully the whole sphere commits and the flight, the conference ticket and the hotel room are definitively booked.

## 2.7 Broadcast Protocols for Distributed Systems

In a distributed system the processes running on the different nodes have to exchange status information and execution results. While some status information or messages need only to be transmitted to a few number of receivers global status updates need to be transmitted to all nodes. Especially when using a shared media broadcasting one single message for all to read could reduce message overhead. Since communication is expensive in terms of power consumption this can be particularly important when the nodes are running on battery power. How can be ensured that every node successfully receives one broadcast message without implementing a complicated and messaging intense acknowledging protocol?

Melliar-Smith, Moser, and Agrawala propose a broadcast based [MSMA90] protocol (which they intent to use for distributed decisions) that allows reliant transmitting of messages to several recipients. They point out that broadcast medias such as wireless are often only used for point to point messaging. They propose a protocol that makes use of the broadcast based nature of those medias to reduce message overhead.

The proposed replica coordination strategy proposed in this thesis also needs to transmit some execution information to all nodes in the system. This means sending an status update message to every node and making sure the message has been delivered successfully. Relying on a broadcast based messaging approach when sending execution information updates to all replicas could greatly reduce the message overhead. To guarantee successful transmission to every node the presented paper proposes to piggy back reception acknowledgement on existing message flow between the nodes. That way messaging overhead could be greatly reduced.

## 2.8 Consensus Services

As explained in Section 5.2 the execution non idempotent tasks has to be coordinated globally. In this thesis a replicated coordinator decides about the execution of such tasks. Once the system has agreed on a central coordinator no further agreements are needed until a new coordinator needs to be elected for whatever reasons. Another approach to decisions in distributed systems could be the use of a distributed consensus service [GS96].

As consensus can be hard to solve in distributed asynchronous systems [GS96] Guerraoui et al. propose the use of a consensus server processes executed on several nodes of the distributed system.

## 2.9 Fault Tolerance in Dataflow-based Scientific Workflow Management

In contrast to business processes that rely on transactions that can be rolled back in case of a failure recovery strategies in scientific workflows are based on re executing the failed work [YMV$^+$10]. In their paper Yildiz, Mouallem, Vouk, Crawl, and Altinats see two possible strategies for designing a fault tolerant workflow.

The designer could specify a fault tolerant workflow by including workflow fragments that will be re executed in case of failures or the workflow engine provides these fragments without the intervention of the designer deriving them from the workflow model.

The goal of their work is to achieve the latter by trying to automate the structuring of recovery fragments within a workflow model. A recovery strategy is proposed that allows generation of such fragments by the workflow management system at design time. This approach deals with failures on a workflow level and proposes strategies for handling different kinds of failures like tolerating a number of failed items, discarding a complete activity, immediate reelection or recovery of dependent instances.

## 2.10 Failure Detection

To detect failing replicas this thesis utilizes a lazy timeout based failure detection strategy. (See Section 5.3.4.) This was inspired by [HCK02]. Hayashibara, Cherif, and Katayama discuss the problem of implementing scalable failure detection systems. Having identified typical issues raised in the context of Grid systems they evaluate existing approaches accordingly. Despite focusing on Grid systems this paper gives a good overview over failure detection strategies in distributed systems in general.

Chapter 3

# Workflow Languages

This chapter gives a short overview over workflow languages and highlights the difference between imperative and declarative languages. Focusing on extended Declare task types present in declarative languages are identified.

A workflow language is a formal language that can be processed by a computer. It can be used to model business processes or complex applications on a high level. Workflow languages usually consist of tasks and relations [WV97].

A task is a functional unit describing an operation. This can be a human assignment like "Take the dog for a walk." or an instruction for a computer like "Write input A to database D.". It can even represent the call of an complex external service like "Book an airline ticket to Dublin". This of course implies that the necessary information to execute this task has been previously acquired by the system. This would require a task like "Ask the user for Contact and payment information."

To make sure the booking information is available before executing the booking task the workflow needs to specify the execution order of both tasks. This introduces task relations. To add steering information to a workflow relations are used. The simplest relations would be ordering relations like before and after.

Additional constraints could specify the way tasks are executed. For example how often a task can be executed or if two tasks need to be executed in parallel.

## 3.1 Imperative languages

Humans tend to describe what they can "see" thus the most common method to model a workflow is to "just state" what has to be done in which order. This type of workflow definition is called "imperative" less common "procedural". In such a language the tasks are arranged in a graph like structure that reminds a little bit of a flowchart. An imperative language can be roughly compared with an ordinary programming language. The instructions of the programming language resemble the tasks of a workflow language. Like in a programming language the tasks of a workflow need to be executed in the order they are specified in the graph.

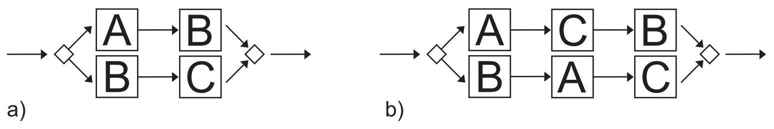


**Figure 3.1:** a) An imperative workflow modeling different execution orders of two tasks. b) The restructured workflow after adding a new task (C) after task A.

If it needs to be possible to execute some tasks in different orders those orders have to be explicitly defined. See Figure 3.3.a. As all alternatives have to be added to the model during build time this could lead to an over specification of the workflow [PWZ+12].

Introducing new tasks to an imperative workflow could cause a restructuring of the whole workflow. This is shown in Figure 3.3. To the workflow on the left side an additional task C needs to be added after task A. This could be a major effort if the workflow would be more complex than this small example. Another good example for this problem is given in [HM11].

In conclusion imperative languages are rather rigid and all flexibility must explicitly be modeled into an imperative workflow. Generally speaking the more tasks and rules a workflow consists of the more function and flexibility is provided.

## 3.2 Declarative languages

Declarative languages are a family of workflow languages that define a workflow using constraints. Constraints are statements (rules) restricting the way a task is executed. For example restricting the number of executions for a tasks or defining an absolute order between two tasks.

Figure 3.2 depicts such a workflow that consist of three tasks, (A, B, and C) and two constraints.

The arrow between task A and B is a constraint that limits the possible numbers of execution orders. It means that before any execution of task B, task A has to be executed. This is a really relaxed example for an ordering constraint since it only constrains the way task B is executed. While task A can still be executed at any time task B requires that task A has at least been executed once before. After that task B can be executed arbitrarily often.

The constraint attached to task C defines its minimal and maximal number of execution. In this case task C has to be executed at least once. Constraints are discussed in detail in Section 3.4.5.

To pick up the example presented in Section 3.1 Figure 3.3.a depicts a workflow with two tasks that can be executed in an arbitrary order modeled in a declarative language. Note that in a declarative language the tasks are not in an explicit order by default like this is the case for imperative languages. Thus modeling the possibility to execute tasks in an arbitrary order means just not constraining the order. Figure 3.3.b depicts how the declarative workflow depicted in 3.3.a has to be changed when a task C is added that has to be executed after task A.

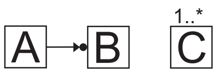


**Figure 3.2:** A declarative process model.

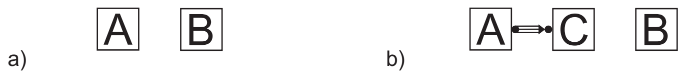


**Figure 3.3:** a) A declarative workflow is implicitly modeling arbitrary execution orders of two tasks. b) The restructured workflow after adding a new task (C) that has to be executed directly after task A.

Basically a declarative language does not specify the tasks in an explicit order. The language just defines constrains on tasks and orders between them when necessary. By not being forced to specify each and every small, possibly unimportant detail declarative workflows do not tend to over specify [PWZ+12]. When adding constraints to a declarative workflow flexibility and functionality decrease because the possibilities to interpret such a flow are further limited.

## 3.3 Declarative vs. Imperative Workflow Modeling Languages

The previous Section introduced declarative and imperative workflow modeling languages. Imperative languages are widely spread and the traditional choice for workflow modeling however declarative languages seem to have certain advantages. There is some work comparing maintainability [FMR+09] and understandability [FLM+09] of both approaches.

Based on that work Pichler et al. have conducted experiments to evaluate which approach is superior in respect to process modeling and understandability [PWZ+12]. They conclude that in general imperative languages seem to be more comprehensible especially regarding sequential information. Circumstantial information is however better understandable using declarative languages.

As previously mentioned the same functionality can be modeled quite different in an imperative and declarative languages. While imperative languages depict workflows in a flow like characteristic and predefine an execution order declarative flows rely on constraints. This quite different characteristic is shown in Figure 3.4. While both flows model the same functionality the declarative model appears to be more light weight.

The problem with this slim kind of depiction however could be that it is not as easy to interpret as the declarative model. The fixed execution order of the imperative workflow provides a guideline for execution and interpretation for a human user. Before however a user can interpret all the different execution possibilities of a declarative workflow he first has to fully analyze it. There also might be some hidden characteristics that can be easily overlooked because they do not need to be explicitly specified. In essence the freedom in execution of a declarative flow is

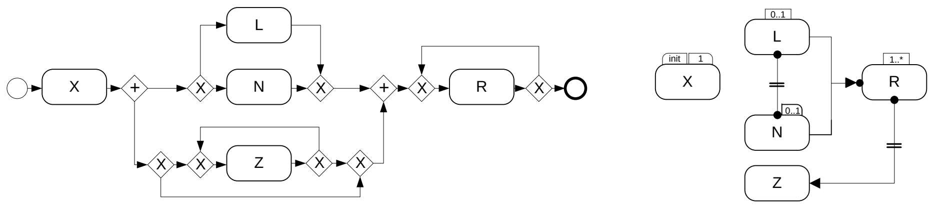


**Figure 3.4:** A workflow modeled in an imperative (BPML) and declarative language (conDec). *Figure copied from [PWZ+12]*

modeled in the absence of constraints. It is literally modeled by nothing. In an imperative flow however this freedom needs to be explicitly modeled as previously discussed.

This conflicting approach is commonly depicted [PA06] [DAPS09] like in Figure 3.5. While both languages can model the core functionality of a workflow imperative languages need to specify additional dynamic and flexibility additionally. The arrows in spreading in different directions illustrate the extra modeling effort that has to be done to cover the complete actual process.

Declarative languages on the other hand leave this problem to the interpretation abilities of the user and just constrain a workflow so that it does not cover something that is not part of the actual process. This however can make modeling difficult since not all ways of interpretation are obvious all the time.

In reality the boundaries between the actual process and the declarative workflow might not be so sharp and clear. Since the modeling process can be difficult the declarative workflow could be modeled more constrained than actually necessary and not cover the actual workflow completly.

The additional flexibility of declarative languages regarding execution order could be beneficial when a declarative workflow needs to be executed in distributed systems. Before a declarative workflow can be executed an execution order of the tasks needs to be generated from the declarative flow since the declarative workflow doesn't necessarily provide a order for all tasks. This makes it possible to generate several different execution orders of a workflow.

In an distributed environment not all nodes might be able to execute a particular task. A flexible workflow execution order could enable these nodes to execute another task. This could be achieved by replicating a workflow in different orders. This thesis will show that this approach has several advantages regarding replication such as reduced execution time and increased robustness.

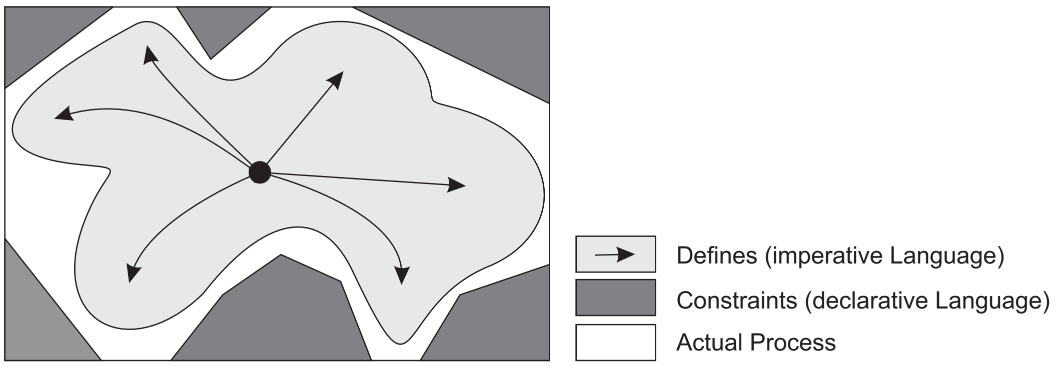

Defines (imperative Language)
Constraints (declarative Language)
Actual Process

**Figure 3.5:** The contrary approach of declarative and imperative languages to model a process.

## 3.4 Declarative Languages

### 3.4.1 ADEPT

ADEPT [DR09] is not a declarative workflow language. It is the approach to design a flexible process management technology for the clinical environment using an imperative workflow. The ADEPT project marks an early approach to provide a flexible business process management environment.

The motivation for the ADAPT project was to provide a tool to support clinical processes. The situation in the clinics was that a lot of legacy software existed that was used by the staff but did not provide any process support. The processes only existed in the mind of the users and notes on paper or in a calendar were the only help for physicians not to forget things.

Modeling the clinical workflow using an imperative workflow language approach similar to petri-nets proved too complex and inflexible. The results have been large and too complex process models. As previously described this can be a common problem when using imperative workflow languages since flexibility needs to be modeled into an imperative workflow at design time.

The approach to solve this was to provide the user with predefined process templates that can be combined by the user in a flexible manner. Despite this is not an example for a declarative language it describes the desperate need for flexible workflow languages and an attempt to bring more flexibility to declarative languages.

### 3.4.2 ROsWeL

The **R**essource **O**riented **W**orkflow **L**anguage represents a declarative language approach to define resource oriented Web Services [BDF+12]. In contrast to existing Web Service technologies that build on a stack of protocols like SOAP, WSDL, etc. ROsWeL is based on the Representational State Transfer (REST) architecture paradigm. In the REST approach the services communicate using bare HTTP protocol utilizing GET, PUT, POST, and DELETE

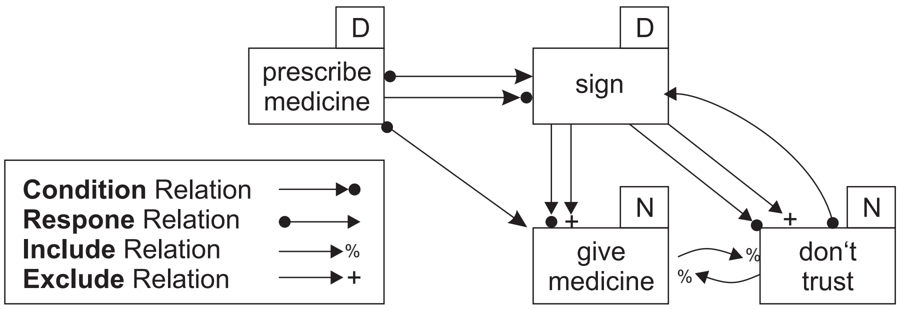


**Figure 3.6:** Distributed Decission Condition Response Graphs example workflow. *Figure redrawn based on [HM11].*

methods. This allows for a complete separation of client and server implementation. The ROsWeL language is another example that shows that declarative languages are also being developed targeting theWeb Services sector.

### 3.4.3 Distributed DCR Graphs

Since LTL (Linear temporal logic) seems difficult to understand for most end users Hildebrandt and Mukkamala propose Distributed **D**ecission **C**ondition **R**esponse Graphs to model declarative event based workflow processes [HM11]. Essentially the language is based on events and four different relations between them (condition, response, include, and exclude).

A distributed DCR Graph is defined [HM11] as a tuple $G = (\mathsf{E},\mathsf{M},\mathsf{Act},\rightarrow\bullet, \bullet\rightarrow, \pm, \mathrm{l})$ where

i) $\mathsf{E}$ is the set of events

ii) $\mathsf{M} \in \mathcal{M}(\mathrm{G}) = \mathcal{P}(\mathsf{E}) \times \mathcal{P}(\mathsf{E}) \times \mathcal{P}(\mathsf{E})$ is the marking and $\mathcal{M}(\mathrm{G})$ is the set of all markings

iii) $\mathsf{Act}$ is the set of all actions

iv) $\rightarrow\bullet \subseteq \mathsf{E} \times \mathsf{E}$ is the condition relation

v) $\bullet\rightarrow \subseteq \mathsf{E} \times \mathsf{E}$ is the response relation

vi) $\pm : \mathsf{E} \times \mathsf{E} \rightharpoonup \{+, \%\}$ defines the dynamic inclusion/exclusion relations by $e \rightarrow +e'$ if $\pm(e, e') = +$ and $e \rightarrow \%\ e'$ if $\pm(e, e') = \%$

vii) $\mathrm{l} : \mathsf{E} \rightarrow \mathsf{Act}$ is a labeling function mapping every event to an actions.

Figure 3.6 shows an example workflow defined using the graphical notation of Distributed DCR Graphs. The graphical notation for the response and condition relation is based on the notation proposed for these by [DAPS09]. The inclusion and exclusion relation are depicted by arrows with a + or respectively a % at the end.

While the condition and response relation define the order of two evens the inclusion and exclusion relations can be used to enable or disable an event dynamically. The workflow is executed successfully when all relations are fulfilled. A disabled event however does not need to be executed to successfully end the execution of a workflow.

The presented example workflow models the prescription and giving of medicine in a hospital scenario. The doctor needs to prescribe medicine and certifies the correctness. This is modeled by the events prescribe medicine and sign. The nurse administers the medicine to the patient. The events are also labeled to define which roll can execute them (D for doctor and N for nurse).

In the given example the sign and give medicine events are defined as responses to the prescribe medicine event and thus need to be executed (unless they are excluded) to successfully end the execution of the workflow. When the doctor has prescribed and signed the medicine the nurse has two options. When the sign event is executed it includes the give medicine and don't trust events to the workflow. One of them can now be executed by the nurse. When one of them is executed the other event is excluded and the workflow and all relations are fulfilled.

Distributed DCR Graphs provide a flexible way to model event based workflows. Like all declarative languages the control flow is defined implicitly providing maximum flexibility to the user. They also provide a clear and easy readable graphical notation. Compared to languages like Declare they lack the ability to precisely define the order of the tasks when needed. If for example one task needs to be done directly after an other task without any third task being executed in between this cannot be defined. It is also not possible to define temporal constraints like parallel execution or tasks starting simultaneously.

This restrictions might be acceptable for human workflows but is not acceptable when the language is intended as a programming paradigm. When defining workflows that are executed by a distributed system it needs to be possible for the designer to precisely specify the absolute and temporal order of the tasks.

### 3.4.4 Declare

A lot of work in the field of declarative workflow languages has been done at the University of Eindhoven, Netherlands by W.M.P. van der Aalst and M. Pesic. In the need for a flexible declarative workflow language they proposed the Declare workflow management system. The Declare engine is designed to execute workflows specified in LTL (Linear Temporal Logic). For error checking the LTL formula is translated to a finite automaton. This automaton can be used to check if a workflow has been executed successfully.

A workflow can be in three states. If the automaton is in an accepting state the workflow execution has finished successfully. If the automaton is not in an accepting state but an accepting state of the automaton can still be reached the workflow is temporarily violated. If the automaton is not in an accepting state and cannot reach it the workflow execution has failed.

