Institute of Parallel and Distributed Systems
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3431

Robust Execution of Workflows in a Distributed Environment

David Richard Schäfer

Course of Study: Computer Science

Examiner: Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel

Supervisor: MSc. Muhammad Adnan Tariq

Commenced: December 4, 2012

Completed: June 5, 2013

CR-Classification: H.4.1

Abstract

In many business applications, workflows are used to describe business processes. Employees and machines get instructions from a plan (the workflow) to be guided or controlled. The workflows make it easier to create and manage business processes. Therefore, using workflows is the standard procedure in the business area today.
The distributed execution of workflows plays an important role as almost all nodes are connected to a network today. The importance even increases with the emerging of pervasive environments. Because these systems are prone to failures, it is important to develop reliability methods that ensure that the system works properly even if failures occur. When the robustness of a system in a distributed environment shall be increased, the service that has to be executed is usually replicated and executed by two or more nodes. This means that the exact same behavior is executed by multiple nodes and thereby increases the reliability of the system by being able to cope with node failures.
Changing the order of the activities or using alternative activities to increase the robustness is promising because when each node receives a different workflow that achieves the same goal, the possibility of failures should be further reduced by decoupling the replicas in respect of time and hardware dependencies. We developed a robustness metric that evaluates the robustness of a set of workflow replicas. We also developed methods and algorithms that generate workflows with different orders and alternative tasks within reasonable time. Our evaluations show that our proposed methods work significantly better than deploying a brute-force method to achieve the same behavior.

Contents

1 Introduction **9**

2 Background **13**
- 2.1 Basics of Workflows 13
  - 2.1.1 Task 13
  - 2.1.2 Workflow 13
  - 2.1.3 Process Model 13
  - 2.1.4 Process Instance 14
- 2.2 Graph Theory 14
- 2.3 Model Checking 14
- 2.4 Service Composition and AI Planning 15

3 Survey of Workflow Paradigms **19**
- 3.1 Imperative Workflow Languages 20
  - 3.1.1 Classification of Imperative Workflow Languages 21
  - 3.1.2 YAWL 22
  - 3.1.3 FLOWer 26
  - 3.1.4 ADEPT 28
- 3.2 Declarative Workflow Languages 30
  - 3.2.1 Classification of Declarative Workflow Languages 30
  - 3.2.2 Object-Oriented Declarative Workflows 30
  - 3.2.3 Freeflow 32
  - 3.2.4 Intertask Dependencies 33
  - 3.2.5 Declare 36

4 System Model and Problem Statement **39**
- 4.1 System Model 39
  - 4.1.1 Network Model 39
  - 4.1.2 Workflow Model 39
- 4.2 Problem Statement 40

5 Identification and Generation of Robust Workflow Specializations **43**
- 5.1 Robustness Metric 43
- 5.2 Brute-Force Approach 48
  - 5.2.1 Constructing All Workflow Specializations 48
  - 5.2.2 Calculating the Table 51
  - 5.2.3 Finding the Most Robust Set 51

5.2.4 Run-time Complexity of the Brute-Force Approach 56
5.2.5 Mapping to Existing Problems 57
5.3 Model Checking Supported Generation of Workflow Specializations 58
5.3.1 Simulated Annealing 59
5.3.2 Automaton Expansion Enhanced by Pruning 61

6 Evaluation **71**
6.1 Measurements 71
6.1.1 Finding the Optimal m for RAPH 72
6.1.2 Performance of RAPH and SA with Few Workflow Specializations . . . 74
6.1.3 Performance of RAPH and SA with Many Workflow Specializations . . 78
6.2 Discussion 81

7 Conclusion and Future Work **83**

Bibliography **85**

List of Figures

1.1 Robustness improvement through differently structured replicas 11

3.1 Declarative versus Imperative 19
3.2 A Petri net example 23
3.3 The graphical representation of all elements of YAWL 25
3.4 Schematic FLOWer example 27
3.5 Classification of declarative workflow languages 30
3.6 Freeflow's task life cycle 32
3.7 Example CTL tree 34
3.8 Example tree restricted through a CTL rule 34
3.9 Life cycle of a task for intertask dependencies 35
3.10 Graphical representations of Declare constraints 38

5.1 Pruning example 63
5.2 SPOT generated automaton example 64
5.3 Automaton example during expansion 66

6.1 Finding the optimal m ($m = 1$, $k = 4$) 72
6.2 Finding the optimal m ($m = k - 1$, $k = 4$) 73
6.3 Finding the optimal m ($m = n$, $k = 4$) 73
6.4 LTL-formulas resulting in 2-500 workflow specializations ($k = 3$) 74
6.5 LTL-formulas resulting in 200-500 flow specializations ($k = 3$) 75
6.6 Measurement of Figure 6.5 plotted with logarithmic scale ($k = 3$) 76
6.7 LTL-formulas resulting in 200-500 flow specializations ($k = 3, t_{min} = 4$) 77
6.8 LTL-formulas resulting in 200-500 flow specializations ($k = 3, t_{min} = 5$) 77
6.9 LTL-formulas resulting in 500-2500 flow specializations ($k = 3$) 78
6.10 Measuring the dependency of k_{Flows} on t_{min} ($k = 3$) 79
6.11 LTL-formulas resulting in 500-2500 flow specializations ($k = 3$) 80
6.12 LTL-formulas resulting in 500-2500 flow specializations ($k = 3$) 80
6.13 Calculating the dependency of k_{Flows} on t_{min} 81

List of Tables

4.1 The example workflow $a \rightarrow b \rightarrow c$ mapped to time slots 40
4.2 The example workflow $d\&e \rightarrow f$ mapped to time slots 40

5.1 Workflow specialization examples 44
5.2 Workflow specialization of Table 5.1 after adding sublabels 44
5.3 Comparison of different workflow specialization distance approaches 46
5.4 Explanatory example for *Next Set* 53
5.5 Example tableau of expansion rules 59
5.6 Pruning robustness distance table example 67
5.7 Evaluating the quality of workflows according to RAPH 68

List of Algorithms

5.1 Splitting tasks 48
5.2 Construction of all possible workflow specializations 50
5.3 Calculation of all extended workflow specialization distances 52
5.4 Selecting the most robust set 54
5.5 Helper functions for selecting the most robust set 55

1 Introduction

For most things we do, we build a structured plan in mind or, if it is a more complex plan, on paper. In business applications such a plan is called a workflow. It is composed of tasks that have to be executed during the execution of the workflow. Workflow languages allow to build and manage business processes on the computer. In the business area, workflow languages are widely used to organize processes. They make it easier to control and manage these processes and are often employed to optimize these for important properties, such as time or costs. Today, using workflow languages are the standard procedure for organizing business processes. Because almost every node is connected to a network today and pervasive environments are emerging, the distributed execution of workflows gains an increasing importance. As these systems are prone to failures, methods have to be developed that increase the systems robustness against failures. Nodes can be temporarily unavailable or can fail permanently. Also, disturbances can occur in the environment, such as power blackouts. The system should be able to work properly even if failures occur. The value that quantifies this attribute is called robustness.

If a workflow system has a low robustness, it has a high probability to fail, which means that the process has to be started from the beginning again to achieve the goal of the workflow. This increases costs and delays the completion of the workflow. This is never desired, especially in the business area. Thus, robustness is critical for business processes because of costs, deadlines and just-in-time productions. Our work focuses on increasing the robustness of the distributed execution of workflows. We divide robustness into two subcategories.

On the one hand, there is the robustness on the infrastructural level. A (infrastructural) highly robust system is not influenced by events like hardware failures and power blackouts. To improve the infrastructural robustness of workflow systems in distributed environments, the exact same workflow is replicated and executed by different nodes. If one of the nodes fails because of a (internal) hardware error, another will still finish the workflow. Even if a power blackout occurs, there is the possibility that the workflow was replicated to a node with a backup power supply or a mobile node that has an integrated battery.

On the other hand, there is the robustness on the workflow level. This means that the tasks and the order of these are chosen such that the workflow has a lower probability to fail. Considering the replication approach described above, all replicas will fail if one of them fails because they will execute all tasks in the same order and because of this, almost at the same time. Thus, this approach alone does not improve the robustness on the workflow level.

We extend this approach to improve the robustness on the workflow level as well. This means that this work focuses on increasing the robustness on the workflow level while using the replication approach, which increases the robustness on the infrastructural level, as a basis. In the following the word robustness is used interchangeably with the term robustness on the workflow level.

As robustness is always a trade-off between cost and robustness, the problem is to achieve a reasonable compromise. Considering replication, this means that the same process is done by two nodes but in exchange the robustness of the infrastructure increases.
To increase the robustness on the workflow level, we investigate the promising approach of using differently ordered and alternative tasks in the workflow replicas. Consider a task that has to request some information from a server. The server can be non-responsive for a certain amount of time. If the task is happening almost at the same time in both replicas, meaning the difference between the two events is smaller than the constant λ, it is very likely that both will fail or both will succeed.
Now consider a task that happens at different points in time in different replicas. The task is more likely to succeed the further the task is separated in two replicas.