Since LTL can be hard to understand for the end user van der Aalst et al. propose the use of LTL templates that have a graphical representation and can be used for easy design and depiction of the worklfows. [PA06] [VDAP06] [PSSA07] [PSA07] [Pes08]

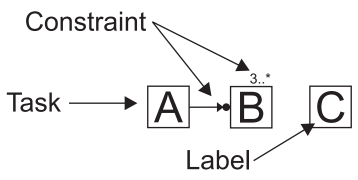
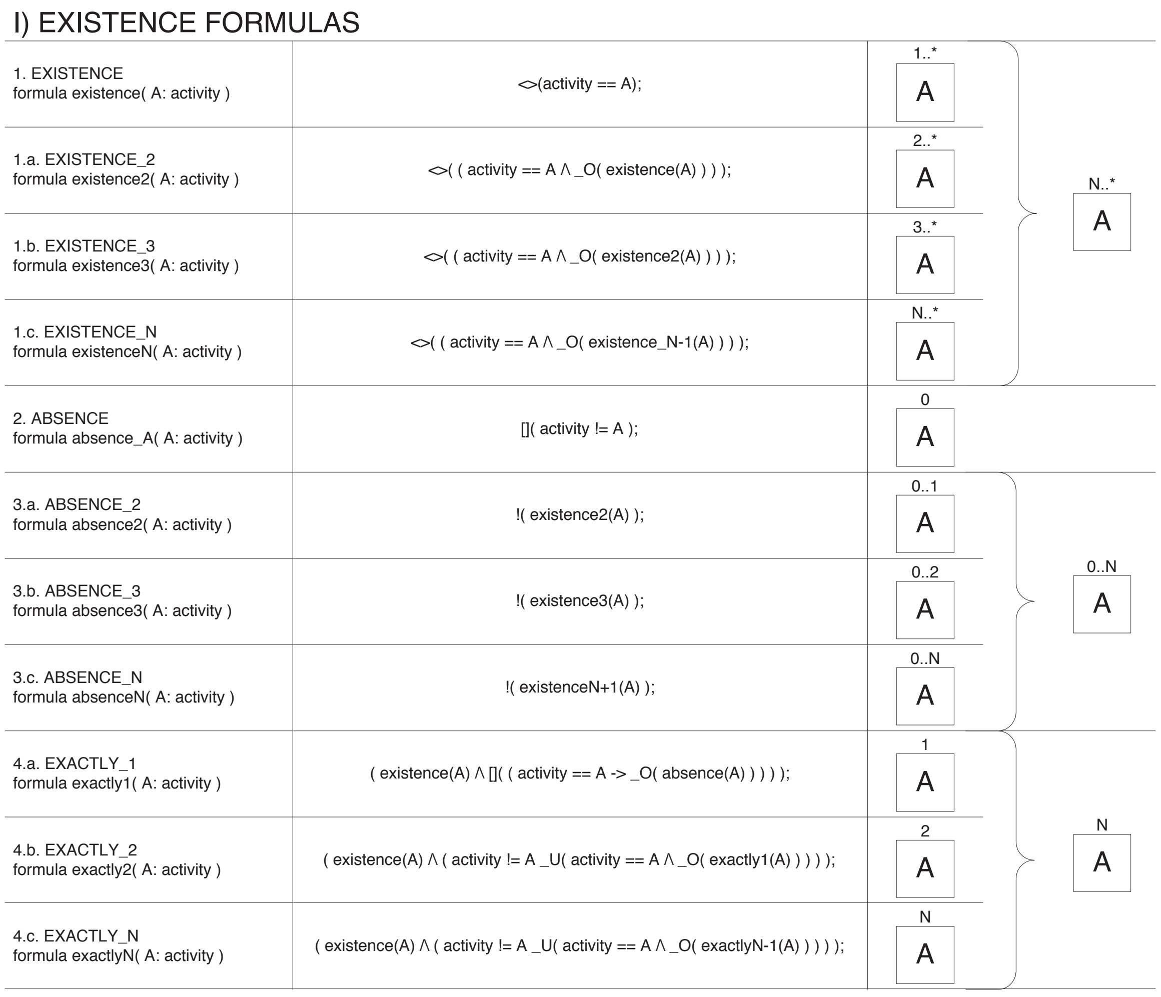


**Figure 3.7:** Graphical workflow representation for declare proposed by [VDAP06].

Figure 3.7 shows a workflow in its graphical representation. A task is denoted by a box with a describing label. The constraints defining the execution order of the tasks are denoted by different kinds of arrows between the tasks. If the number of executions of a task is constrained this is indicated by numbers over the tasks. In Figure 3.7 task $B$ is required to be executed a minimum of 3 times (hence the 3) but can also be executed more often than that.

The different kids of constraints are presented in Figures 3.8, 3.9, 3.10 with their underlying LTL definition.

The Declare workflow management system can be used to precisely model workflows in a declarative fashion. The various constraints defined as LTL templates provide a flexible graphical notation and can be easily understood by end-user. Since the execution engine is designed to run the LTL formulas new constraints can be added.



I) EXISTENCE FORMULAS

| | | | |
|---|---|---|---|
| 1. EXISTENCE<br>formula existence( A: activity ) | <>(activity == A); | 1..* A | N..* A |
| 1.a. EXISTENCE_2<br>formula existence2( A: activity ) | <>( ( activity == A ∧ _O( existence(A) ) ) ); | 2..* A | |
| 1.b. EXISTENCE_3<br>formula existence3( A: activity ) | <>( ( activity == A ∧ _O( existence2(A) ) ) ); | 3..* A | |
| 1.c. EXISTENCE_N<br>formula existenceN( A: activity ) | <>( ( activity == A ∧ _O( existence_N-1(A) ) ) ); | N..* A | |
| 2. ABSENCE<br>formula absence_A( A: activity ) | []( activity != A ); | 0 A | |
| 3.a. ABSENCE_2<br>formula absence2( A: activity ) | !( existence2(A) ); | 0..1 A | 0..N A |
| 3.b. ABSENCE_3<br>formula absence3( A: activity ) | !( existence3(A) ); | 0..2 A | |
| 3.c. ABSENCE_N<br>formula absenceN( A: activity ) | !( existenceN+1(A) ); | 0..N A | |
| 4.a. EXACTLY_1<br>formula exactly1( A: activity ) | ( existence(A) ∧ []( ( activity == A -> _O( absence(A) ) ) ) ); | 1 A | N A |
| 4.b. EXACTLY_2<br>formula exactly2( A: activity ) | ( existence(A) ∧ ( activity != A _U( activity == A ∧ _O( exactly1(A) ) ) ) ); | 2 A | |
| 4.c. EXACTLY_N<br>formula exactlyN( A: activity ) | ( existence(A) ∧ ( activity != A _U( activity == A ∧ _O( exactlyN-1(A) ) ) ) ); | N A | |

**Figure 3.8:** Existance Formulas with graphical representation. Figure from [VDAP06].

II) RELATION BETWEEN EVENTS FORMULAS

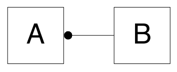
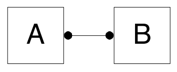
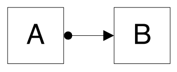
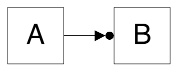
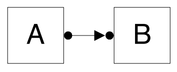
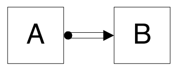
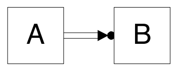
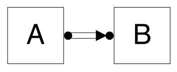
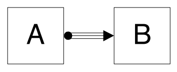
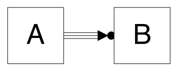
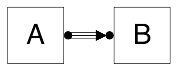
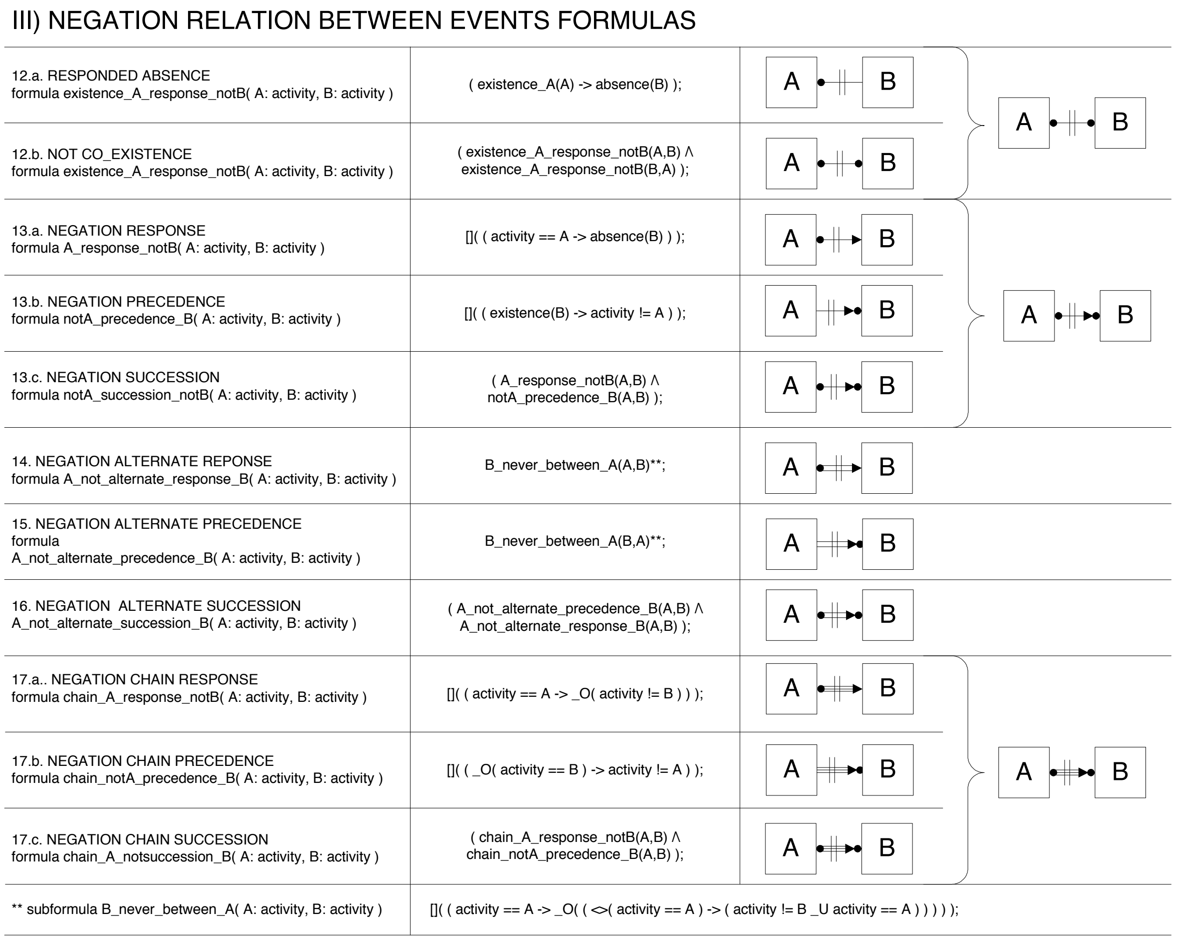
| | | |
|---|---|---|
| 1. RESPONDED EXISTENCE<br>formula existence_A_response_B( A: activity, B: activity ) | ( existence_A(A) -> existenceA(B) ); |  |
| 2. CO-EXISTENCE<br>formula co_existence_A_and_B( A: activity, B: activity ) | ( existence(A) <-> existence(B) ); |  |
| 3. RESPONSE<br>formula A_response_B( A: activity, B: activity ) | []( ( activity == A -> existence(B) ) ); |  |
| 4. PRECEDENCE<br>formula A_precedence_B( A: activity, B: activity ) | ( existence_A(B) -> ( !( activity == B ) _U activity == A ) ); |  |
| 5. SUCCESSION<br>formula A_succession_B( A: activity, B: activity ) | ( A_response_B(A,B) /\ A_precedence_B(A,B) ); |  |
| 6. ALTERNATE RESPONSE<br>formula A_alternate_response_B( A: activity, B: activity ) | ( A_response_B(A,B) /\ B_always_between_A(A,B)* ); |  |
| 7. ALTERNATE PRECEDENCE<br>formula A_alternate_precedence_B( A: activity, B: activity ) | ( A_precedence_B(A,B) /\ B_always_between_A(B,A)* ); |  |
| 8. ALTERNATE SUCCESSION<br>formula A_alternate_succession_B( A: activity, B: activity ) | ( A_alternate_precedence_B(A,B) /\ A_alternate_response_B(A,B) ); |  |
| 9. CHAIN RESPONSE<br>formula chain_A_response_B( A: activity, B: activity ) | A_response_B(A,B) /\ []( ( activity == A -> _O( activity == B ) ) ); |  |
| 10. CHAIN PRECEDENCE<br>formula chain_A_precedence_B( A: activity, B: activity ) | ( A_precedence_B(A,B) /\ []( ( _O( activity == B ) -> activity == A ) ) ); |  |
| 11. CHAIN SUCCESSION<br>formula chain_A_succession_B( A: activity, B: activity ) | ( chain_A_response_B(A,B) /\ chain_A_precedence_B(A,B) ); |  |
| * subformula B_always_between_A( A: activity, B: activity ) | []( ( activity == A -> _O( A_precedence_B(B,A) ) ) ); | |

**Figure 3.9:** Relation formulas with graphical representation. Figure from [VDAP06].



III) NEGATION RELATION BETWEEN EVENTS FORMULAS

| | |
|---|---|
| 12.a. RESPONDED ABSENCE<br>formula existence_A_response_notB( A: activity, B: activity ) | ( existence_A(A) -> absence(B) ); |
| 12.b. NOT CO_EXISTENCE<br>formula existence_A_response_notB( A: activity, B: activity ) | ( existence_A_response_notB(A,B) Λ<br>existence_A_response_notB(B,A) ); |
| 13.a. NEGATION RESPONSE<br>formula A_response_notB( A: activity, B: activity ) | []( ( activity == A -> absence(B) ) ); |
| 13.b. NEGATION PRECEDENCE<br>formula notA_precedence_B( A: activity, B: activity ) | []( ( existence(B) -> activity != A ) ); |
| 13.c. NEGATION SUCCESSION<br>formula notA_succession_notB( A: activity, B: activity ) | ( A_response_notB(A,B) Λ<br>notA_precedence_B(A,B) ); |
| 14. NEGATION ALTERNATE REPONSE<br>formula A_not_alternate_response_B( A: activity, B: activity ) | B_never_between_A(A,B)**; |
| 15. NEGATION ALTERNATE PRECEDENCE<br>formula<br>A_not_alternate_precedence_B( A: activity, B: activity ) | B_never_between_A(B,A)**; |
| 16. NEGATION ALTERNATE SUCCESSION<br>A_not_alternate_succession_B( A: activity, B: activity ) | ( A_not_alternate_precedence_B(A,B) Λ<br>A_not_alternate_response_B(A,B) ); |
| 17.a.. NEGATION CHAIN RESPONSE<br>formula chain_A_response_notB( A: activity, B: activity ) | []( ( activity == A -> _O( activity != B ) ) ); |
| 17.b. NEGATION CHAIN PRECEDENCE<br>formula chain_notA_precedence_B( A: activity, B: activity ) | []( ( _O( activity == B ) -> activity != A ) ); |
| 17.c. NEGATION CHAIN SUCCESSION<br>formula chain_A_notsuccession_B( A: activity, B: activity ) | ( chain_A_response_notB(A,B) Λ<br>chain_notA_precedence_B(A,B) ); |
| ** subformula B_never_between_A( A: activity, B: activity ) | []( ( activity == A -> _O( ( <>( activity == A ) -> ( activity != B _U activity == A ) ) ) ) ); |

**Figure 3.10:** Negation relation formulas with graphical representation. Figure from [VDAP06].

### 3.4.5 extended Declare

To create the Workflow examples presented in this thesis a modified version of the Declare workflow modeling system is used. Leymann, Unger and Wagner [LUW10] [Wag10] extended the Declare system to support temporal constraints. This makes it possible to model properties like the parallel execution or the same start time for two tasks. Since no graphical notation for the temporal constraints is clearly defined in either references the graphical workflow notation described in [VDAP06] (3.8, 3.9, 3.10) was extended to support temporal constrains inspired by [Wag10]. Some concepts introduced in [Wag10] like branching which is not necessary in declarative flows was not included.

Figure 3.11 condenses the extended Declare language and can be used as a quick reference to interpret the workflows presented in this thesis. The figure displays all different types of constraints and a short explanation in prose.

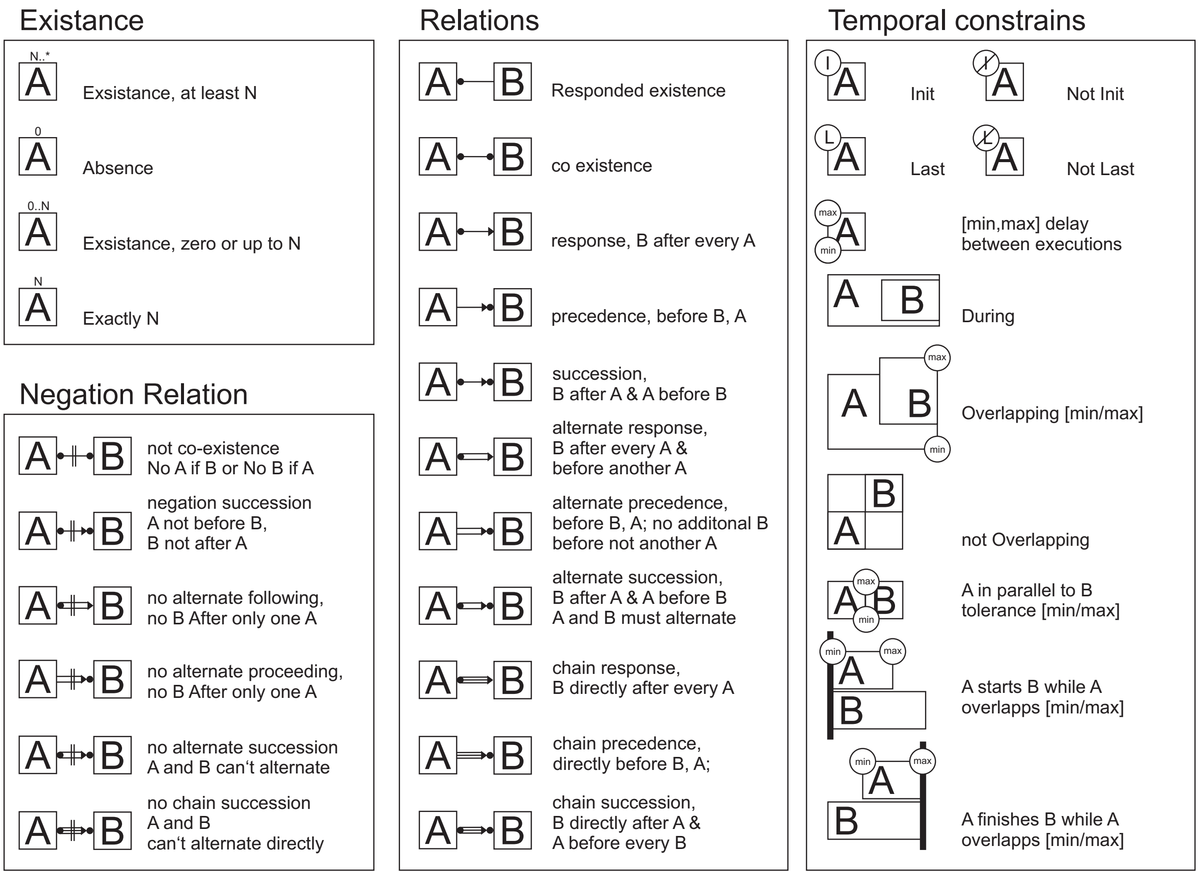


**Figure 3.11:** The extended Declare Language based on [VDAP06] and [Wag10].

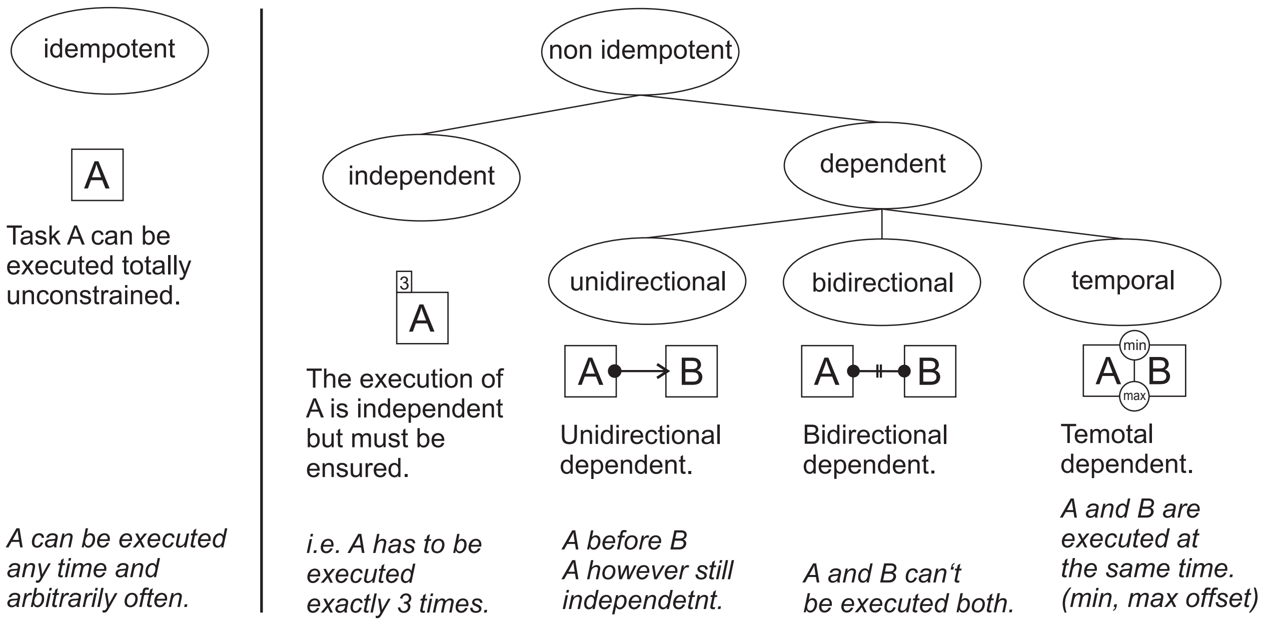

idempotent
A
Task A can be executed totally unconstrained.
A can be executed any time and arbitrarily often.
non idempotent
independent
dependent
3
A
The execution of A is independent but must be ensured.
i.e. A has to be executed exactly 3 times.
unidirectional
A B
Unidirectional dependent.
A before B A however still independetnt.
bidirectional
A B
Bidirectional dependent.
A and B can't be executed both.
temporal
min
A B
max
Temotal dependent.
A and B are executed at the same time. (min, max offset)

**Figure 3.12:** Task types in declarative languages.

## 3.5 Task types in declarative languages

This section presents the different task types that can be identified in declarative languages. The task types are displayed in a hierarchical order in Figure 3.12. Their impact on replication is discussed in Section 3.5.