In Figure 1.1, there are three different workflow replicas w_1, w_2 and w_3, constructed of the tasks a, b, c, d and e. Three of the tasks (a, c and e) request information from a server. To achieve its goal, each workflow replica has to execute a, b, c and d or it has to execute a, b, d and e. Therefore, e is an alternative task to c. Also, the order of the tasks is not constrained. The server S_1 fails temporarily and has time to get back to a responsive state after the first replica did not get a response and failed (Figure 1.1).
There are two more servers S_2 and S_3. Task c requests information from server S_2, which is permanently available. Task e requests information from server S_3, which has failed permanently. Because c is an alternative task to e, only the replicas with task e will fail whereas all replicas that use task c will succeed.
Figure 1.1 shows how alternative and differently ordered tasks can be used to increase the robustness on the workflow level. To be able to use different workflow replicas that achieve the same goal, we have to generate these from the process model, which, in our case, is linear temprol logic (LTL) based. The replicas should increase the robustness of the system as much as possible, which means that they should differ as much as possible. To be able to measure the difference of the workflows, a metric has to be specified.
Checking if a model conforms to a LTL specification and generating workflows that conform to a LTL-formula is strongly related. In model checking theory, checking if a model conforms to LTL-formulas is done by translating the LTL-formulas into an automata [GPVW96, CGP99, Cou99, GH01, GL02, Thi02, ST03, DLP04, RV07, BKRS12]. The automata can be exponential in size compared to the LTL-formula [GL02] and therefore, much of the research focuses on keeping the generated automaton small [GL02, ST03] and making the translation more efficient [Thi02, BKRS12]. This shows that the problem is memory and time consuming. They use different models of automata, intermediate automata that are reduced by algorithms and on-the-fly checking to find violations before the full model is evaluated against the automata. All of the mentioned research checks if a model conforms to the formula whereas we want to generate workflow replicas from the given formula which renders all speed ups that depend on the model useless. This means we have to find a solution to cope with memory and time consumption of the generation of workflows.
Haley [ZD07, Zha09], a web service composition framework, improves the robustness by generating a policy that defines which actions are to be performed in each state. Therefore, it is able to react to any state, which also includes a failed task execution. Because the policy has to store all actions for all states, this approach has to deal with the state space explosion

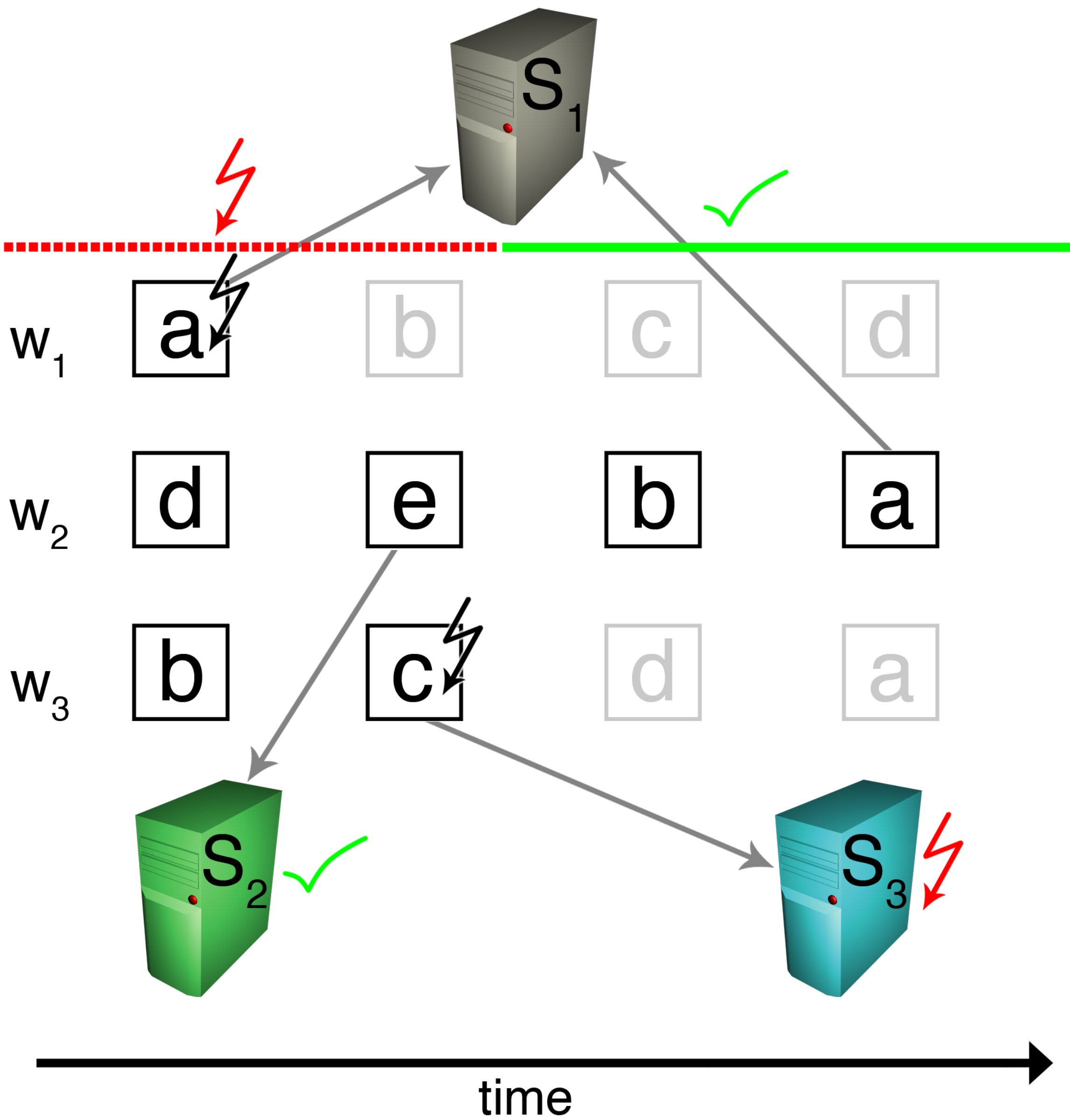

Figure 1.1: Robustness improvement through alternative and differently ordered tasks in the workflow replicas. Three workflow replicas w_1, w_2 and w_3 with the same goal are executed. The server S_1 has a temporary failure and is non-responsive during the time denoted by the red dashed line and responsive during the time denoted by the green continuous line. Server S_2 is permanently available and S_3 has failed permanently.

problem. Also, it does not improve the robustness on the infrastructural level because if the node that is executing the service composition fails, the whole process fails. To improve the robustness on the infrastructural level, replications is needed.
In conclusion, we need to find a robustness metric and a possibility to efficiently generate a highly robust set of workflow replicas. Thus, a system has to be created that automatically constructs workflow replicas from LTL specifications. If the tasks can be ordered arbitrarily or almost arbitrarily and/or many alternative tasks exist, the amount of possible workflow replicas is huge. Because of this, creating the most robust set or creating all possible workflow executions and then finding the most robust workflows out of all the created ones, is a difficult problem that can lead to very time and memory consuming calculations. Therefore, this approach needs techniques that reduce the run-times and memory usage.
In this work, we present methods to generate differently structured workflow replicas and find a reasonably robust set in a time and memory efficient manner. The evaluations show that our methods work significantly better than deploying a brute-force method to generate the workflow replicas.

Document structure

The thesis is structured as follows:

Chapter 2 – Background In this chapter, all necessary background is presented. This includes explanations of the fundamental terms of workflow languages, a short introduction to model checking theory and a discussion of service composition and AI planning techniques that overlap with our work.

Chapter 3 – Survey of Workflow Paradigms Here, we provide a classification for workflow languages and present, for each the classifying attributes, a common representative.

Chapter 4 – System Model and Problem Statement The topic of this chapter is the definition of the system model, which consist of the system architecture and the workflow model, and the problem statement.

Chapter 5 – Identification and Generation of Robust Workflow Specializations In this part of our work, we present the robustness metric and then analyze the brute-force solution of the problem and present a more sophisticated approach that builds on model checking theory. We present two different enhancements to it. The first approach is to combine it with a simulated annealing technique to approximate the most robust set. The second one is to extend the model checking approach by pruning with a heuristic.

Chapter 6 – Evaluation In this chapter, the pruning approach and the simulated annealing approach are evaluated against each other and the model checking approach with a brute-force identification of the most robust set.

Chapter 7 – Conclusion and Future Work Finally, we summarize our work and identify aspects which are left for future work.

2 Background

The chapter provides the background information on which our work is built. It is composed of a brief introduction into the basics of workflows (Section 2.1), graph theory (Section 2.2), model checking theory (Section 2.3) and a discussion of the overlap of existing service composition and AI planning techniques with our work (Section 2.4).