- **Idempotent** tasks are tasks that can be executed at any time arbitrarily often. Such tasks do not need another tasks to be executed before or after them. They usually represent mandatory or optional tasks like optional checks or preparations.

  For example if you prepare for running you need to check if your shoelaces are tightened. It doesn't matter when you do it but you need to do it at least once.

- **Non Idempotent** tasks have some constraints or properties that do not allow an arbitrary number of executions. Such tasks in most cases change their environment. Sometimes this change can be so severe that an execution cannot be undone. Non Idempotent tasks can be further divided into dependent and independent tasks.

  – **Dependent** tasks are somehow connected to another task regarding their total or temporal order.

    ∗ **Unidirectional** dependent tasks have a connection to another tasks but the other task is not further constrained. For example if task A can only be executed before B this affects only the way task A is executed. Task B can still be executed at any time.

    The filesystem on a partition can only be repaired before the partition is mounted. The partition can however be mounted at any time and is not related to the repair task.

    * **Bidirectional** dependent tasks affect the way the other tasks is executed. Non coexistence of two tasks would be an example.

      For example if the color to paint something must be poured into an airbrush only one of the available colors can be selected. If the tasks pour color A has been executed the task pour color B cannot be executed too.

    * **Temporal** dependencies define how two or more tasks need to be executed in temporal relation to each other. Possible examples are parallel execution or same start time.

      An example for this would be to start to measure the time when another process is started.

  - **Independent** tasks are tasks that are not related to other tasks but are limited in the number of possible executions.

    If a software wants to gather information about the satisfaction of the user with the software this should only be asked once or twice because the user might get upset if he is disrupted several times and even has to enter the same information several times.

Chapter 4

# Problem Formulation

This chapter defines the basic terminology used in this thesis, describes the system model, elaborates the problem, and gives an overview over the approach.

## 4.1 Terminology

- A **Workflow** consists of a set of tasks and specifies how this tasks need to be executed. Conditions and relations between the tasks define how they are executed. They determine the order of the tasks as well as the number of times the tasks needs to be executed. To successfully execute a workflow all conditions and relations must be fulfilled.
- **Tasks** are atomic components that combined with relations and conditions make up a workflow. A task provides a specific functionality to the workflow and cannot be split up any further. A task is either executed successfully or fails. The execution of a task can however be dependent on an external service or data received from another entities. This can make the task execution vulnerable to errors and failures of resources or services outside of the tasks reach. Resulting from its nature and context in the workflow a task can either be idempotent or non idempotent. See Section 3.5 for more information about task types.
- The **Semantic of a Workflow** refers to its meaning. It is defined by the tasks, their relations, and execution conditions. When they are changed by intent or accident the semantic of the workflow changes. Another way to think of this could be that every workflow addresses a specific problem or describes a certain functionality. Changing any of the workflows properties changes the semantic of the workflow. Just like the meaning of a sentence is changed when changing the punctuation or words within it.
- The **Execution Order** of a workflow refers to the order in which the tasks defined in the workflow are executed. Since a declarative workflow only specifies what needs to be done and not how, a specific execution order is not predefined. This means that the tasks can be executed in different orders without violating the semantic of the workflow. For more information refer to Section 3.2.

- **Workflow Replication** means that a workflow is run several times in parallel to increase robustness. As described earlier not all tasks defined in a workflow can be run several times and so the execution of such tasks need to be coordinated.
- A **Replica** is copy of a workflow.

## 4.2 System Model

This thesis considers replication to make the execution of workflows in distributed environments robust against the typical difficulties. Before the system model is presented these difficulties are highlighted:

- **Unreliable Nodes** Targeting distributed systems with nodes ranging from smart phones, laptops, and workstations to external web services the failure of one node is not an unlikely scenario. The methods presented in this thesis thus have to take into account that nodes or the replicas running on them can fail at any time.
- Nodes may be mobile and **run on battery power**. This can result in a limited lifetime that is also influenced by their system load and the communication overhead. Since communication is expensive for wireless devices in in terms of energy consumption it should be kept to a minimum and the replicas should run as independent as possible.
- Due to **network characteristics** the latency and throughput, and structure of the network might be subject to immediate change. While some nodes are connected to a wired network others might be connected via wireless communication services over a shared medium.
- **Services** might be bound to network nodes and their availability might change depending on the condition or presents of this nodes.
- Some tasks might require access to **External Services** with variating availability. In this context an external service is understood as a service provided by a third party and not by a local node.

### 4.2.1 Failure Models

This thesis focuses on the following failure scenarios:

- **Replicas can Fail** at any time. Failed replicas need to be detected and handled by the system. If the replica was running a non idempotent task the task needs to be executed by another replica.
- **External Services** can be unavailable. To simulate such services tasks with a variating availability are used.

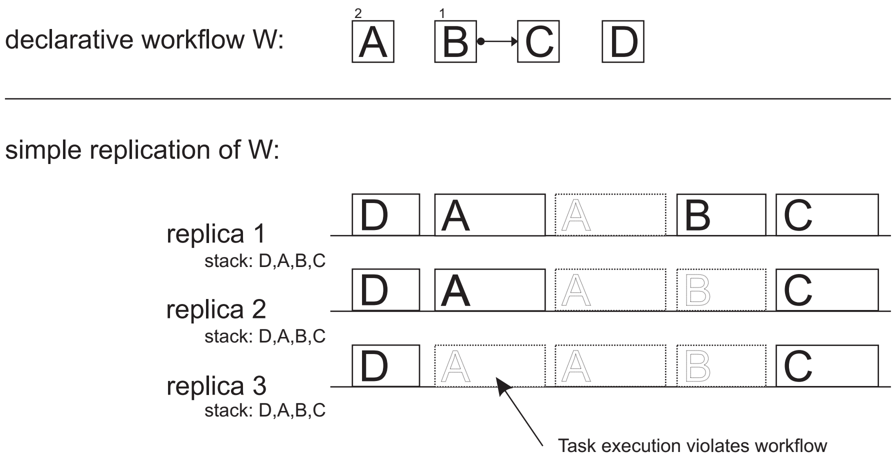


**Figure 4.1:** Simple replication approach.

## 4.3 Problem Statement

This Section describes the problems addressed by this thesis.

As described in Section 4.2 distributed and especially pervasive systems can be a very unreliable environment and present certain difficulties that a software running on such systems needs to account for. One of the biggest occurring problems is spontaneous failure of nodes. Data stored only on one node of a distributed system simply vanishes from the system when the node fails.

Existing work (See Section 2) has proposed many approaches addressing this issues such as transactions, backups, or surrogates for failed services.

Backing up data to other nodes of the system can help to overcome this difficulty. This so called replication is a well known architectural paradigm not only for computer systems. Replicating of workflows is thus nothing entirely new. Replication of web services for the use in Ad-hoc networks has been proposed by Dustdar and Juszczyk in [DJ07].

This thesis approaches replication of workflows from a language point of view. Since the traditionally used imperative languages [PWZ+12] are thought of to be inflexible and rigid [FLM+09] declarative languages are researched as a programming paradigm for workflows in distributed systems.

The first problem that needs to be solved is to find a suitable declarative language that provides the necessary flexible and formal background to precisely define workflows for distributed systems. Available workflow languages have been presented in Chapter 3.

Since not all tasks in a workflow are the same possible differences between them had to be researched (See Section 3.5) and their influence regarding replication had to be analyzed (See Section 5.2).

Figure 4.1 shows a workflow W and a simple approach to replicate it. The workflow is defined using the extended Declare language introduced in Chapter 3. It consists of four tasks

$(A, B, C, D)$. Unfortunately only two tasks ($C$ and $D$) are idempotent and can be executed arbitrarily often. Additionally task $B$ needs to be executed before task $C$.

Before the workflow can be executed an actual execution order has to be chosen. For this example the order will be $D \rightarrow A \rightarrow A \rightarrow B \rightarrow C$. Please note that task $A$ has to be executed exactly two times.

In the replication approach shown in Figure 4.1 all three replicas start executing the workflow and execute task $D$. Since task $D$ is idempotent this is absolutely fine. The next task is task $A$ which is limited to two executions. When all the replicas execute task $A$ only once the semantic of the workflow is already violated. A similar problem applies to task $B$. This displays the basic coordination problem that needs to be solved when replicating a workflow. The arising problems that need to be solved are as follows:

- How can the non idempotent tasks be coordinated?
- In distributed systems communication overhead can be a critical issue. How can the replication be accomplished with minimal communication overhead?
- The example shows that replicas are idle while a non idempotent task is executed. Can this idle time be used to speed up the overall execution of the workflow? ... And can declarative languages help to achieve this?
- If a replica fails how can it be detected and how does the replication system need to react?
- The nodes in a distributed system might execute the workflow at different speed. Will this be a problem or can this knowledge even be used to increase reliability?
- How can the flexibility of declarative languages benefit replication?
- What what are the key points for an efficient replication strategy?
- How well do replication strategies work in the real world?

## 4.4 Approach Overview

To address these issues the workflow is run on several nodes in parallel. If a replica fails the execution of the workflow needs to be able to continue until even the last replica has failed. Non idempotent tasks are run on a single replica while other replicas stand ready to take over in case the replica executing the non idempotent task fails. To coordinate the execution and to synchronize the state of the replicas messaging is used.

One of the replicas is elected to decide about the execution of non idempotent tasks and becomes an active coordinator. The state of this coordinator is synchronized to backup coordinators to avoid a single point of failure. For small systems all the replicas could act as a backup coordinator.

To detect the failure of replicas a timeout based error detection protocol is used.

The thesis will show that replication can also help to speed up workflow execution. This can be achieved by replicating the workflow in different orders and executing repetitive tasks in parallel.

Chapter 5

# Workflow Replica Coordination

Replication is a basic architectural paradigm to improve reliability, fault-tolerance or accessibility of technical systems. The basic idea of replication is to prevent total system failure by having backup systems standing by to take over if necessary. The Redundancy Array of independent Disks (RAID) System is a popular example. In terms of redundancy the RAID system proposes two different approaches. The first approach is mere mirroring of two disks. The second approach uses a block level striping strategy with distributed parity to enhance performance and provide robustness against single disk failures.

Unfortunately simple duplication is not possible for workflows since non idempotent tasks can only be executed as often as defined. See Section 3.5. This thesis researches methods to coordinate the execution of workflow replicas to deal with this problem. It will also show that clever replication can decrease the needed execution time of a workflow and increase the overall performance.

The basic ideas of workflow replication are introduced by replicating a simple computational workflow. In this computational workflow the basic task types identified in Section 3.5 (idempotent and non idempotent tasks) are present. This first example is used to identify and explain the basic replication issues.

After that the impact of idempotent and non idempotent tasks on replication is discussed in more detail and methods for their replication are proposed. Based on this a replication system is designed. Issues like messaging, data handling, task execution, and failure detection are explained.

To further increase the performance the benefits of replicating a workflow in different orders are discussed. It is demonstrated that this increases the robustness against the failure of external services while reducing the number of idle replicas and execution time.

The chapter is concluded with an example that describes how a declarative workflow is executed on three replicas step by step.

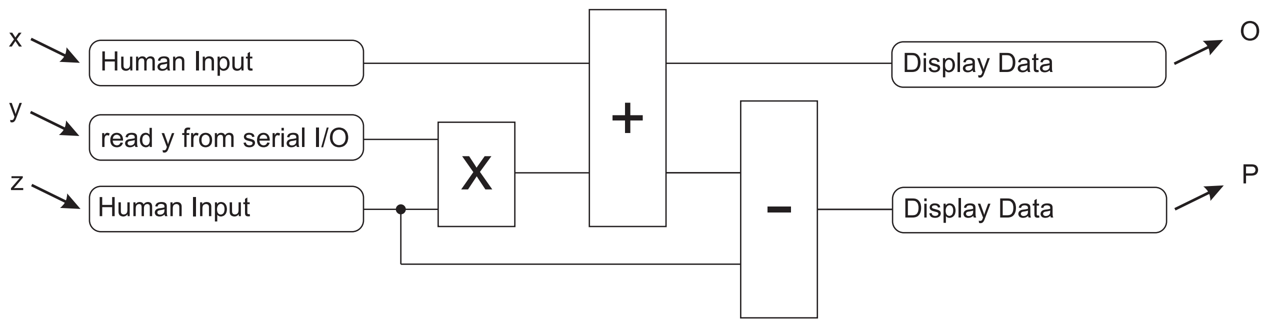


**Figure 5.1:** A computational workflow

## 5.1 Introductive Example

The development of cybernetic models is nowadays done with the aid of graphical block diagramming tools. Such models are the basis of developing and testing ECUs (electronic control units) which have a large range of applications and can be found in any modern machine. The desired functionallity is modeled using graphical blocks representing either computational or I/O tasks.

Using such a high level prototyping tool the developer can focus on functionality and does not need to fiddle around with the constraints of different hardware platforms. This is a good real life example how workflows are used as a programming paradigm to reduce complexity. This section introduces the issues of workflow replication looking at such an rather computational example workflow like shown in Figure 5.1.

The workflow consists of several I/O and commutation blocks. The variables x, y, and z are read from different inputs and used in the different computation blocks. The result of each computation is passed on to the next block until it reaches an output block. In this basic workflow example the two major types of tasks identified in Section 3.5 can be found.

- Computational blocks represent **idempotent tasks**. They only modify data within the system (local data) having no influence whatsoever outside of the system. This kind of tasks will only changes local data. Having no influence on the "outside world" such tasks don't need to be coordinated and can be executed anytime by any replica.
- I/O Blocks represent **non idempotent tasks**. They interact with other processes, systems, or even the real world. Such tasks modify data or real objects which are not within the reach of the system and cannot be undone easily. The data they receive on or without request can be subject to continuous change. Even reading information from a shared database twice can thus yield different results. Interactions with a human user bear similar problems. The user might not be delighted to enter login information several times just because the process in the background is replicated and all the processes are asking for the needed information on their own. Having such an impact not only on local data the execution of such tasks must be coordinated.

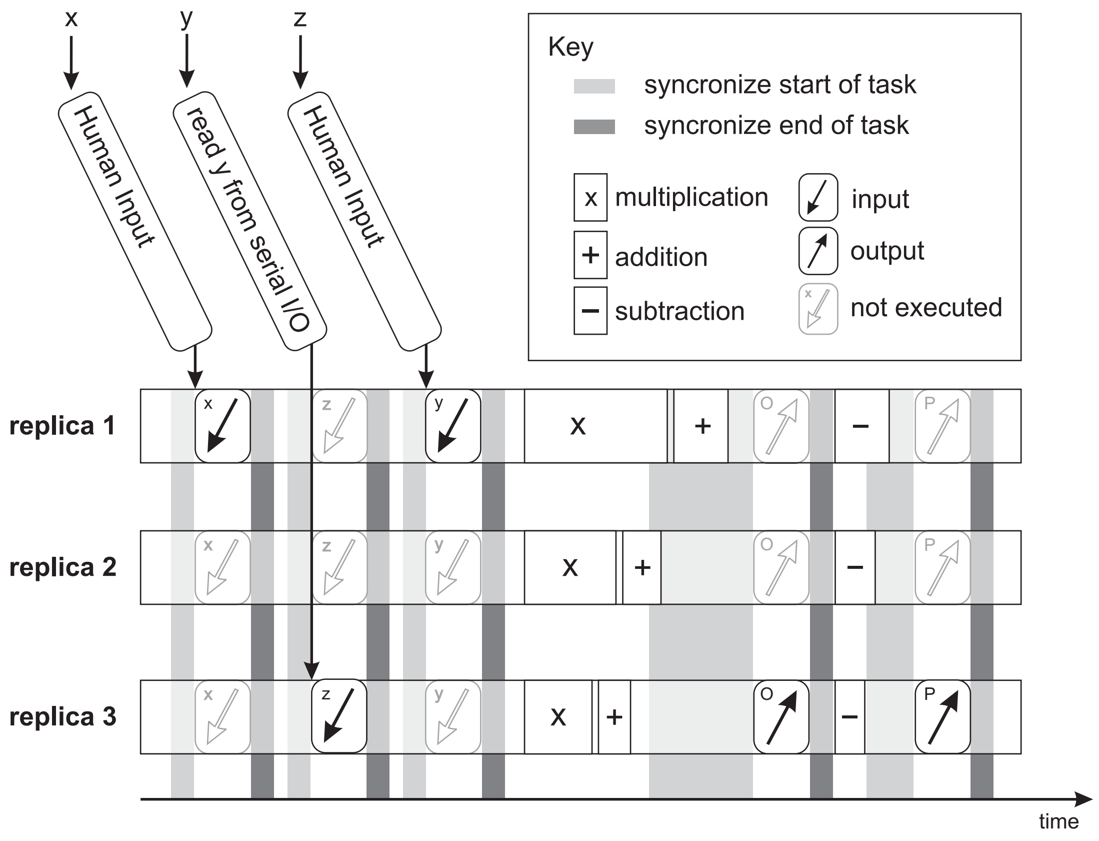


**Figure 5.2:** Execution of a replicated workflow

Figure 5.2 shows the basic idea of coordinating replicas running a copy of the workflow presented in Figure 5.1. In this example the workflow is replicated three times. The state of the replicas is synchronized before and after the execution of a non idempotent task. If a replica is not ready the others have to wait to avoid assynchronism. To avoid communication overhead the execution of the idempotent tasks is scheduled by each replica separately. To focus on basic coordination it is assumed that all tasks are executed successfully, all messages are delivered successfully, and no replicas fail.

Before executing the first non idempotent task (Input $X$) the replicas synchronize and agree on replica 1 to execute this task. After the successful execution the replicas synchronize again and replica 1 sends the result to the other replicas. The execution continues and the next two non idempotent tasks are executed in the same way (Input $Y$ and $Z$).

All the replicas now have the necessary information to run the upcoming idempotent tasks represented by the computations. Since idempotent tasks can be executed arbitrarily often their execution is scheduled by each replica on its own to reduce communication overhead. Due to differing computational resources or other external influences the execution speed of the replicas may differ. In Figure 5.2 this is depicted by the different lengths of the idempotent tasks.

Before the next non idempotent task can be executed all replicas need to synchronize again. This means replicas two and three have to wait for replica one to finish. Following this strategy the rest of the workflow is executed step by step.

Considering this naive setup the main issues of coordination can be summarized as follows. Each of them will be addressed by a separate section later in this Chapter.

- **Need for coordination:** As pointed out in Section 3.5 for non idempotent tasks parallel execution on several replicas is in general not possible. I.e. if a task books a flight that can only be done by one replica because otherwise one would end up having as many flights booked as there are replicas. The execution of such tasks must thus be coordinated. This is addressed in in Section 5.3.3.3.
- **Different execution speed:** As pointed out in the previous example the time needed for executing certain tasks may differ between the replicas and cause a delay when replicas have to wait idle for others to finish. This problem is addressed by an adequate coordination strategy discussed in Section 5.3.3.3.
- If a task is dependent on an **external service** this can also have influence on the workflow execution. An external service can for example be the booking system of an airline which is directly accessed by a task and needed for successful completion. If such a service is unavailable over a longer period of time this could significantly influence the duration of the workflow execution or even make it impossible. A detailed example is presented in Section 5.4.0.2.
- **Idle replicas:** In the example above non idempotent tasks are executed on one replicas while the others are idle. While it is good to have a replica standing clear to take over in the case of a failure the other replicas could be used to execute another task. This would also speed up the execution time of a the workflow. Section 5.4 discusses how the flexibility of declarative languages (See Section 3.2 ) can be used to reduce the number of idle replicas.
- **Errors and failures:** One of the original reasons for replication was to make the overall system more reliable and resilient against failures. Especially but not only when running on a network of loosely connected and unreliable nodes failure of a replica is a very likely scenario. Implementing a good error detection strategy having backup replicas stand by and scheduling tasks in a keen order are key elements to make the system robust. This will be discussed in Section 5.3.4.
- **Communication:** The messages needed to coordinate task execution and distribution of execution results are introduced in Section 5.3.2.
- **Data Handling:** Section 5.3.1 discusses how the workflow data is stored in a replica and how the information about task execution is organized. This is needed later in this chapter when explaining how a single replica processes the workflow and executes the tasks.

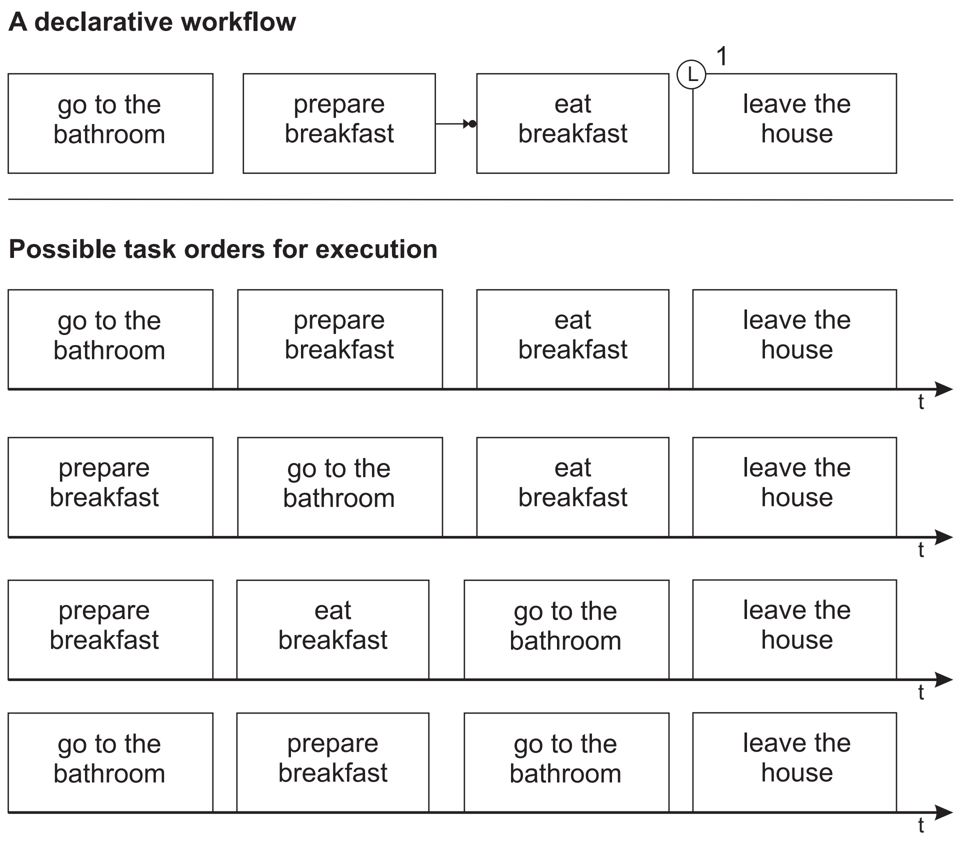


**Figure 5.3:** A workflow and its possible task orders.

## 5.2 Coordination of Different Task Types

In Section 3.5 different task types defined in declarative languages have been identified. This Section explains what influence this types have on coordination.

As explained in Chapter 3 in contrast to imperative languages declarative languages do not define an execution order for all tasks. When however a workflow is executed the tasks are executed in a particular order. Before a workflow can be executed a definitive order of the tasks has to be generated. This has to be done with respect to the existing constraints. While an idempotent task can be executed at any time other tasks need to be executed following an order precisely defined in the workflow.

Figure 5.3 shows the workflow of a morning schedule and possible orders of execution. Note that for a successful execution only the last task "leave the house" needs to be executed once. The other tasks do not need to be executed at all or can be executed more than once. Note that not all possible execution orders are depicted.

Generating a task order of a workflow can be a hard problem which is out of the scope of this thesis. When task orders are shown they are "generated" manually.

Such a predefined order is stored in the task-stack of a replica. See 5.3.1. A workflow usually translates into more then one task order.

The example above demonstrates how different task types influence the order in which the tasks of a workflow can be executed. Non idempotent tasks can further be categorized in one or more of the following subcategories (See also task types identified in Section 3.5):

- **Independent** tasks are very similar to idempotent except that they have to be executed a specified number of times. This can be a definitive number, a range, a minimum, or maximum of executions. Just like idempotent tasks they can be placed anywhere when generating the execution order and are thus pretty flexible. When executed they have to be coordinated since their number of executions must be guaranteed.
- The execution of **Dependent** tasks is related to the execution of other tasks. In terms of execution order this restricts the available positions for such a task in the execution order. Dependent Task can be either unidirectional or bidirectional dependent.
  - **Unidirectional dependent** tasks are tasks that need to check if another task was executed before them or make sure that a specific task is executed after them. In the this case only one of the tasks is affected, the other one remains unconstrained. The task prepare breakfast and eat breakfast in the scenario presented above are a good example. While a breakfast needs to be prepared before eating it, it can be prepared without eating it as shown in the last execution order. The eating task is thus unidirectional dependent on the preparing task.
  - **Bidirectional dependent** constraints like the "no coexistence" need some additional coordination. Before executing one of the tasks the replica has to check if the related task has been scheduled for execution or has already been executed.
- Tasks bound to a **temporal constraint** are a little more difficult to deal with. Some have influence on the order of the tasks others need some temporal coordination when executed. The last constraint used in the example described above only influences the task order while for example parallel execution needs to be handled at run time.

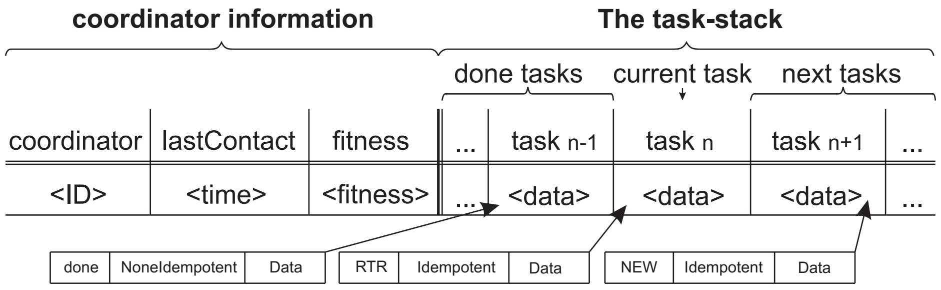


**Figure 5.4:** Data stored by an ordinary replica

## 5.3 Workflow Coordination System

This Section proposes a coordination system that addresses the issues identified in Section 5.1. Starting with the data handling within the replicas, the messages interchanged between them, task execution and coordination algorithms, and failure detection strategies are explained.

A coordinator will be introduced to coordinate the distributed execution of a workflow and ensure that its semantic (See Section 4.1) is not violated. To avoid a single point of failure every replica is basically the same but only one replica is chosen to coordinate the execution while others serve as backup. See Section 5.3.4.

### 5.3.1 Data Handling

To enable workflow replication every replica needs to store some information.

As described in Section 5.2 an execution order of the tasks in a workflow needs to be generated before execution. This task order is then stored in the task-stack of a replica. See Figure 5.4. The replica starts to execute the workflow beginning with the first task on the stack. Every task in this stack has a data field holding execution information related to that task like **Status, TaskType, and TaskData**. See Figure 5.5. When initialized the status of a task is "NEW" and changes according to the current execution state of the task.

The possible states of a task can be:

- **NEW:** The *NEW* status indicates that a task has not been run. This is the initial state.
- **RTR:** This only applies to non idempotent tasks. The status of a non idempotent task is *Ready To Run* when the replica is waiting for clearance from the coordinator to proceed with the execution.
- **CTR:** This only applies to non idempotent tasks. When a CTR message (See 5.3.2) is received by a replica the replica sets the state of a task to *Ready To Run* enabling execution.
- **done:** Indicates that a task has been executed.
- **running:** Indicates that the task is currently being executed.

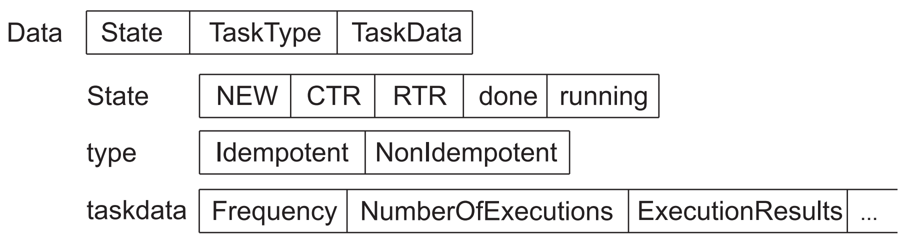


**Figure 5.5:** The data field of the task-stack

The **TaskType** filed (See Figure 5.5) indicates if the task is non idempotent (and has to be coordinated) or idempotent and can be run without coordination.

The **TaskData** field contains the following information. (See Figure 5.5):

- **Frequency:** The frequency field contains the number of times a task needs to be executed.
- **NumberOfExecutions:** This field stores the number of times this task has already been executed successfully for tasks that need to be executed more than once. If a *Task Executed Successfully* message is received this number is increased. Before execution of a task it is compared with the *Frequency* field. If the required number of executions is already reached the task status is set to*Done* and not started by the replica.
- **ExecutionResults** The execution results field contains the actual data resulting from the execution of a task. This data is also updated by received *Task Executed Successfully* messages.

#### 5.3.1.1 Global vs. Local Queue

When coordinating the replicas in a distributed system the question rises how to keep track of the execution progress of all tasks at all replicas to decide about execution. There are two basic possibilities. Either all replicas have a global overview over the execution progress (Global task queue) of every replica or only store their own progress (Local task queue). Both scenarios are depicted in Figure 5.6.

If progress information is stored globally the coordinator needs to maintain an active copy of the current global state on ever replica. See Figure 5.6. This might be possible when the replication system only consists of a small number of replicas but can significantly increase communication overhead when maintaining a large number of replicas.

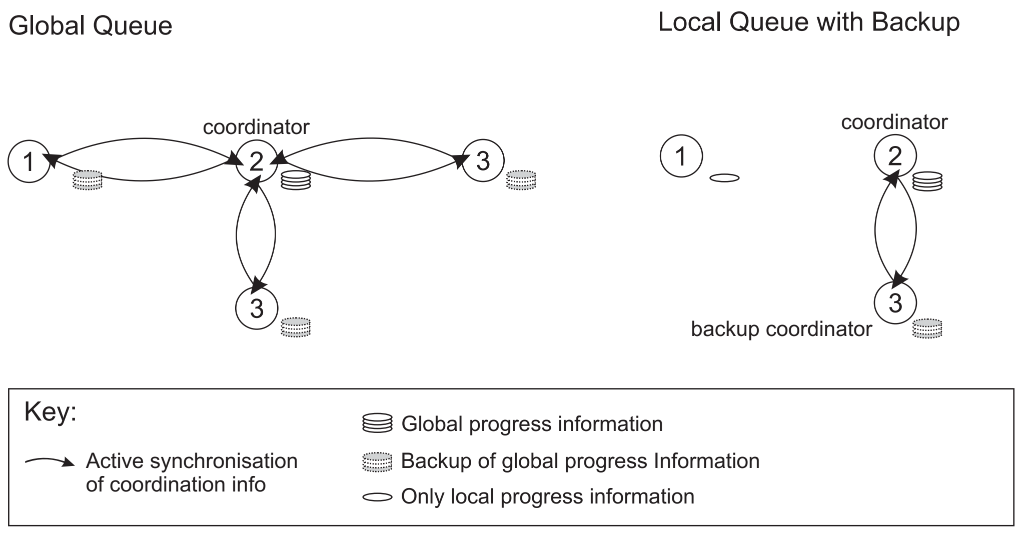

Global Queue
Local Queue with Backup
coordinator
1
2
3
3
coordinator
1
2
3
3
backup coordinator
Key:
Active synchronisation of coordination info
Global progress information
Backup of global progress Information
Only local progress information

**Figure 5.6:** Global Queue vs. local Queue

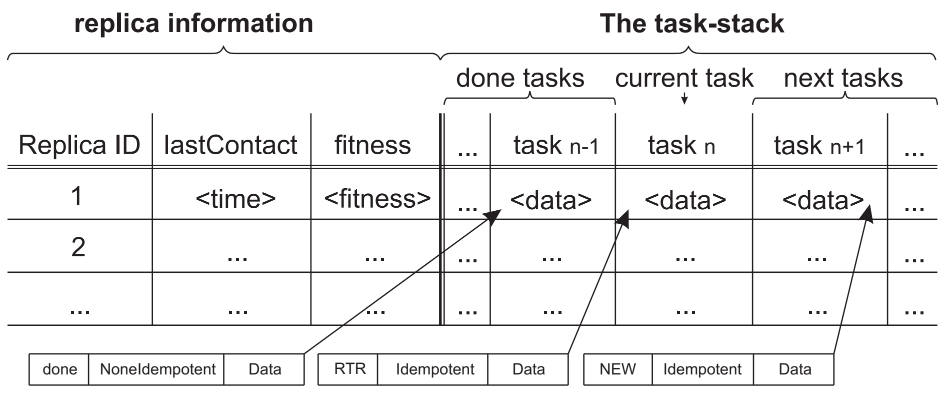


**Figure 5.7:** Data stored by the coordinator

Ordinary replicas (replicas being neither coordinator nor backup coordinator) only store the ID of the coordinator, the time of the last contact to the coordinator, and information of their own progress. See Figure 5.4.

The coordinator and backup coordinator also store progress information, time of last contact, and fitness information for every single replica to make task execution decisions. Figure 5.7 depicts the task-stack of the coordinator containing information of all replicas. Every column is dedicated to a different replica.

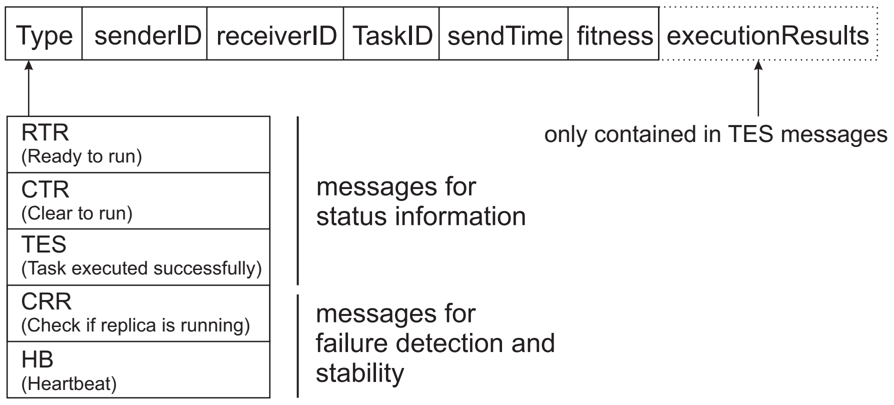

Type
senderID
receiverID
TaskID
sendTime
fitness
executionResults
only contained in TES messages
RTR
(Ready to run)
CTR
(Clear to run)
TES
(Task executed successfully)
CRR
(Check if replica is running)
HB
(Heartbeat)
messages for status information
messages for failure detection and stability

**Figure 5.8:** The generic structure of a message.

### 5.3.2 Messaging

To coordinate the distributed task execution status information and execution results need to be shared between the replicas. This information is transported using different types of messages. This Section will introduce the different message types used for communication and the algorithm that updates the task-stack with the received information.

All messages are structured like depicted in Figure 5.8. A message has a type indicating its purpose. It contains the ID of the sending and receiving replica, the ID of the task concerned, the time when the message has been sent, and the current fitness of the sending replica. The *Task Executed Successfully* message (TES) additionally contains a field with execution results.

The fitness information contained in the messages helps the controller to schedule the execution of non idempotent tasks (See Section 5.3.3.3). The send time of a message is part of the timeout based failure detection system. The timestamp helps the controller to determine if a replica is still alive and connected to the network. If no message was received in a certain timeframe a heartbeat message is requested. For more information regarding failure detection and error handling please see Section 5.3.4.

A message can be of one of the following types (defined by the type field of the message):

- **RTR:** The *Ready To Run* message is send by a replica to the controller to indicate that it is ready to execute a non idempotent task. The ID of the non idempotent task is contained in the message (taskID). The receiving controller sets the status of the current task of the corresponding replica (identified by the senderID in the message) to RTR. (See Section 5.3.1) The controller also updates the last seen information for the sending replica with the send time of the message. The fitness of the replica is also updated with the information contained in this message.
- **CTR:** The *Clear To Run* message is send from the coordinator to start a specified task at replicas listed in the targetReplicaIDs filed. This message follows the RTR message. Receiving replicas update their task-stack and execute the corresponding task. The last

seen and fitness information transmitted in the sendTime and fitness fields is used to update the information of the sending replica (the controller).

- **TES:** The *Task Executed Successfully* message is used to distribute the execution results. A replica sending the TES message has just executed a task successfully and distributes this information along with the execution results using the TES message. A receiving replica updates its task-stack and checks if the execution of this task is done. If a task has to be executed several times before execution is completed the replica decreases the number of remaining executions by one. The last seen timestamp for the sending replica along with the fitness value is also updated.
- **CRR:** The *Check Replica Running* message requests a heartbeat from the receiving replica. Receiving replicas update their last seen and fitness information for the sending replica and response with a HB message.
- **HB:** The *HeartBeat* message is used to respond on a CRR message. It contains information about the currently running task, the send time, the ID of the sender and the fitness indicator. Receiving replicas update their last seen and fitness information for the sending replica.

**Algorithmus 5.1** Handling incoming messages

```
procedure updateStackWithReceivedMessages
    while $MessageQueNotEmpty$ do

        $M$ ← getNextMessageFromQueue();
        updateLastSeenInformation(M.senderID, M.lastSeen);
        updateFitnessInformation(M.senderID, M.fitness);

        if M.type == CRR then
            sendHeartbeatMessageToSender($M.senderID$);
        end if
        if M.type == CTR then
            setTaskClearToRun($M.taskID$);
        end if
        if M.type == TES then
            updateTaskData(M.taskID, M.TaskData);
        end if

        if thisReplicaIsCoordinator then
            if M.type == RTR then
                setReplicaReadyToRun(M.ReplicaID, M.taskID);
            end if
        end if

    end while
end procedure
```

### 5.3.3 Task Execution and Workflow Coordination

The previous Sections introduced the communication between replicas and the way the replicas handle the workflow data and received coordination information. This Section discusses how the system decides which replica executes non idempotent task. Subsection 5.3.3.2 introduces the algorithm managing task execution on each replica. The coordination algorithm used on the coordinating replica is discussed in Subsection 5.3.3.3.

#### 5.3.3.1 Coordination Decissions

As introduced in Section 5.1 the execution of non idempotent tasks has to be coordinated. This means that somehow it has to be decided which replica executes a specific non idempotent task. Obviously if there is more than one entity involved in the decision process there is a need for communication. In a distributed replication system running on a couple of mobile nodes communication can be quite expensive in terms of energy consumption and should thus be reduced as much as possible.

The question behind this is how can the number of decisions involving more than one entity be reduced to reduce the need for communication. The first step was not to coordinate the execution of all tasks since idempotent tasks could simply be executed on all replicas in parallel. To avoid a need for communication to decide which replica is allowed to execute a non idempotent tasks this thesis uses a coordinator.

While the replicas still need to ask the coordinator for permission there is no more need for a distributed decision algorithm to be invoked to decide about the execution of each non idempotent task. When a replica is ready to execute such a task it just signals this to the coordinator and waits to receive either the allowance to execute it or the result of the task distributed by another replica.

To make them interchangeable all the replicas use the same data structure and algorithms (and are thus basically the same). Before the workflow execution is started they agree on one replica to coordinate the execution. To avoid a single point of failure backup coordinators are selected being ready to take over if the coordinating replica fails.

This does not totally remove the need for distributed decisions but in the best case such a decision is only needed at the beginning to select a coordinator and a backup. If the coordinator fails and the backup coordinator takes over a new backup coordinator could simply be chosen by the "old" backup coordinator to avoid further decisions involving all replicas. It is out of the scope of this thesis to develop yet another decision algorithm for distributed systems. An existing distributed decision strategy can be selected when needed. A paper discussing distributed decision processes has been discussed in the related work. See Chapter 2.

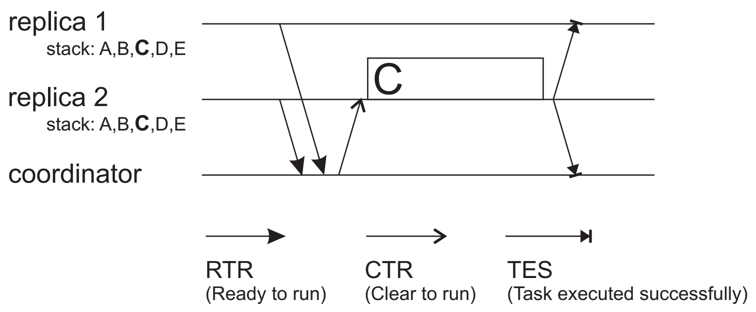


**Figure 5.9:** Basic workflow coordination. In the first step the replicas signal RTR to the coordinator. In the second step the coordinator decides which replica should execute the task. In the last step the executing replica sends a TES message to distribute the results.

#### 5.3.3.2 Task Execution

As introduced in Section 5.3.1 every replica maintains a task-stack containing the tasks it has to execute. The replica starts executing a workflow beginning with the task at the top of the list.

If the task is non idempotent the replica sends a *Ready To Run* message (RTR) to the coordinator. If no other replica has started the task jet and the fitness of the replica is sufficient the coordinator sends a *Clear To Run* message (CTR) to the replica. When the replica received the clearance from the coordinator it immediately starts executing the task. If the task is idempotent it can be executed without asking the coordinator for permission. When a task has been executed successfully the replica sends a *Task Executed Successfully* message (TES) to all other replicas containing the execution result of the task.

The basic task execution of a non idempotent task is depicted in Figure 5.9 and described in Algorithm 5.2. Constrained tasks are requested before they are run. If a task has been executed it is checked if it has been executed the required amount of times. If not its state is set back to NEW and the number of done executions is increased.