2.1 Basics of Workflows

This section provides an overview to the fundamental terms of workflow languagues.

2.1.1 Task

A task is an atomic piece of work that someone or something is able to execute [AH04]. The words activity and task are used interchangeably in the following.
Example: Shout very loudly: “Dinner is ready”.

2.1.2 Workflow

A workflow describes a piece of work that can be specifically labeled and is or will be composed of one ore more tasks or workflows. This piece of work is also called *process*. It is either a imperative workflow that consists of an fixed order of tasks or it is a declarative workflow that consists of a goal that must be achieved. Workflows are used to plan and manage processes, especially business processes, and to give support to the executors of the workflows. The words workflow and flow are used interchangeably in the following.
Example: Getting everyone at home to come to dinner. The workflow consists of the one task: Shout very loudly: “Dinner is ready”.

2.1.3 Process Model

The process model is the representation and description of a workflow. It is built to plan and manage the workflow.
Example: The workflow of Section 2.1.2 is an example of a workflow model as it is written down and thereby represented and described.

2.1.4 Process Instance

A process instance of a process model is created every time a workflow is actually executed. Example: Every time someone executes the workflow of the process model of Section 2.1.3, a process instance is created.

2.2 Graph Theory

A graph G is defined through *vertices* V and *edges* E.

$$G = (V, E)$$

Whereupon the edges define which vertices are reachable from a vertex. One edge always starts at one vertex and ends at one vertex and thus, connects them, i.e., makes the vertices reachable from each other.
It is further specified that in a directed graph each edge has a direction and therefore, acts as a "one-way road". For example, if there is an edge from vertex A to vertex B but no edge from B to A, it is possible to reach A from B but not the other way round. For a further description of graphs, we refer to Bondy and Murty [BM08].

2.3 Model Checking

As model checking is a huge topic itself, we refer the interested reader to Clarke et al. [CGP99] and Baier and Katoen [BK08], as the following short introduction to model checking is based on these books.
Model checking theory is used to verify models of systems at design time. Such systems can be a hardware design or a piece of software. Finding errors at design time is important, especially when designing hardware, as these errors can not be corrected with an software update. When designing software, there are also cases in which software can not be updated easily or a software bug would have severe consequences, like in software for financial markets or medical devices. Also, costumers will be upset when they get poorly conceived products. Therefore, companies have huge interest in delivering their product as bug and error free as possible. To verify models, model checking consists of three different parts: the *model*, the *specification* and the *verification*.
The model description is an implementation of the system design. For software, the model description is the code of the software. This model description is used to generate the model that is verified. This step mostly works automatically.
The model itself is usually a transition system. This means that all states in which a system can be are the nodes of the transition system. The transitions specify which action(s) will migrate the system from one state to another.

The specification defines what the model is allowed or not allowed to do. This is often expressed via a temporal logic, like linear temporal logic (LTL) or computational tree logic (CTL).
The temporal logic allows to specify constraints which restrict the order of events. This means that the temporal logic defines when which events are not allowed to happen, depending on other events and the sequence of these events. An example constraint is: a is only allowed to happen after bt.
The verification checks if the model is conforming to the specification. In order to do this, every trace of going through the model (by using the transitions) will be explored. Every trace is evaluated against the specification. If the specification is given in LTL, it is usally translated into an automaton that accepts all allowed event traces but only considers those events that are specified in the LTL-formula. Thus, all sequences of events from the model will be used to go through the automaton of the specification. If the automaton does not accept the trace, the specification is violated. If a violation occurs, the trace that leads to the violation, is returned to the user. Thus, the user is able to understand how the error arose and fix the problem.
Consider a house that is in construction. The bare brickwork is done. Now the water pipes pipes and the electricity have to be installed. The painting work has to be done. The furniture has to be assembled and much more things have to be done.
Many craftsmen are needed to do this and somebody has to make a plan who is going to work when. There are several things to consider. For example, the painting should not start before the electricity is finished within the specific room. Also, there should not be to many people in one room at once, which depends on the space of the room. For instance, in a small bathroom the plumber and the electrician can not work at the same time.
All these things can be defined in the specification. Thus, when making the plan for the craftsmen, which is the model description, the system can verify if all specifications were met. This is useful, because it is easy to forget one of the specifications while planning.
While this example is not very flexible because its model is a very specific plan, it shows clearly what the advantages of model checking are. For a more flexible system, it gets even harder to overview everything and therefore, model checking is even more useful.
In this work, we exploit the possibility of translating a LTL-formula into an automaton that accepts all event traces that are not forbidden by the LTL constraints. It will be applied to generate workflow replicas from a LTL specification. Because much research of creating such automatons is focusing on keeping the automata small and speeding up the translation into the automata [Cou99, Thi02, ST03, BKRS12], this proves to be a very time and memory consuming process. This means we need to cope with these problems to use this approach.