**Algorithmus 5.2** Task execution

```
procedure TASKEXECUTION
    while $ReplicaIsRunning$ do

        UPDATESTACKWITHRECEIVEDMESSAGES();
        $T \leftarrow$ GETCURRENTASKFROMSTACK();

        if T.state == CTR then
            RUN($T$);
            SENDTES($T$);
        end if

        if T.state == NEW then
            if T.type == non idempotent then
                SENDRTR($T$);
            else
                RUN($T$);
                SENDTES($T$);
            end if
        end if

        if T.numberOfExecutions < T.specifieNumberOfExecutions then
            SETSTATENEW($T$);
            INCREASENUMBEROFEXECUTIONS($T$);
        else
            SETTASKDONE($T$);
            SETCURRENTTASK($T.next$);
        end if

    end while
end procedure
```

#### 5.3.3.3 Task Coordination

As described above the execution decisions for non idempotent tasks are made by a coordinator. The purpose of the coordinator is to ensure the correct execution of all tasks contained in the workflow to maintain its semantic (See Section 4.1). The coordinator is just a regular replica that runs an additional coordination algorithm. (See Algorithm 5.3.)

The coordinator processes the incoming messages just like an ordinary replica (See Algorithm 5.1) and stores the information as described in Section 5.3.1. In addition to the progress information about himself he has to store the progress information of all other replicas and maintain a global view like described in Section 5.3.1.1. This information is stored in an advanced task-stack that holds the additional information about other replicas. This extended task-stack is depicted in Figure 5.7.

After the task-stack of the coordinator has been updated he compiles a list of replicas that are ready to run non idempotent tasks. Based on that list replicas ready to run a task are enabled to run taking the remaining number of executions for this task into account.

The last step of the coordination algorithm is to check the last contact filed of every replica to check if a replica seems to be dead. Failure detection is described in detail in Section 5.3.4.

Note that this is just a coordination schema. In a real world system an additional commit protocol would have to be added to provide for mid transaction failure and message loss. This has been left out to keep the depiction of the coordination as clear and clean as possible.

**Algorithmus 5.3** Replica coordination

```
procedure CoordinateReplicas
    while ReplicaIsRunning do

        updateStackWithReceivedMessages();
        runList ← generateListOffTasksNotDone();

        for T in runList do
            replicaList ← generateListOfReplicasReadyToRun(T);
            sortListByReplicaFitness(replicaList);
            startAppropriateNumberOfTasks(replicaList);
        end for

        checkForDeadReplicas();

    end while
end procedure
```

### 5.3.4 Failure Detection and Error Handling

To detect failed replicas a lazy timeout based error detection system is used. Since messages have to be passed on regular basis from all replicas to the coordinator the timestamp in the message is used to keep track of the replicas. If a replica remains silent for an unusual long time the coordinator sends a CRR message (See Section 5.3.2.) to the suspicious replica. If no heartbeat is received within a certain timeframe it must be assumed that the replica is dead or not reachable.

The coordinator on the other hand has to send status updates and coordination messages on a regular basis and can thus be monitored by the other replicas. If the coordinator remains silent for an unusual long time the replicas take the initiative and check if the coordinator is still alive. A random back off time could be used to avoid killing the coordinator with a message storm.

To avoid a single point of failure the coordinator is replicated within the system to a backup coordinator. This backup coordinator can take over when the main coordinator has crashed.

The lazy failure detection strategy proposed in this Section for replica coordination avoids message explosion since it uses the regular coordination messages and thus avoids flooding. It even can remain silent if no problems occur.

The related work section discusses a paper comparing different error detection strategies in distributed systems. (See Chapter 2.)

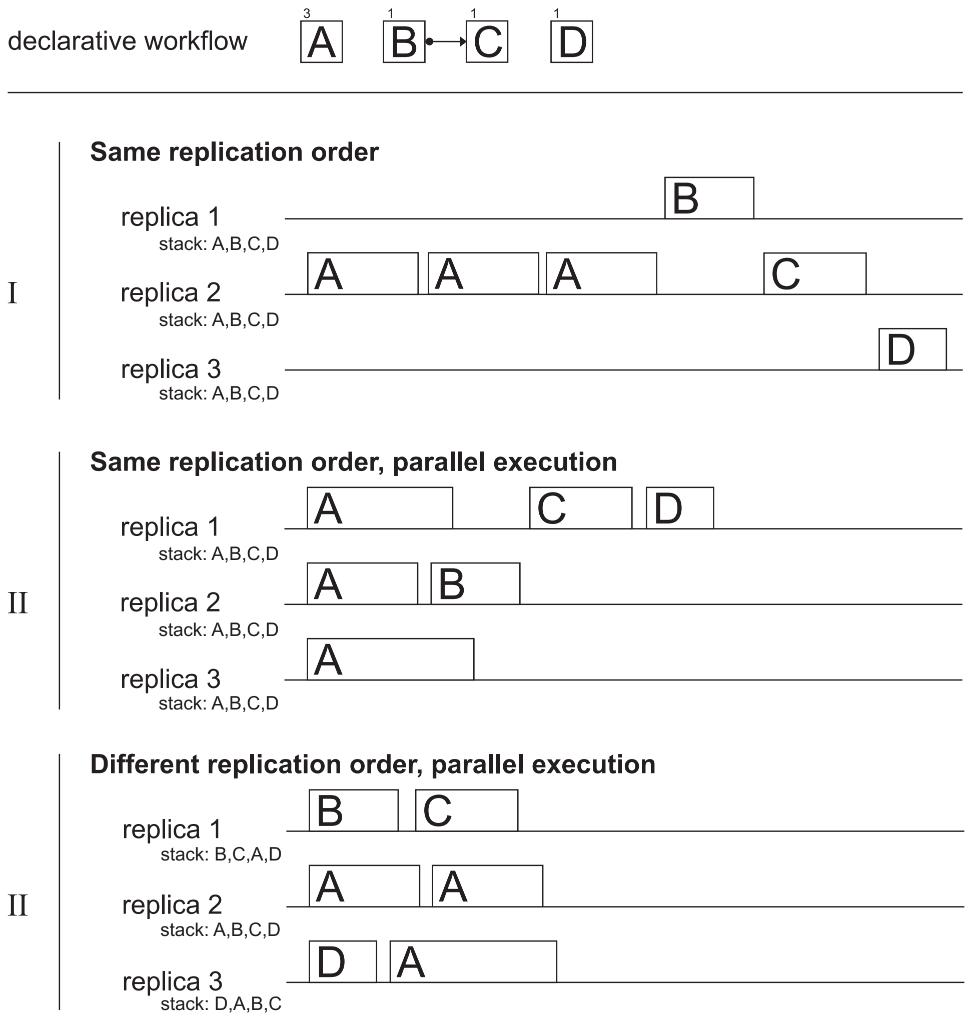


**Figure 5.10:** Benefits of using different task orders when replicating a workflow.

## 5.4 Execution Order

As mentioned above a declarative workflow does not define a fixed order of its tasks. Before the workflow can be executed a concrete task order has to be generated and loaded into the task-stack of the replicas. See Section refsec:taskTypeCoordination. Instead of choosing one of this orders for replication on all replicas a different order could be used for each replica. This would have the following advantages:

#### 5.4.0.1 Reducing the Number of Idle Replicas and Speedup the Workflow Execution

Figure 5.10 shows how replicating a workflow in different orders can reduce the execution time of a workflow. Under each replica its task-stack is depicted to show the replicated task order.

On the line after the replica name only the tasks actually executed by the specific replica are shown.

In example I all replicas replicate the workflow in the same order. Without any parallelization the execution will take as long as it task one replica to execute the whole workflow. Since most of the tasks are non idempotent they can only be executed on one replica. In effect replica 1 and replica 2 are idle most of the time.

In example II task $A$ which is supposed to run three times is run on all replicas in parallel reducing the time required to execute the workflow. Despite this improvement two of the replicas are still idle most of the time because they have to wait for replica 1 to finish executing tasks $C$ and $D$.

In example III every replica replicates the workflow in a different order. Replica 1 starts executing task $B$, replica 2 starts executing task $A$ and replica 3 starts with task $D$. After successful execution the results are synchronized. Using this strategy no replica remains idle and the workflow can be executed in a fraction of the time originally required. If a replica would fail during execution another replica could take over. This exemplifies that replicating a workflow in different orders can reduce the amount of idle replicas, speed up the workflow execution process, and still be fault tolerant.

#### 5.4.0.2 Robustness Against External Service Failure

Another advantage of replicating a workflow in different orders is that it becomes more resistant against the failure of an external service. This is depicted in Figure 5.11. If replicating a workflow in the same order on every replica the replicated system could be vulnerable to the failure of an external service which is required by a task of the workflow. Example I (in Figure 5.11) shows that in the worst case even the execution of the whole workflow could fail if the external service remains unavailable. In an average case the execution could be slowed down dramatically if the execution of the whole workflow is forced to wait in that external service.

Replicating the workflow in different orders (as shown in example II Figure 5.11) the workflow can still be executed successfully if the external service becomes unavailable.

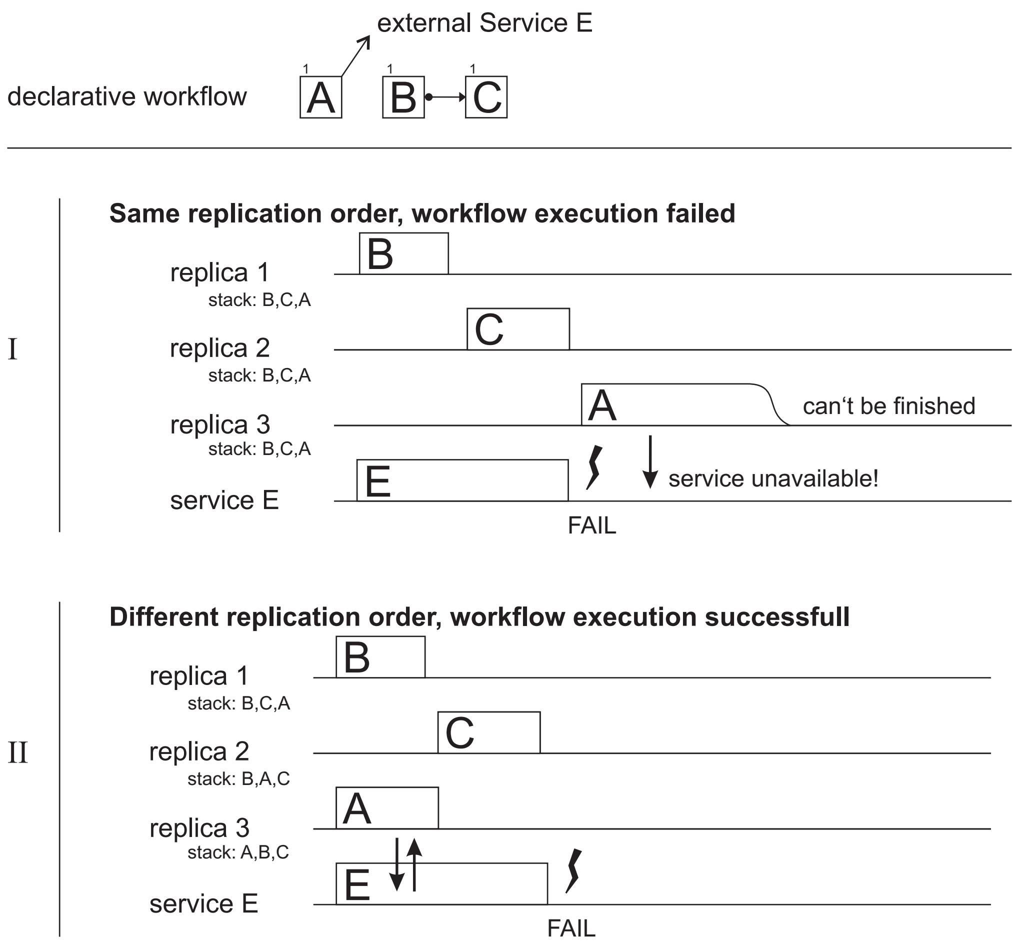


**Figure 5.11:** Replicating a workflow in different orders can provide robustness against the failure of an external service.

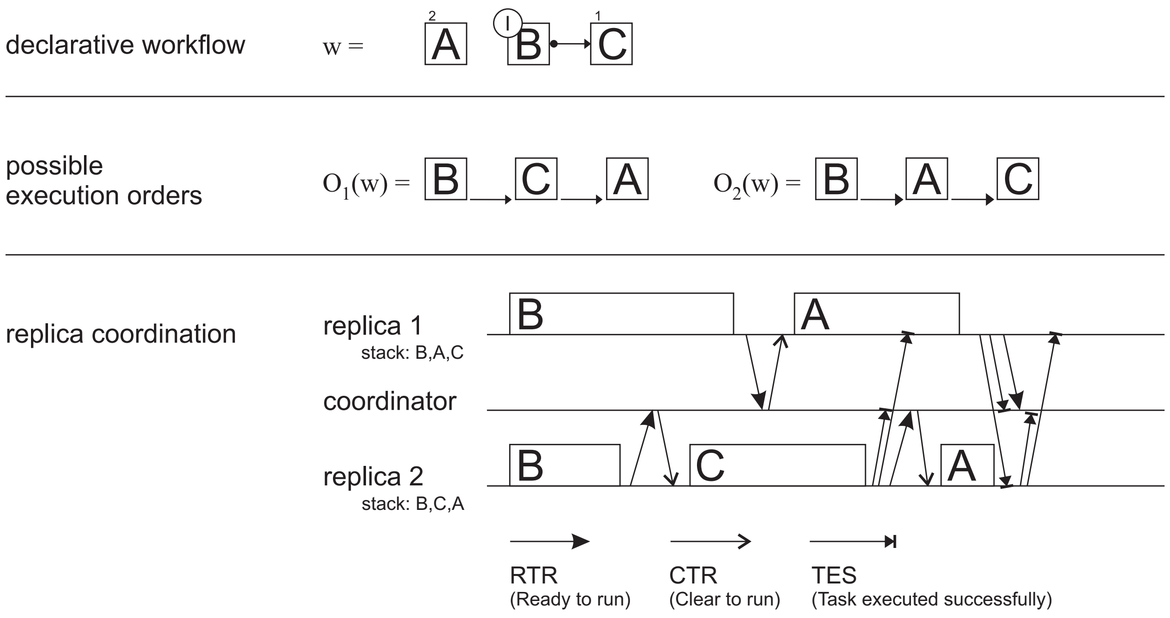


**Figure 5.12:** A concrete declarative workflow is replicated and executed on two replicas. The coordinator usually running as a service on one of the replicas is depicted separately to avoid confusion.

## 5.5 Example

After describing methods of coordination in Section 5.3.3.2 and defining the algorithm via pseudocode this Section presents a concrete execution scenario. The replicas use a set of messages to communicate described in detail in Section 5.3.2. For more information about data handling please see Section 5.3.1.

The workflow $(W)$ proposed in this example can only be executed in two different orders. Order $O_1(w) = B \rightarrow C \rightarrow A$ and order $O_2(w) = B \rightarrow A \rightarrow C$. See Figure 5.12. Both orders are replicated on different replicas. The benefits of using different execution orders have been discussed in Section 5.4.

Task $B$ has been defined as initial task and has to be executed first. Task $C$ is defined as a response to task $B$: $\Box((activity == B \rightarrow existence(C)))$. Since task $C$ needs to be executed exactly once: $existence(C) \wedge \Box((activity == A \rightarrow \bigcirc(absence(C))))$ it just has to be executed anytime after task $B$. Task $A$ needs to be executed exactly two times: $(existence(A) \wedge (activity \neq A \cup (activity == A \wedge \bigcirc(exectlyN - 1(A)))))$ and can be executed as second or third task. For information about the notation see Section 3.4.5.

**Workflow execution:**

The replicas start executing task $B$ without coordination since it is an idempotent task and does not need to be coordinated. Being faster in general replica 2 executes task $B$ faster and sends a $RTR(taskC)$ message to the coordinator which responds with a $CTR(taskC)$ message. While replica 2 is executing task $C$ replica 1 finished executing task $B$ and sends a $RTR(taskA)$ message to the coordinator which responds with a $CRT(taskA)$ message.

As replica 2 finishes task $C$ it sends a $TES(taskC)$ message to the other replicas (including the here separately depicted coordinator). After that replica 2 requests execution of task $A$ which is granted by the coordinator sending an $CTR(taskA)$ to replica 2. While replica 2 executes task $A$ replica 1 has finished one execution of the same task and notifies all other replicas with a $TES(taskA)$ message.

From replica ones point of view one execution of task $A$ is missing and thus replica 1 sends a $RTR(taskA)$ message to the coordinator. Since replica 2 is already running task $A$ for the second time this is just noted by the coordinator. If replica 2 would fail while executing task $A$ replica 1 can immediately be ordered to take over the execution. After finishing task $A$ replica 2 sends a $TES(taskA)$ message to all replicas. The workflow has been executed successfully.

Chapter 6

# Evaluation

This chapter evaluates the coordination methods proposed in the previous chapter. The evaluation will determine the fitness of the algorithms regarding failures, communication overhead, and workflow characteristics. In order to do that the proposed strategies have been implemented to be simulated using the PeerSim peer to peer simulator. See section 6.1.

The replication algorithms will be evaluated using a large number of synthetically generated workflows and simulated in the peer to peer simulator PeerSim. To draw conclusions about the significance of this synthetic tests to the real world a "real" workflow example is replicated.

The implemented replication protocol needed for the simulation has a size of 1700 lines of java code. Since this is a quite invisible amount of work contained in this thesis a short architectural overview is given in Section 6.1.2.

## 6.1 Evaluation Methods and Environment

### 6.1.1 PeerSim

To evaluate the proposed algorithm the PeerSim simulator is used. See [MJ09]. PeerSim is a peer to peer simulator written in java. It provides two different simulation engines. A more scalable, and simplistic cycle-based engine that is ignoring transport layer issues like message loss and delay. An event based, however less efficient engine that can among other things be used to take transport layer obstacles into account.

Focusing on the synchronization methods the cycle based engine has been used to implement and simulate the proposed coordination algorithm. It is assumed that messages are transmitted reliably between the nodes.

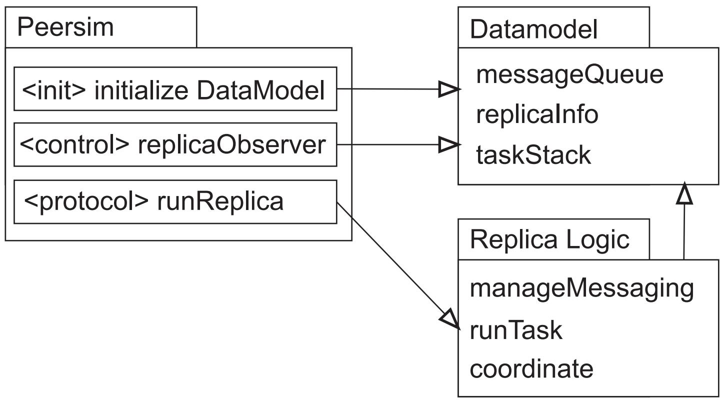


**Figure 6.1:** Architectural overview over the replication protocol.

### 6.1.2 Implemented Simulation Software

This section briefly presents the simulation software that was implemented as a protocol for the PeerSim simulator to generate the evaluation results. PeerSim is controlled using a config file that is used to specify the network size (the numbers of peers) and the protocols to run on the peers. To initialize the simulation environment initializers are used. In contrast to protocols initializers are only called once in the very beginning of the simulation. The actual simulation process is controlled by so called controls. They are used to end the execution and execute maintenance and monitoring tasks which are not part of the protocol that is simulated.

Figure 6.1 shows the simplified architecture of the replication protocol. A replica is represented in the PeerSim simulator as network node. The coordination algorithms (Replica logic) is implemented as a PeerSim protocol. Every node runs the "manageMessaging" and "runTask" algorithms. The coordination algorithm however is only run by one node (the coordinator). To store task information and coordination data the data handling described in Section 5.3.1 is implemented in the "Datamodel". Every node maintains a separate data model where its task-stack and coordination information is stored. A message queue buffers all incoming messages and works as described in Section 5.3.2.

The data model is initialized using the initializeDataModel initializer which evaluates the settings from the config file. A workflow is passed to the replicas using the configuration file or the workflow generator. The input via config file is intended to simulate a specific workflow like the later presented kitchen workflow. See Section 6.2.5. The workflow generator can be used to generate a specifically tailored set of different workflows. This is needed when simulating a test case for hundreds of different workflow configurations.

Using the cycle driven engine all nodes are called every cycle one by one and execute the defined protocols. As described in Section 5.3.3.2 non idempotent tasks are coordinated before execution and idempotent tasks can be executed directly without coordination. To simulate task complexity every task has a duration property. The duration property models of "how many instructions" a task consists. A replica has a fitness parameter that basically defines how many instructions can be processed per cycle. In combination these two parameter model the duration of the execution of a task on a specific replica.