2.4 Service Composition and AI Planning

Service composition [SdF03, ZD06, ZD07, Zha09] and AI planning [HTD90, Bly99] aim to automatically generate a process or plan that achieves a given goal. Service composition is mostly known in the field of service oriented architecture (SOA). Services can be reused and composed to execute complex functionality. This simplifies the creation of complex services because when the complex service is analyzed the sub-steps of it may already be specified by a separate service, which can be used. Forming a complex service with other services is called

service composition.
AI Planning aims to intelligently control a machine without any human help after giving the machine the plan. To create autonomous systems, they have to have a plan that they can follow. Within a deterministic environment this can usually be solved by creating a strategy. This strategy tells the system what to do in which situation, i.e., it is an event condition action system. Since the environment is deterministic for every event there is only one action to do for the system.
In non-deterministic environment this is no longer possible because a defined action from a defined state can lead to several different states. Finding the optimal plan for such a system is an NP-hard problem [HTD90, GN90]. The problem is to create the optimal plan that leads from a starting state to the goal of the machine (that the plan is given to). The plan often has to handle competing goals (like costs and the quality of the outcome) and uncertainty (e.g., because of imprecise sensor data).
Much research has been done in the topic and many techniques were presented [HTD90, Bly99]. An interesting technique is the use of Markov decision processes (MDPs) [Bly99]. In the following, we give a short introduction to MDPs of Blythe [Bly99].
MDPs are composed of a set of states and a set of actions. Furthermore, there is a function that maps an action and a state to a probability distribution of states. This means it tells how likely it is that a specific state is reached, when being in a specific state and performing an action. Additionally, a reward function is defined that maps combinations of states and actions to reward values.
Then a policy is created, which decides which action will be performed in which state. The policy is evaluated with the probability distribution of reaching new states and the reward function. Then the policy will be improved iteratively until it does not improve anymore. This means it does not rely on a predefined order of actions and is able to react to all possible states (e.g. a failed action) by using its policy.
The AI planning partly overlaps with our work because we also try to find the optimal plan (set of workflows). It shall make our system most robust. This is necessary because our system is also running in a non-deterministic environment. The non-determinism arises because of perturbations that can lead to the failure of the workflow. We want to create the most robust set of workflow replicas, which are sequences of tasks. Seeing the tasks as actions, the composition of the workflow is equal to AI planning because both want to achieve a given goal. Besides that, model checking theory already provides the functionality to create workflow replicas from LTL-formulas and that is why we use the model checking techniques instead of AI planning and service compostion approaches.
Also, few works have focused on ensuring robustness. Haley [ZD07, Zha09], a web service composition framework, improve the robustness by using first order semi-Markov decision processes (FO-SMDPs) to build a policy that decides based on the current state, which service will be executed next. This way, it is able to react to system state changes at run-time and has no prespecified fixed order, which obviously increases the robustness (like MDPs (see above)). FO-SMDPs are a generalization of MDPs that provides support of hierarchy to make the system scaleable [ZD07]. We will use a different approach to increase the robustness for several reasons.
Firstly, we assume that hardware failures can occur in our system, which makes replication necessary. Secondly, the tasks may depend on external resources, whose behavior would have

to be included in the policy. Thus, Haley has to deal with the state explosion problem. This leads to our last argument. We do not want to care about each and every possible state, but create a robust system in an efficient way.
To do so, we exploit declarative workflow descriptions to generate differently ordered workflow replicas that will be executed by different nodes of a distributed environments. To the best of our knowledge, this work presents a new approach.

3 Survey of Workflow Paradigms

This Chapter is a survey of existing workflow languages. They are shortly presented and compared to each other. The workflow languagues choosen for this chapter are a brief survey of the large assortment and represent the most common approaches.
In order to provide better insight into the main features of the different workflow languages, they are classified by their properties. One of the most common approaches is the differentiation between imperative and declarative workflow languages.
While in imperative workflow languages every allowed order in which the tasks can be performed has to be defined explicitly in the process model by the process designer, in declarative languages the process model defines the goal that must be achieved by the workflow. Constraint-based workflow languages do this by the definition of tasks and constraints. Thus, in an process instance of a constraint-based, declarative workflow language, every possible way of execution is allowed that does not violate any constraints permanently. This means that this approach works the other way round because every execution order of the tasks is allowed unless it is explicitly forbidden, i.e., a constraint prohibits this execution order. [SMRM07, SMR+08b, FMR+09b, FMR+09a, Pes08]
These definitions lead to some specific advantages and disadvantages for each paradigm (Figure 3.1).