### 6.1.3 Simulation Parameters

To configure the workflow generator the config file is used. The following parameters can be used to influence the workflow generator:

- **sameOrder:** If true the workflow will be replicated in the same order on every replica. If not defined the value true is assumed.
- **numberOfDifferentWorkflows:** Defines how many different orders of a workflow are replicated. This parameter must be specified.
- **numberOfTasks:** Defines of how many tasks the generated workflow will consist. If not defined the default value 10 is used.
- **{min,max}Duration:** The duration of all tasks will be randomly chosen from this range. Defaults are min 1 and max 5.
- **{min,max}executions:** Minimum and maximum number of executions of all tasks will be randomly chosen from this range. Defaults are min 1 max 3.
- **percent idempotent:** Percentage of idempotent tasks.
- **number of failing replicas:** Number of replicas that will fail during execution of a workflow. The system tries to distribute the fails equally over the execution time. By default no replicas will fail.
- **numberOfServices:** Defines the number of external services. By default no external service dependencies are simulated.
- **{min,max}serviceAvailability:** define the minimum and maximum chance of all the external services to be available. When the workflow is generated the probability is randomly chosen individually for each replica from this range. The availability of every service is separately determined based on its probability at the beginning of every cycle of the simulation.

The following options can be used to influence the way workflows are executed:

- **{min,max}fitness:** The fitness of a replica is randomly defined within this range.
- **taskList:** If the workflow generator is not used the *tasklist* input can be utilized to feed a specific workflow into the system for simulation. If not used the system uses a generated workflow.
- **networkSize:** This parameter defines the number of nodes in the simulation system. A node represents a replica. This parameter is not optional and must be defined.

- **experiments:** Simulating the execution of a workflow is called an experiment. The number of experiments conducted can be defined using the *experiments* parameter. When the workflow generator is used as an input a new workflow is generated for every experiment using the parameters set in the config file. If the parameter is not specified only one experiment is conducted.
- **maxParallel:** This option enables TES messages for idempotent tasks. When replicating a workflow in different orders this speeds up the execution since the other replicas do not need to execute the idempotent tasks when they have been already executed elsewhere. On the down side this increases the message overhead since additional TES messages have to be send which would not be necessary to successfully execute the workflow. If not defined the default value is true.

### 6.1.4 Synthetic Tests

To find out how the proposed system performs under different conditions the workflow generator is used to generate specifically tailored workflows. The quality of the generated workflows can be influenced by the parameters described in Section 6.1.3. This makes it possible to run every test case on a large number of different workflows.

The evaluation results presented for all the test cases in this section have been generated simulating 500 specifically generated workflows. Preliminary tests indicated that after such a large amount of simulations the resulting values have sufficiently converged to stable values. The statistics have been gathered by the replicaObserver (See 6.1) and written to a text file. The resulting data has been arithmetically averaged and plotted using gnuplot.

### 6.1.5 Real World Test

To not only rely on synthetic test cases a real workflow is simulated to test the system on a real example. Like the synthetic test cases this workflow is executed 500 times to take different failure situation into account. In contrast to the synthetic test cases the execution orders for the real world example are deduced by hand. The real world workflow is evaluated regarding execution speed, idle time, message overhead, and replica failure.

## 6.2 Evaluation

This Section presents the results of the different test cases. Most of the synthetic tests combine more parameters to show their close relation. For example speedup and idle time is combined to show how the speedup of the execution process is related to the actually working replicas. In most cases the diagrams on the left side show replication with the same task order on every replica and on the right side with a different task order for every replica.

At the beginning of each evaluation the key configuration options are pointed out. If values are modified during the experiment the values are presented in curly brackets. The default configuration is as follows:

**Default configuration:**

| newtork.size | Tasks | minExecutions | maxExecutions | minDuration | maxDuration |
|---|---|---|---|---|---|
| 10 | 10 | 1 | 4 | 1 | 5 |

| sameOrder | DifferentWorkflows | failingReplicas | externalServices | Experiments |
|---|---|---|---|---|
| {true, false} | 20 | 0 | 0 | 500 |

### 6.2.1 Speedup and Idle Time

This section evaluates the impact of replication on execution time of the workflow and idle time of the replicas regarding the percentage of non idempotent tasks, the number of tasks, and the number of replicas. To get a more fine grained impression of the influence of different workflow orders the workflow has been simulated using only one workflow, five different workflows, and a different workflow for every replica.

The evaluation of speedup and idle time has been combined in one simulation scenario to show their close relation. The diagrams show that a decreased execution time also means that the replicas have been less idle and were taken to better use.

As the configuration table shows Figure 6.2 evaluates speedup and idle time with a fixed number of tasks and a variable number of replicas. In Figure 6.3 it is the other way round. The number of tasks is variated while the number of nodes is kept constant.

**Configuration:**

| | newtork.size | Tasks | DifferentWorkflows | idempotent |
|---|---|---|---|---|
| figure 6.2 | {1, 3, 5, 12, 20} | 5 | {1, 5, max} | {0, 33, 66, 100} % |
| figure 6.3 | 10 | {5, 10, 15, 20, 25} | {1, 5, max} | {0, 33, 66, 100} % |

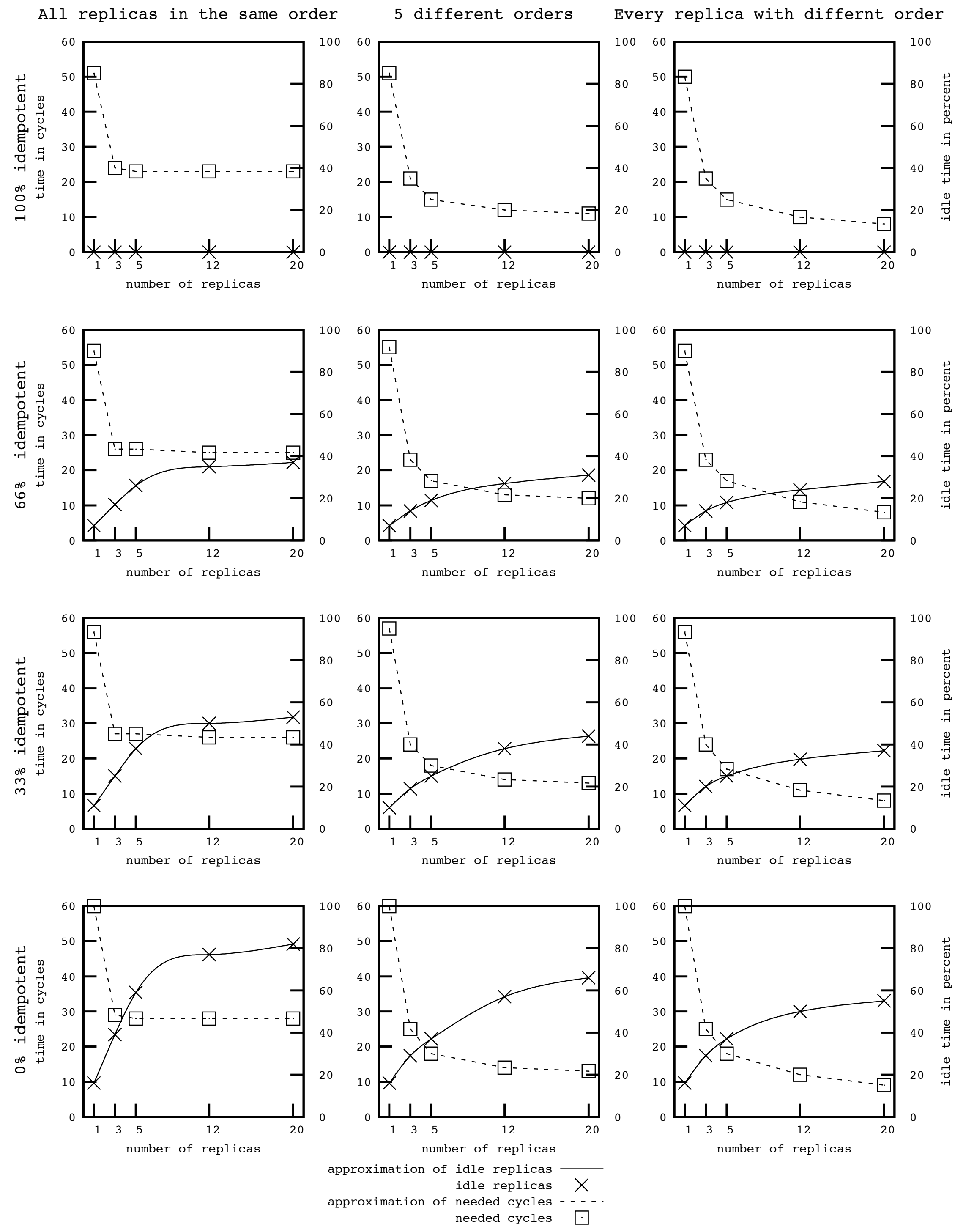


**Figure 6.2:** Speedup and idle time regarding number of replicas, task type, and order.

### Description of Figure 6.2

Figure 6.2 shows how increasing the number of replicas can benefit execution speed and idle time of the replicas while the number of tasks is constant (5). The diagrams in the left column are generated replicating the workflow in the same order on every replica. For the generation of the diagrams in the middle five different workflow orders are used. This means that when using twenty replicas four replicas replicated the workflow in the same order. In the right column a different order was used on every replica. The four rows represent different percentages of idempotent tasks.

**Speedup**
Replicating a workflow in the same order on every replica benefits the execution time only as long as the number of replicas is not greater than the number of times a task needs to be executed. This can be see in the left column. Since the number of times a workflow needs to be executed is randomly chosen between one and fife the execution speed does not decrease any further when more than 5 replicas replicate the workflow in the same order. When the workflow is however replicated in different orders the execution time can decrease further since different tasks are scheduled at the same point in the task-stack of the replicas.

The percentage of idempotent tasks has almost none effect on the execution speed of a workflow.

**Idle time**
In contrast to the execution speed the idle time is dependent on the number of the replicas used and the percentage of idempotent tasks. If 100 per cent of the tasks are idempotent no replicas are idle no matter of the number of used task orders. This is because idempotent tasks are executed on every replica and can be executed arbitrarily often. The idle time increases with a decreasing number of idempotent tasks.

When the workflow is however replicated in an increasing number of different orders the time of the replicas can be used more efficient and the number of idle replicas decreases. When the percentage of idempotent tasks however decreases the idle time of the replicas rises.

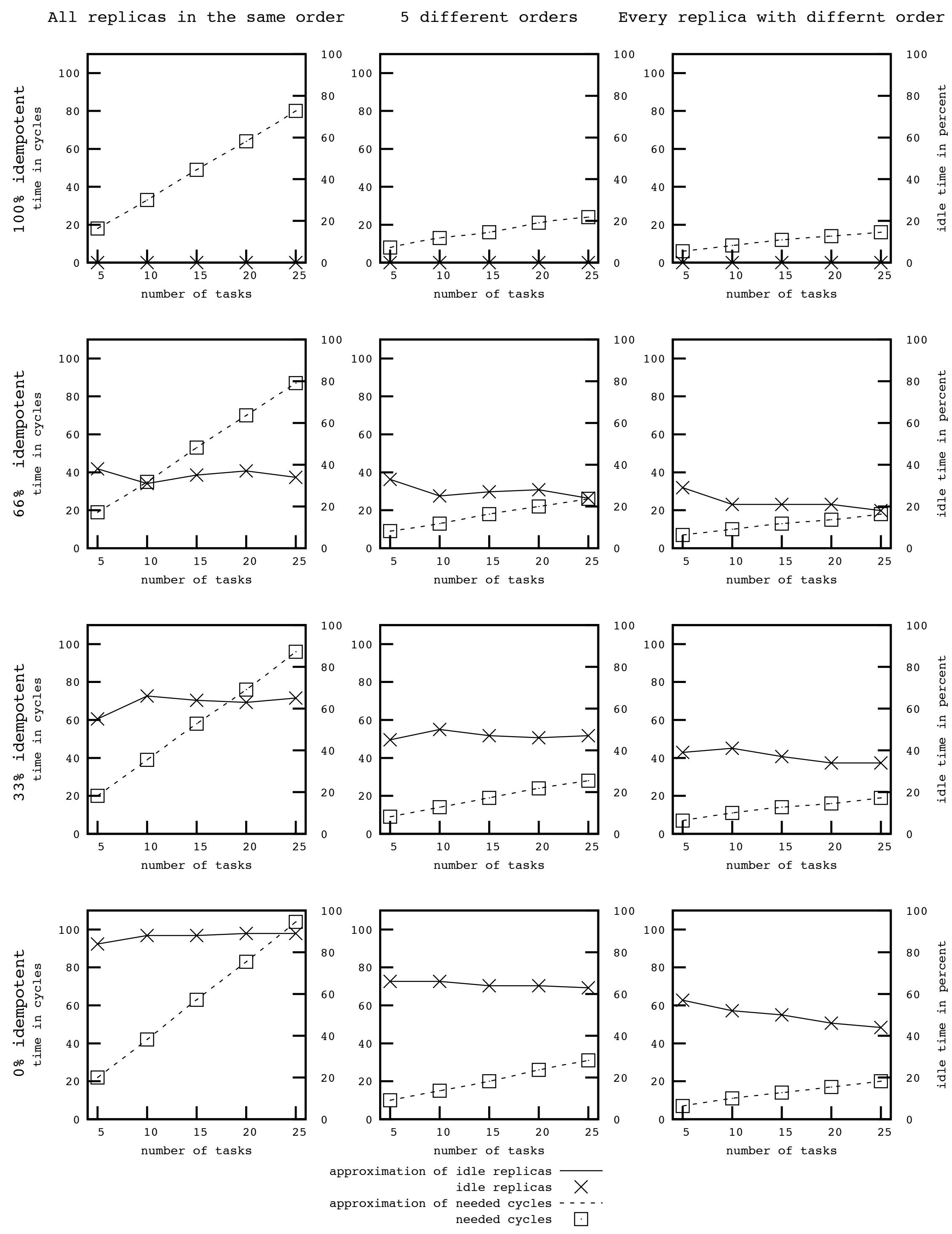


**Figure 6.3:** Speedup and idle time regarding number of tasks, task type, and order.

**Description of Figure 6.3**

Figure 6.3 shows how increasing the number of tasks influences the execution time and idle time of the replicas while the number of replicas is constant (10). Apart from that Figure 6.3 is organized in the same way as Figure 6.2.

**Speedup**
When replicating a workflow in the same order a linear increase of the tasks increases the execution time of a workflow in the same fashion. Replicating the workflow in different orders can not change this however the execution time does not increase as fast as it does when the workflow is replicated in the same order on every replica. The percentage of idempotent tasks has almost no influence.

**Idle time**
The number of idle replicas is only related to the percentage of idempotent tasks and stays almost constant when increasing the number of tasks. The only exception to this poses the very last diagram. The reason for this is that if every task is non idempotent and there are more replicas then tasks (5 tasks and 10 replicas in the beginning) the idle time is quite high. As the number of tasks increases more replicas can be employed and the idle time decreases.

### 6.2.2 Messaging Overhead

This Section evaluates the message overhead of replication regarding the number of tasks, the number of replicas and the percentage of non idempotent tasks. Quite expectedly the message overhead increases with the number of replicas or the number of tasks in a workflow.

**Configuration:**

| | newtork.size | Tasks | DifferentWorkflows | idempotent |
|---|---|---|---|---|
| figure 6.4 | {1, 5, 10, 15, 20} | 10 | {1, max} | {0, 33, 66, 100} % |
| figure 6.5 | 10 | {5, 10, 15, 20, 25} | {1, max} | {0, 33, 66, 100} % |

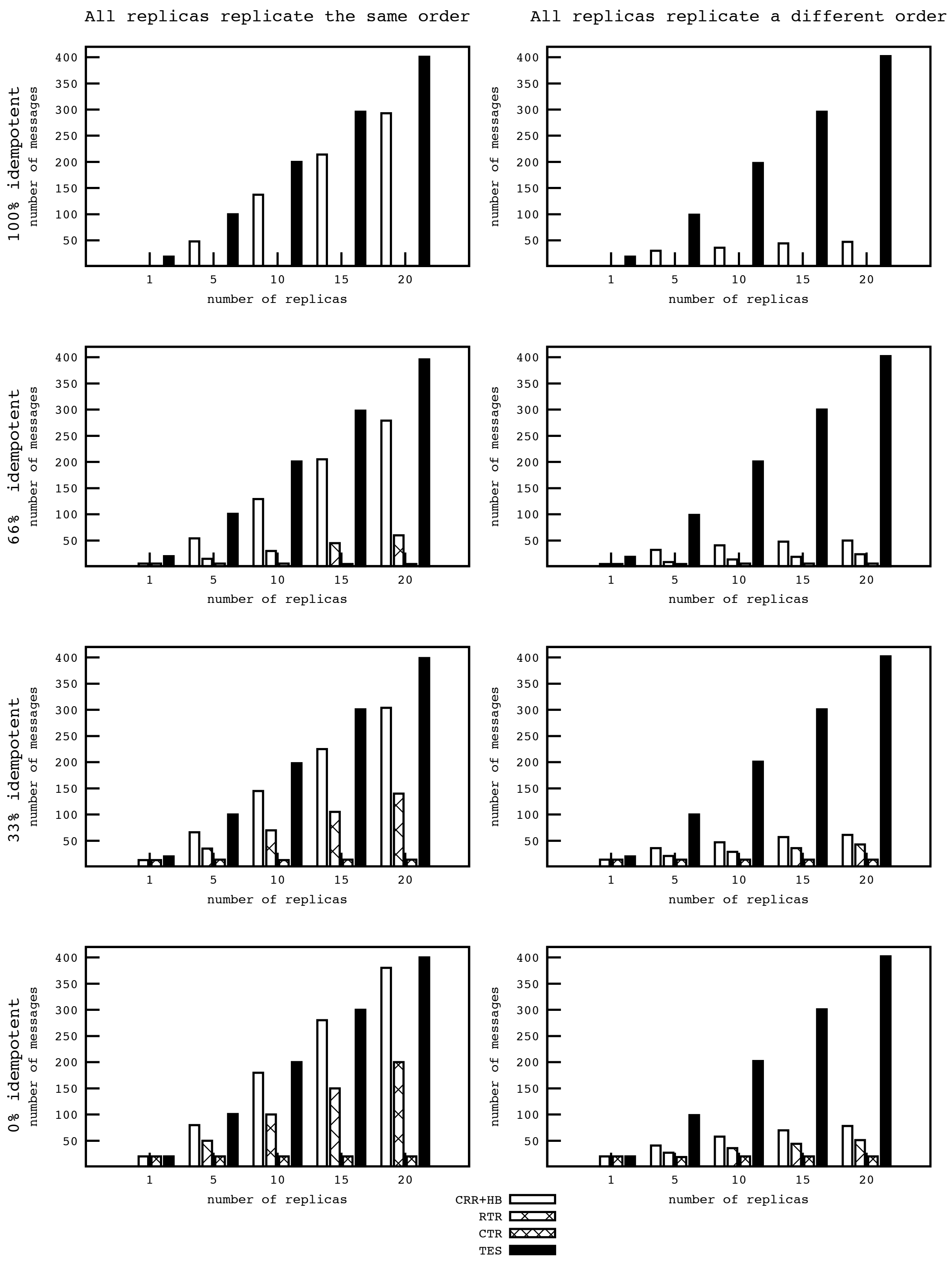


**Figure 6.4:** Messaging overhead regarding number of replicas, task type, and order.

### Description of Figure 6.4

Figure 6.4 depicts how an increasing number of replicas influences the message overhead when replicating a workflow in the same or different order (compare left and right column). The number of tasks for all diagrams is constant (10). From top to bottom the number of non idempotent tasks is increased.

**Task executed successfully messages**
To propagate the execution state of a specific task TES (task executed successfully) messages are send. See section 5.3.2. To keep all replicas up to date these messages need to be send to every replica when a task has been executed successfully and thus grow proportionally to the number of tasks in a workflow and the number of replicas used.

**Stability management messages**
When a replica remains idle no messages are send to the controller and thus after a while (when the timeout has passed) the controller asks the idle replica for a heartbeat. As already discussed in Section 6.2.1 the idle time of the replicas increases when the number of idempotent tasks decreases. When the idle time increases and a lot of replicas remain idle for a longer time the amount of stability management messages increases. This effect can be seen in the left column. As shown in the right column this can be avoided when replicating the workflow in different orders keeping more replicas occupied.

**Workflow coordination messages**
As the percentage of non idempotent tasks increases the amount of coordination messages also grows. When running a workflow in the same order all replicas report ready for the same task at the same time and thus a lot of RTR and CTR messages are created. When replicating the workflow in different orders the ratio of send run requests (RTR messages) and acknowledgement (CTR) messages gets close to one since less replicas have scheduled a specific task at the same "time" in their stack.

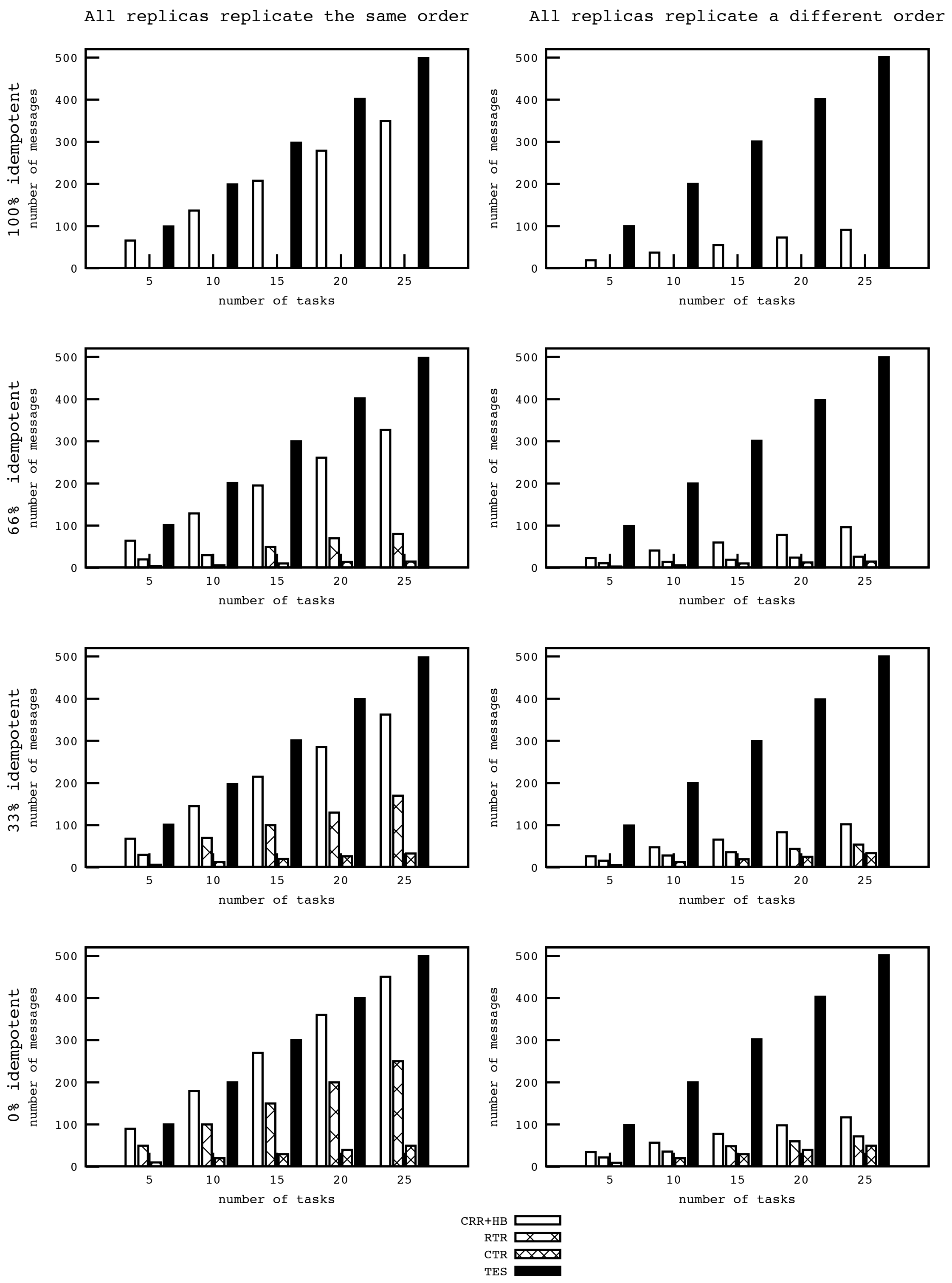


**Figure 6.5:** Messaging overhead regarding number of tasks, task type, and order.

**Description of Figure 6.5**

Figure 6.5 depicts how an increasing number of tasks influences the message overhead when replicating a workflow in the same or different order (compare left and right column). The number of replicas for all diagrams is constant (10). From top to bottom the number of non idempotent tasks is increased.

Figure 6.5 shows the same phenomena like the ones described regarding 6.4. In general this scenario shows that increasing the number of replicas while keeping the number of tasks constant or increasing the number of tasks while keeping the number of replicas constant has the same effect on the message overhead. In both scenarios it grows proportional to the variable parameter.

### 6.2.3 Failure of Replicas

This Section evaluates the impact of failing replicas on replication regarding execution speed and stability management messages. Figure 6.6 evaluates performance for a fixed number of failures and a variated number of replicas. Figure 6.7 evaluates the performance for a fixed number of replicas and a variated number of failures.

To model replica failure the system randomly disables a replica. To detect failed replicas the controller uses the timeout based error detection strategy described in Section 5.3.4. After a timeout the controller requests the replica to send a heartbeat and restarts the replica if he does not receive an answer. This random failure models an equal chance of every replica to fail. If an already failed replica has been "selected" it remains failed. If the failing replicas would only be chosen from the running replicas this would increase the probability of these replicas to fail. The systems tries to distribute the replica failures equally over the execution time of the workflow. This is achieved using execution duration prediction based on the characteristics of the workflow and duration of already executed experiments.

**Configuration:**

| | newtork.size | Tasks | FailingReplicas | idempotent |
|---|---|---|---|---|
| figure 6.6 | {1, 5, 10, 15, 20} | 10 | 30 | {0, 33, 66, 100} % |
| figure 6.7 | 10 | 10 | {0, 2, 5, 10,30, 50} | {0, 33, 66, 100} % |

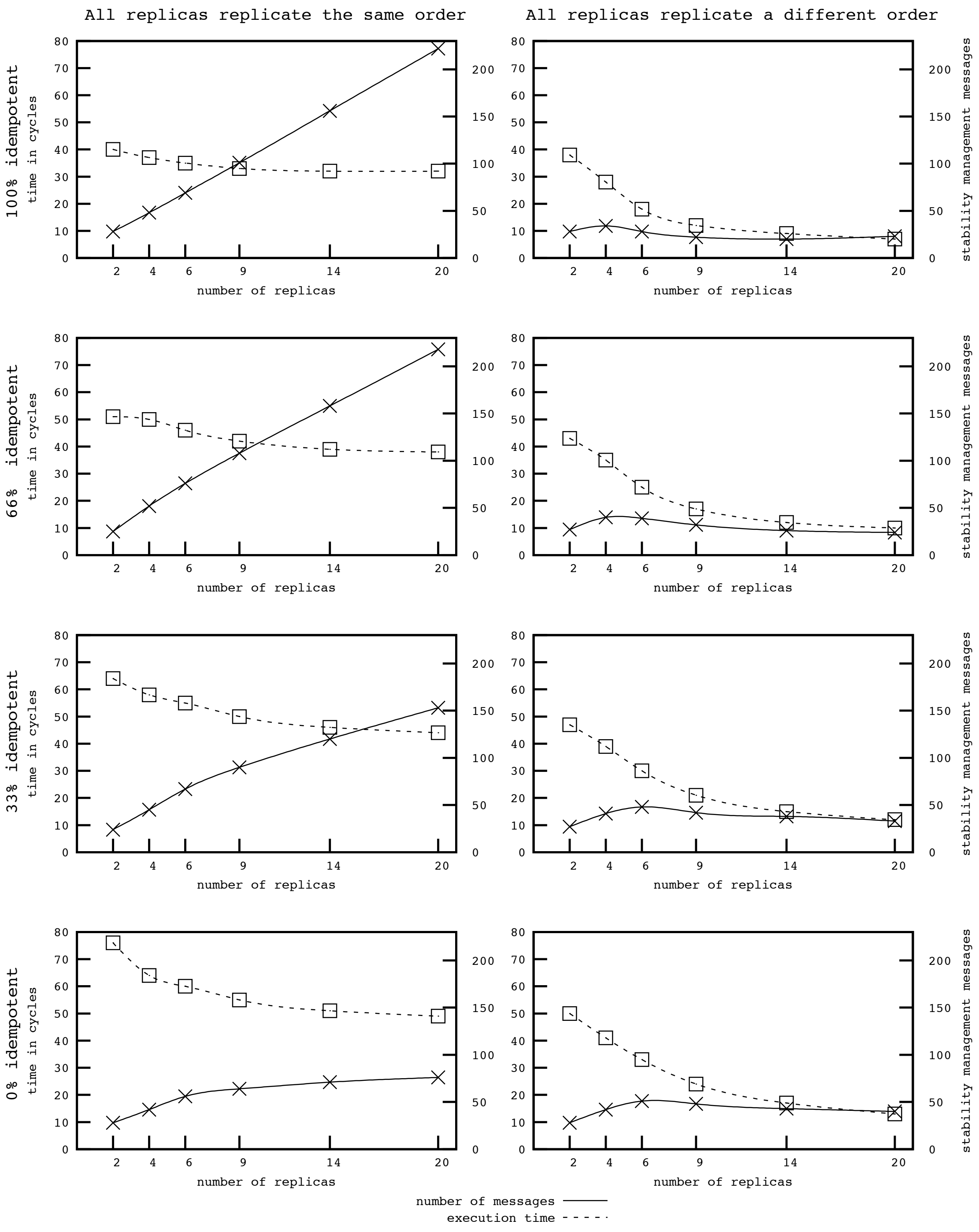


**Figure 6.6:** Speedup and message overhead regarding number of replicas and constant failures.

### Description of Figure 6.6

Figure 6.6 displays how an increasing number of replicas deals with a constant amount of failures equally distributed over the execution time of the workflow. In the left column the replicas all replicate the same order of the workflow while in the right column a different order is used for every replica. The percentage of idempotent tasks decreases from the top to the bottom.

**Messaging overhead**
The figure shows that the messaging overhead in an error prone environment is dependent on the used task orders and the percentage of idempotent tasks. As already pointed out in Section 6.2.2 same order replication and a high percentage of idempotent tasks already cause a great number of stability management messages. This number is further increased by the messages which are caused by actually failing replicas.

When replicating the workflow in different orders the overhead of stability management messages is much lower in general (even without failures) as pointed out in Section 6.2.2. Same order replication has another key disadvantage in the case of a failure which can help to explain the tremendous difference in message overhead. As pointed out before when a workflow is replicated in the same order its progress mostly depends on a few replicas while the others are idle. When one of the replicas actually processing a task fails this causes an additional hold up. During this time the other replicas are also idle since they all wait for the successful execution of one task. Since all the replicas are idle none has a need to communicate with the controller. Because of the timeout based failure detection strategy this causes the controller to check all replicas for being still alive which leads to an additional message overhead.

**Execution speed**
As previously explained replicating a workflow in different orders benefits the overall execution time of a workflow especially when a lot of the tasks are non idempotent. See Section 6.2.1. This behavior is amplified when replica failures occur. Workflows mostly consisting of idempotent tasks are quite robust against failures when replicated since all the tasks have a backup running in parallel on another replica. When however the number of non idempotent tasks rises failures have more and more influence on the execution since non idempotent tasks can only be run on a single replica at once.

When using different workflow orders for replication the time while a failed replica is repaired is used to execute other parts of the workflow. When the replica is finally repaired the overall execution has progressed and not much time was lost. This results in a dramatically shorter execution time for workflows replicated in different orders.

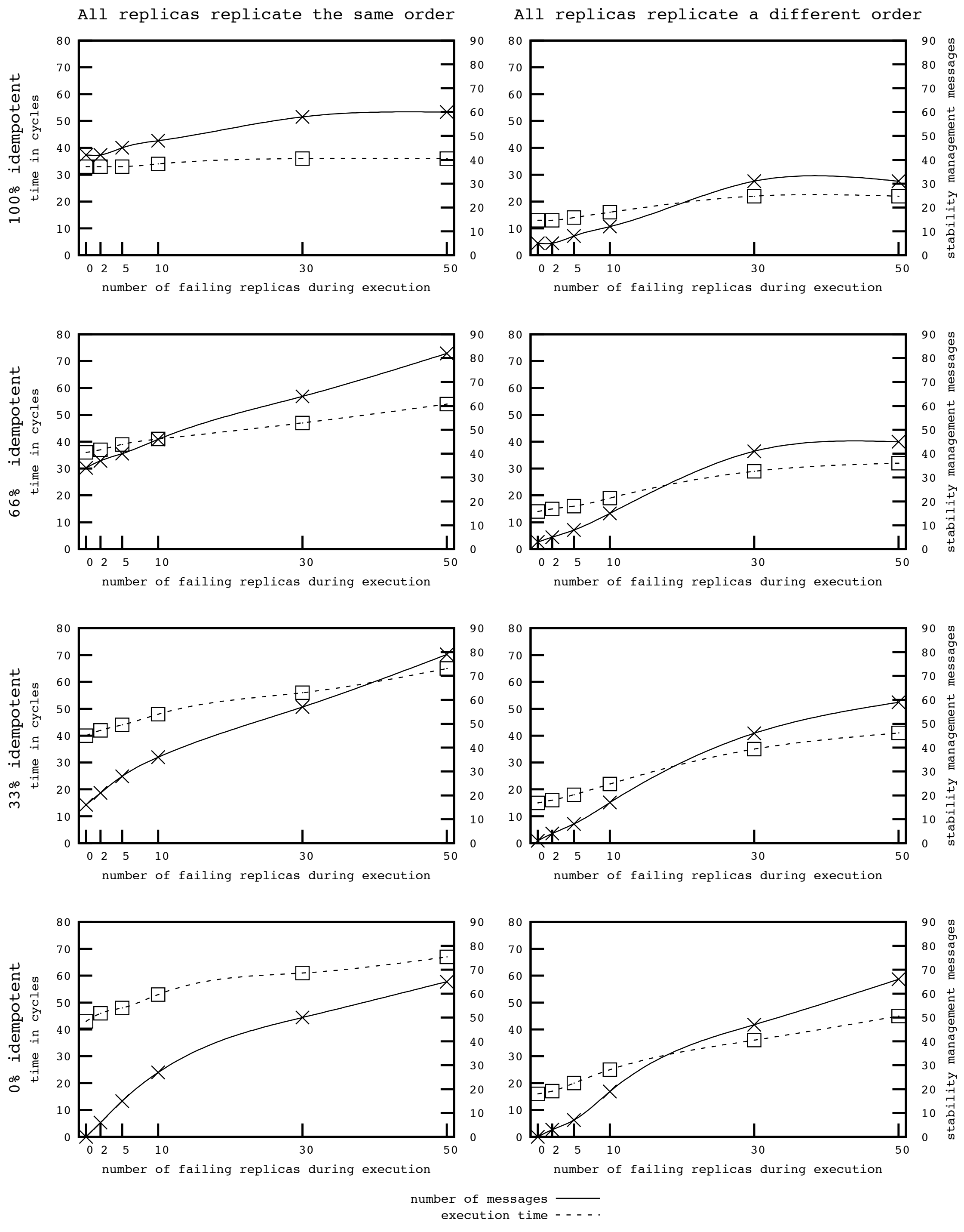


**Figure 6.7:** Speedup, message overhead regarding failing replicas, task type.

### Description of Figure 6.7

Figure 6.6 displays how a constant number of replicas deals with an increasing amount of failures equally distributed over the execution time of the workflow.

The theories presented regarding Figure 6.6 also apply to Figure 6.7. In addition the Figure 6.7 shows that increasing the amount of failures while keeping the number of replicas constant affects same and different order replication in a similar way. Both message overhead and execution time increase however different order replication seems to perform better especially when most of the tasks are idempotent.

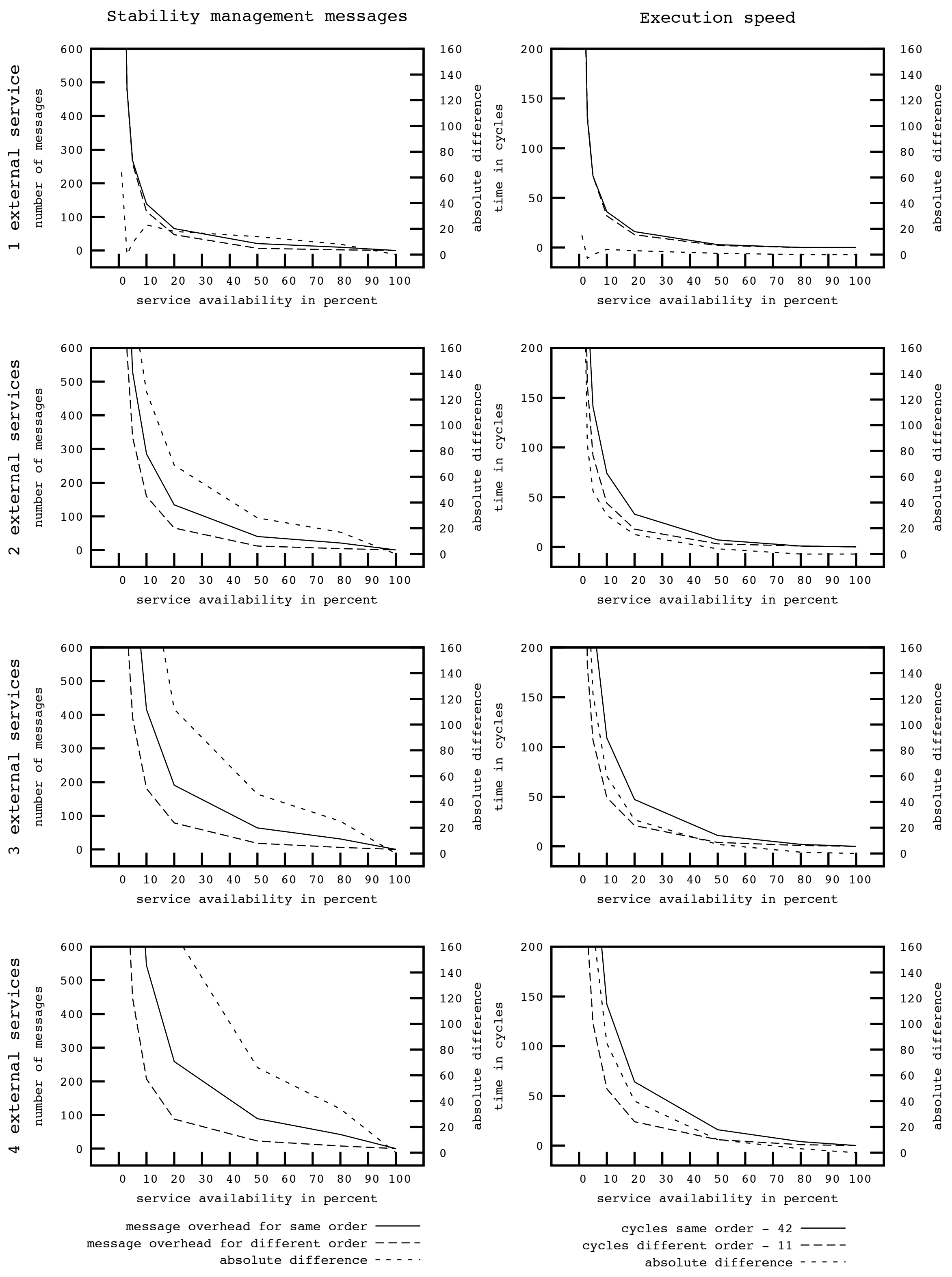


**Figure 6.8:** Execution speed and error handling regarding external service availability.

### 6.2.4 Failure of External Services

This Section evaluates execution speed and message overhead regarding the number of external services, service availability, and different vs. same order replication. Unlike all the previous plots the right column depicts the execution speed and the left column the stability messaging overhead. The plots contain same and different order value plots and their absolute difference for one, two, three, and four external services.

**Configuration:**

| | newtork.size | Tasks | ExternalServices | idempotent | Duration | Executions |
|---|---|---|---|---|---|---|
| figure 6.8 | 10 | 10 | {1, 2, 3, 4} | 0% | 2 | 1 |

In the simulation an external service is modeled as a non idempotent task. A service is initialized with a random availability between min and maxServiceAvailability. The simulation system executes the workflow in cycles. See 6.1. At the beginning of every cycle the availability of every service is determined. The task linked to a service can only be executed by a replica when a service is available in the current cycle.

The simulation shows that a linear increase of tasks depending on external services leads to a proportional increase of stability management messages. One service causes 21 stability messages to be sent at 50 % availability and same order replication. Two services 40, three 64, and four services 89. For different order replication the numbers are similar: 7, 12, 18, 23 also at 50 %. Unsurprisingly one additional service thus causes a fixed amount of additional stability management messages. Again different order replication seems to be beneficial in terms of message overhead.

For execution time a similar effect is present which however is not so easy to detect. To make the effect visible in the plot the execution time at 100 % service availability has been subtracted from the measurements (42 for same and 11 for different order replication). If the duration is compared it is evident that an additional service increases the execution time by a constant factor (which is approximately 16 at 20% availability in this simulation). This can be seen when comparing i. e. the values of same order replication for different numbers of external services at 20 % availability. For one service its 16, for two its 33, for three its 47 and finally 64 for four services. (The expectancy values are 16, 32, 48 and 64).

This shows that the effects of external services do not influence each other and do not make things worse when several external services are combined.