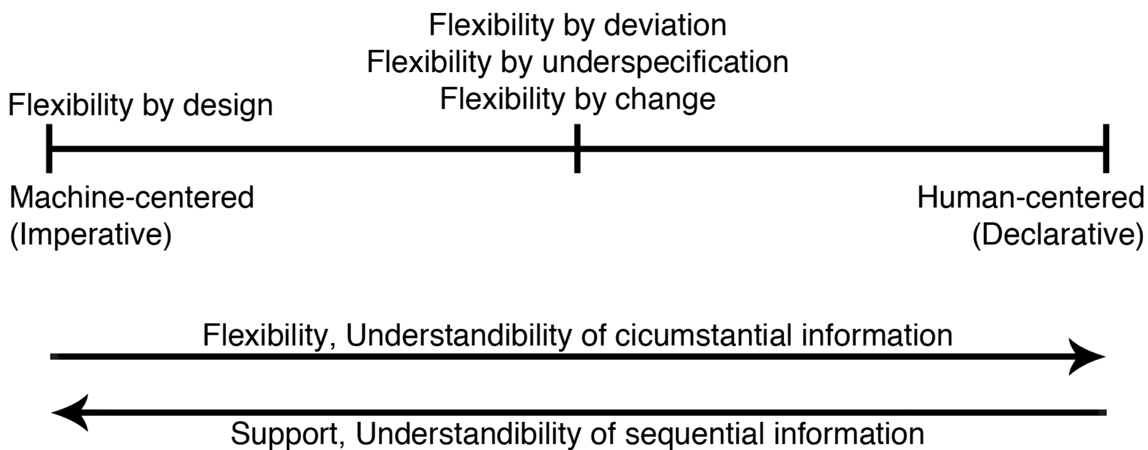

Figure 3.1: Declarative versus Imperative. Flexibility by design, deviation, underspecification and change (see Section 3.1.1) are only meant as extensions of imperative approach in this figure.[SMRM07, SMR+08a, SMR+08b, APS09, HHJ+99, FMR+09a, FMR+09b]

The imperative paradigm features a clear structure of the workflow and therefore, gives much support to a person who has to carry out the tasks [APS09] (Figure 3.1). On the downside, this means that there is no freedom left, i.e., the work has to be carried out exactly as it is specified in the workflow model. Likewise, this is an advantage in case of a process that has a strict order and in which every deviation would be wrong. Also, thinking about computers and machines in gerneral, which do need a well-defined and structured plan to work, the imperative paradigm seems perfectly adequate (Figure 3.1). In contrast to that, a person might be frustrated having to follow a strictly ordered plan instead of doing the work the way he prefers to.
The declarative paradigm gives a worker the complete freedom to do the work in every possible way that leads to the desired result. Although this can be regarded as a positive feature, it leads to some problems. A worker that needs support doing his work, does not get any except the used workflow management system is extended to make suggestions on what a worker should do next [APS09] (Figure 3.1). If a constraint-based workflow language is employed, this can be achieved by checking the task history and evaluating the remaining tasks against the constraints [APS09]. Thus, using declarative workflow languages means spending additional processing power to give support to the user.
Similar arguments apply in context of workflows for machines and computers. Given the fact that they need specific instructions, it would be necessary to produce an imperative behavior to control a machine with a declarative workflow (Figure 3.1). However, this can also be a benefit. If the workflow is highly context dependent, it might be easier to build a declarative workflow model and implement an extension to the workflow management system that decides actively at run-time which task should be executed next.
Concerning understandability, Fahland et al. argue in [FMR+09b] that it is easier to extract sequential information from an imperative workflow model than from a declarative workflow model (Figure 3.1). However, they argue that circumstantial information is simpler to obtain the other way round (Figure 3.1). This is because in case of an imperative workflow model, it is not always that easy to see the "big picture", i.e., to imagine what all the single tasks will cause as a whole. In contrary to that, in case of a declarative workflow, it is not always easy to see in which order the tasks are allowed to happen.