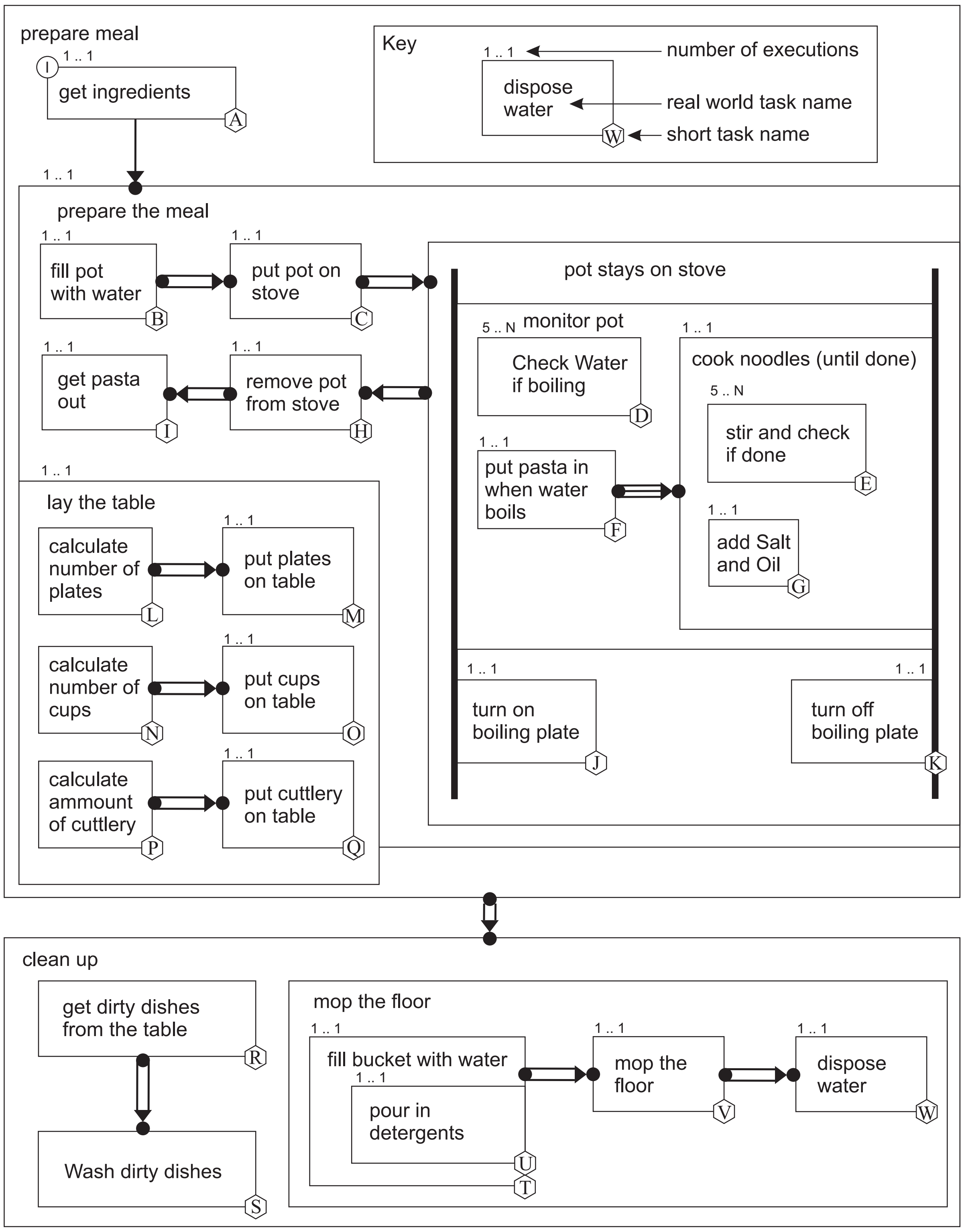


**Figure 6.9:** The kitchen workflow – a real world workflow example.

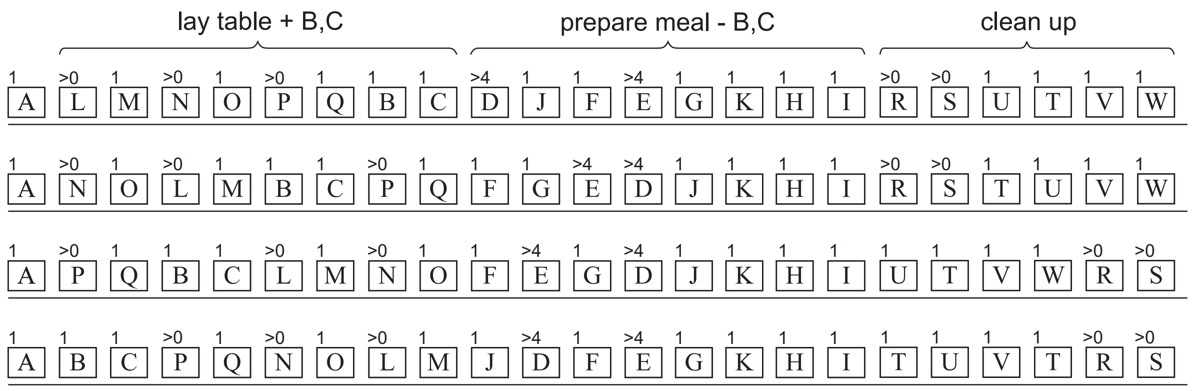


**Figure 6.10:** Four different execution orders of the kitchen workflow. The tasks are labeled with their short name (See fugure 6.9). Every row represents a different order.

### 6.2.5 Real World Example

To test the algorithm on a real world example the workflow shown in Figure 6.9 is used. The depicted workflow is defined in extended Declare (Compare Section 3.4.5) and models the process of preparing a meal and cleaning up the kitchen afterwards.

Before the workflow can be executed workflow orders need to be generated. Four different execution orders have been created manually for the kitchen workflow example. See Figure 6.10. For information about workflow orders compare Section 5.2.

Like in most human flows most of the tasks are non idempotent and can be only executed once. The reason for this is that most human flows describe how things are made or accomplished. Most of the tasks are small steps that add up to something concrete. If one for example has a look at the assembly guide from a famous swedish furniture shop one will recognize that most of the the tasks have to be done exactly once or a fix number of times. Most of the tasks even have to be done in a specific order that canont be changed. For this reason most of these flows have a high percentage of non idempotent tasks. For the presented kitchen workflow this is 70 %. (16 out of 23).

The replication of the kitchen workflow is simulated using the same workflow order for every replica and different orders for every replica (Which are presented in Figure 6.10). Evaluation is done regarding execution time, idle time, number of messages and number of replicas. The proposed execution orders are directly feed into the simulation and thus only the simulation parameter have to be defined in the config file (No workflow coordinator needs to be configured):

**Configuration:**

| | newtork.size | sameOrder | DifferentWorkflows | experiments |
|---|---|---|---|---|
| figure 6.8 | 4 | {true, false} | {1,4} | 500 |

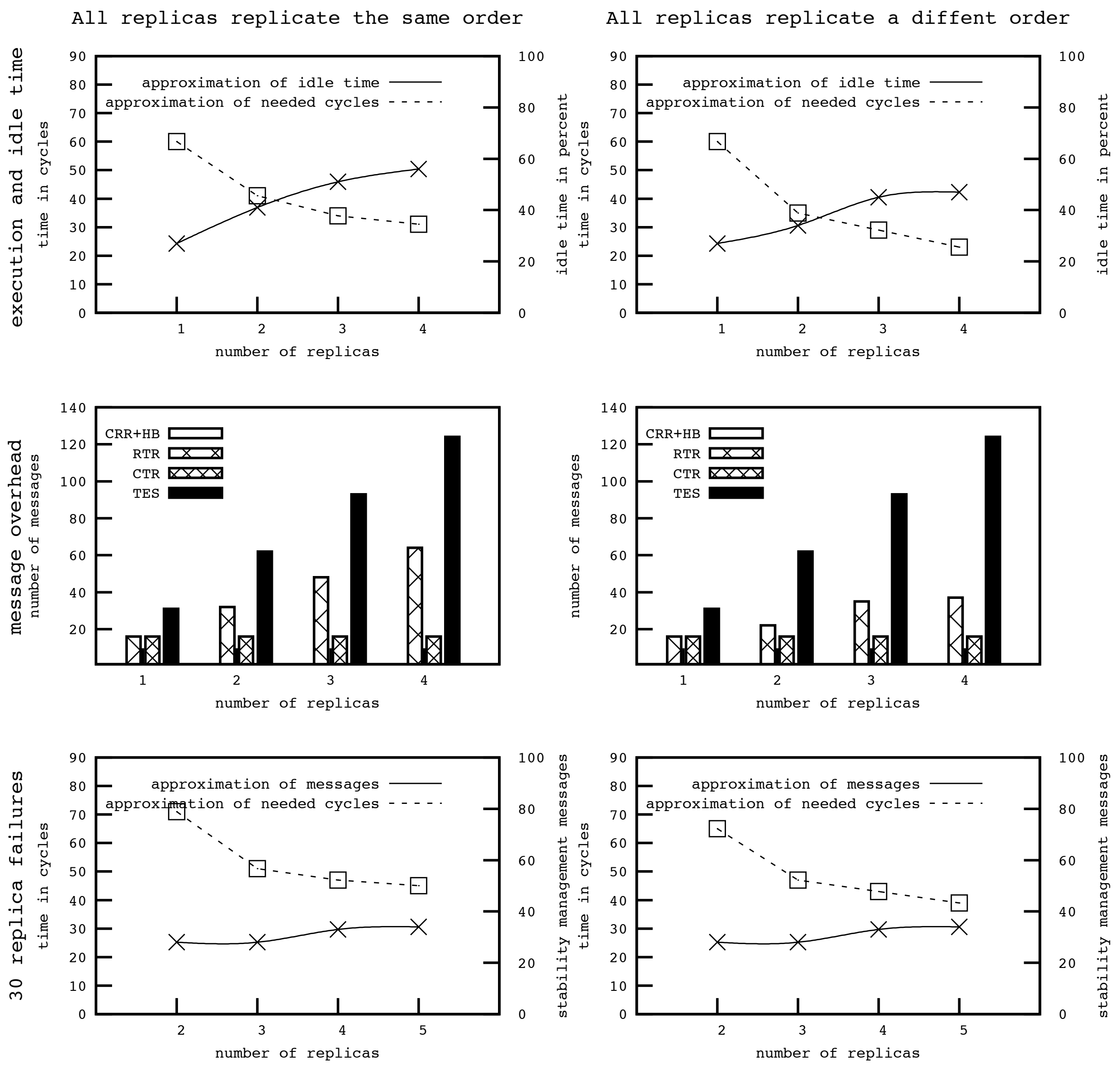


**Figure 6.11:** Kitchen workflow evaluation

Figure 6.11 shows the simulation results of the presented real world workflow (Figure 6.9). Simulations regarding message overhead, execution and idle time have been done without failing replicas. The real world workflow shows a similar behavior like the correspondent synthetic examples. Using the replicas to run tasks in parallel that need to be executed several times proves once again to be beneficial regarding execution time. Running the workflow in different orders increases the execution speed by 30 %. Increasing the number of replicas however has the disadvantage of increasing the message overhead and idle time of the replicas.

Looking at the idle times it is conspicuous that the idle time for one singe replica is 25 %. This is due to the used cycle based engine. When an execution request is sent in one cycle the controller answers in the next cycle. This means that the replica is at least idle for one cycle.

Since most of the tasks take only about two cycles to be executed this has a big impact on the measured idle time.

Comparing the plots regarding speedup and idle time for same and different order replication implies that most of the performance gain comes from parallel execution of tasks that need to be done several times. The performance gain from using different order replication is visible but not breathtaking. This example also shows that finding optimal replication orders is a crucial task. The quality of the generated workflow execution orders has a direct influence on the execution time and the time a replica remains idle.

As already described in Section 6.2.2 replicating a workflow in different order is beneficial regarding message overhead. While the number of TES messages remains the same there are less RTR messages. Replicating a workflow in different orders leaves less replicas to execute one task at the same time and fewer RTR messages are sent. The absence of error handling message overhead shows that the timeout based error detection strategy seems to be a good choice since it can remain silent while no errors occur.

To simulate an unreliable environment random replicas have been failed 30 times. The result is shown in the last two rows. Regarding replica failure the real world workflow performs similar to the synthetic workflows described in Section 6.2.3.

In general the results of this real world workflow replication show that a real workflow behaves very similar to the corresponding synthetic workflow examples. The biggest problem however remain the non idempotent tasks which cannot be run several times in parallel for backup. Unfortunately in the presented real world example this tasks make up 70% of all tasks. This could be addressed by balancing the costs for workflow violation and expected failure as proposed by [SPJ11]. (Compare related work Chapter 2.3.)

Chapter 7

# Conclusion

This thesis shows that workflow replication can be used to increase the robustness of workflows executed in a distributed environment. Different declarative workflow languages have been presented and their modeling capabilities have been analyzed. Task types present in declarative languages have been identified and their impact on replication was evaluated. Methods to coordinate the execution these task types have been developed. To evaluate the proposed coordination methods a replication protocol was implemented and simulated utilizing the PeerSim peer to peer simulator. The evaluation was done using a large number of synthetically generated workflows as well as a real workflow example.

It has been shown that in contrast to imperative languages declarative languages provide more flexibility and freedom for workflow execution. This is possible because a declarative language only defines what needs to be done and not how. Given that freedom it is possible to execute a declarative workflow in many different orders which do not need to be explicitly defined by the workflow.

The presented replication algorithms make use of this additional flexibility. The simulations show that if the workflow is executed by the replicas in different orders it can be executed faster and with less message overhead. Replica failures also have less effect on execution time when the workflow is replicated in different orders.

Among the presented declarative workflow languages Declare was chosen to model the workflows presented in this thesis. Augmented with additional temporal constraints it can be used to precisely model the coherences between the different tasks in a workflow. The identified task types have been divided into two major categories, idempotent and non idempotent. This classification is important for replication because non idempotent tasks can only be executed in a limited way and have to be coordinated. To globally coordinate their execution one of the replicas is selected as a coordinator and governs their execution. To avoid a single point of failure the state of this coordinator is replicated to a backup.

# Bibliography

[AAEA$^+$96] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Gunthor, C. Mohan. Advanced transaction models in workflow contexts. In *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*, pp. 574–581. IEEE, 1996. (Cited on page 8)

[BDF$^+$12] J. Brzeziński, A. Danilecki, J. Flotyński, A. Kobusińska, A. Stroiński. ROsWeL Workflow Language: A Declarative, Resource-oriented Approach. *New Generation Computing*, 30(2):141–164, 2012. (Cited on page 15)

[DAPS09] W. M. van Der Aalst, M. Pesic, H. Schonenberg. Declarative workflows: Balancing between flexibility and support. *Computer Science-Research and Development*, 23(2):99–113, 2009. (Cited on pages 14 and 16)

[DJ07] S. Dustdar, L. Juszczyk. Dynamic replication and synchronization of web services for high availability in mobile ad-hoc networks. *Service Oriented Computing and Applications*, 1(1):19–33, 2007. (Cited on pages 2, 5 and 27)

[DR09] P. Dadam, M. Reichert. The ADEPT project: a decade of research and development for robust and flexible process support. *Computer Science-Research and Development*, 23(2):81–97, 2009. (Cited on page 15)

[FLM$^+$09] D. Fahland, D. Lübke, J. Mendling, H. Reijers, B. Weber, M. Weidlich, S. Zugal. Declarative versus imperative process modeling languages: The issue of understandability. *Enterprise, Business-Process and Information Systems Modeling*, pp. 353–366, 2009. (Cited on pages 13 and 27)

[FMR$^+$09] D. Fahland, J. Mendling, H. Reijers, B. Weber, M. Weidlich, S. Zugal. Declarative versus Imperative Process Modeling: The Issue of Maintainability. In *Proc. ER-BPM'09*, pp. 65–76. 2009. (Cited on page 13)

[GCG05] A. Guabtni, F. Charoy, C. Godart. Spheres of isolation: adaptation of isolation levels to transactional workflow. In *Business Process Management*, pp. 458–463. Springer, 2005. (Cited on page 8)

[GS96] R. Guerraoui, A. Schiper. Consensus service: a modular approach for building agreement protocols in distributed systems. In *Fault Tolerant Computing, 1996., Proceedings of Annual Symposium on*, pp. 168–177. IEEE, 1996. (Cited on page 9)

[HCK02] N. Hayashibara, A. Cherif, T. Katayama. Failure detectors for large-scale distributed systems. In *Reliable Distributed Systems, 2002. Proceedings. 21st IEEE Symposium on*, pp. 404–409. IEEE, 2002. (Cited on page 10)

[HM11] T. Hildebrandt, R. Mukkamala. Declarative event-based workflow as distributed dynamic condition response graphs. *arXiv preprint arXiv:1110.4161*, 2011. (Cited on pages 12 and 16)

[LUW10] F. Leymann, T. Unger, S. Wagner. On Designing a People-oriented Constraint-based Workflow Language. *Services und ihre Komposition*, p. 25, 2010. (Cited on pages 2 and 22)

[MJ09] A. Montresor, M. Jelasity. PeerSim: A Scalable P2P Simulator. In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, pp. 99–100. Seattle, WA, 2009. See http://peersim.sourceforge.net. (Cited on page 53)

[MMTG08] F. Montagut, R. Molva, S. Tecumseh Golega. The pervasive workflow: A decentralized workflow system supporting long-running transactions. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 38(3):319–333, 2008. (Cited on page 8)

[MSB08] Z. Maamar, Q. Z. Sheng, D. Benslimane. Sustaining web services high-availability using communities. In *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*, pp. 834–841. IEEE, 2008. (Cited on pages 2 and 6)

[MSMA90] P. Melliar-Smith, L. Moser, V. Agrawala. Broadcast protocols for distributed systems. *Parallel and Distributed Systems, IEEE Transactions on*, 1(1):17–25, 1990. (Cited on page 9)

[PA06] M. Pesic, W. van der Aalst. A declarative approach for flexible business processes management. In *Business Process Management Workshops*, pp. 169–180. Springer, 2006. (Cited on pages 14 and 17)

[Pes08] M. Pesic. Constraint-based workflow management systems: shifting control to users. 2008. (Cited on page 17)

[PSA07] M. Pesic, H. Schonenberg, W. van der Aalst. Declare: Full support for loosely-structured processes. In *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*, pp. 287–287. IEEE, 2007. (Cited on page 17)

[PSSA07] M. Pesic, M. Schonenberg, N. Sidorova, W. van der Aalst. Constraint-based workflow models: Change made easy. *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, pp. 77–94, 2007. (Cited on page 17)

[PWZ$^+$12] P. Pichler, B. Weber, S. Zugal, J. Pinggera, J. Mendling, H. Reijers. Imperative versus declarative process modeling languages: An empirical investigation. In *Business Process Management Workshops*, pp. 383–394. Springer, 2012. (Cited on pages 12, 13, 14 and 27)

[SJP06] S. Stein, N. R. Jennings, T. R. Payne. Flexible provisioning of service workflows. *FRONTIERS IN ARTIFICIAL INTELLIGENCE AND APPLICATIONS*, 141:295, 2006. (Cited on page 7)

[SPJ07] S. Stein, T. Payne, N. Jennings. An effective strategy for the flexible provisioning of service workflows. *Service-Oriented Computing: Agents, Semantics, and Engineering*, pp. 16–30, 2007. (Cited on page 7)

[SPJ11] S. Stein, T. R. Payne, N. R. Jennings. Robust execution of service workflows using redundancy and advance reservations. *Services Computing, IEEE Transactions on*, 4(2):125–139, 2011. (Cited on pages 2, 6 and 75)

[TLHL09] C. Tang, Q. Li, B. Hua, A. Liu. Developing Reliable Web Services Using Independent Replicas. In *Semantics, Knowledge and Grid, 2009. SKG 2009. Fifth International Conference on*, pp. 330–333. IEEE, 2009. (Cited on page 8)

[VDAP06] W. Van Der Aalst, M. Pesic. DecSerFlow: Towards a truly declarative service flow language. *Web Services and Formal Methods*, pp. 1–23, 2006. (Cited on pages 17, 18, 19, 20, 21 and 22)

[Wag10] S. Wagner. *A Concept of Human-oriented Workflows.* Ph.D. thesis, Diploma thesis, University of Stuttgart, Germany (January 2010), 2010. (Cited on page 22)

[WV97] M. Weske, G. Vossen. *Workflow Languages.* Univ., 1997. (Cited on page 11)

[YMV$^+$10] U. Yildiz, P. Mouallem, M. Vouk, D. Crawl, I. Altintas. Fault-Tolerance in Dataflow-based Scientific Workflow Management. In *Services (SERVICES-1), 2010 6th World Congress on*, pp. 336–343. IEEE, 2010. (Cited on page 10)

All links were last followed on April 3, 2013.

**I like to thank ...**

Prof. Dr. Dr. h. c. Kurt Rothermel for offering me the opportunity to conduct this thesis at the department of distributed systems.

MSc. Muhammad Adnan Tariq for supervising, motivating, and supporting me.

My family and friends that provided me with good advice and always had an open ear when I wanted to speak my mind.

The ERASMUS exchange program for giving me the opportunity do a semester abroad in Ireland ♣

## Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

(Thomas Bach)