3.1 Imperative Workflow Languages

Most of the present workflow systems that are based on the imperative paradigm, are extended to grant more flexibility. There are different approaches that allow more flexibility, which will be presented in Section 3.1.1. In the remainder of this section three imperative workflow languages will be presented: YAWL [AH05, AHEA06, YAW] (Section 3.1.2), FLOWer [AB01, ASW03, AWG05] (Section 3.1.3) and ADEPT [RD98, RRD03, RRD04b, DR09, ADEa, ADEb] (Section 3.1.4).

3.1.1 Classification of Imperative Workflow Languages

Heinl et al. [HHJ+99] classified workflow languages through different types of flexibility. This approach was adapted by Schonenberg et al. [SMRM07, SMR+08b, SMR+08a]. We consider this approach a good way of classifying imperative workflow languages. Therefore, the four classes into which flexibility is divided, are presented in the following as Schonenberg defined them [SMRM07, SMR+08b, SMR+08a].

Flexibility by Design

Flexibility by design is achieved through the possibility of designing a flexible workflow with the language by supporting workflow patterns [Wor, AHKB03] like parallelism and choice. Thus, the support for these patterns enable the designer to built a workflow that gives more freedom to the user of the workflow, e.g., by including a choice.
Some of the workflow patterns are supported by all workflow languages. The relevant value is how much of these patterns are supported and how important the supported patterns are. Therefore, we consider flexibility by design a 'soft' classification feature.

Flexibility by Deviation

Flexibility by deviation is given when it is possible to assign different roles to different users or groups of users. A workflow language that is not supporting flexibility by deviation supports the execution role only. This means, a user that is assigned to a task, has no option but to execute the task. Flexibility by deviation is extending this behavior by additional roles, such as the skip role and the redo role. Thus, it is possible to give a supervisor the permission to skip a task whereas the average worker can only execute or redo a task.

Flexibility by Underspecification

Flexibility by underspecification is gained by adding a 'container' task. It has an associated pool of tasks, from which one task has to be chosen to execute the 'container' task. The pool of tasks has to be specified in advance by the workflow designer whereas the binding of the task to the 'container' task can either happen at the instantiation of the process model or at run-time of the process instance depending on what the workflow language is supporting and what the workflow designer specifies.

Flexibility by Change

Flexibility by change is supported if it is possible to alter the process model or instance at run-time. A change of the process instance is often refered to as an *ad-hoc change* and is especially useful if unforeseen events occur at run-time. An *evolutionary change* usually means the adaption of the process model at run-time because when the execution of the workflow is

exposing shortcomings of the current process model, an enhanced version is needed and can be instantly deployed with this technique. However, it has to be considered in which way the workflow language is supporting the evolutionary change. It may instantly apply the change to all current process instances and leave the conflict resolution to the user if any conflicts arise because of the change or it may apply the new version of the process model exclusively to new process instances.

Conclusion

Although we consider this classification of workflow languages a good approach for imperative workflow languages, we do not use it to categorize declarative workflow languages. This is because of the nature of declarative workflow languages. They do not define an ordered, step-by-step plan. Thus, they support flexibility by design naturally. The same holds for flexibility by deviation. There is nothing to deviate from. Therefore, Schonenberg et al. alter the meaning of this category for declarative workflow languages and argue that flexibility by deviation is given if it is possible to specify optional constraints, so that it is possible to violate them and thereby deviate. We consider this alteration a fundamental change to the meaning causing that flexibility by deviations of imperative and declarative workflow languages are not comparable. Furthermore, optional constraints are not such a crucial property that declarative workflow languages should be classified by it.
Flexibility by underspecification is a feature that can be argued about in declarative workflow language. If it improves the support or simplifies the designing process, might depend on the specific realization of the language. However, this paradigm is fully applicable.
Flexibility by change is a valuable feature for imperative workflow language and likewise it is for declarative. However, in declarative workflow languages this paradigm has a slightly different meaning, e.g., there is no need to alter the sequence of tasks. Also, it is generally necessary to check at run-time if the current execution can possibly lead to the declaratively specified goal and therefore, it is relatively easy to check if a change can be applied to a process instance because the tool is already implemented.
All these reasons made us classifying the declarative workflow languages differently (see Section 3.2.1).

3.1.2 YAWL

"A pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts." [RZ96]
The Workflow Pattern Initiative [Wor] examines workflow patterns since 1999. The definition and investigation of such patterns for workflow languages resulted in the publication of [AHKB03]. With respect to the conclusions of that work van der Aalst and Hofstede presented a new workflow language in 2005: YAWL [AH05, YAW].
YAWL is built on high-level Petri nets [AH05]. Therefore, Petri nets are shortly presented in the following.

Petri nets

Our description of Petri nets leans on the work of van der Aalst and van Hee [AH04].

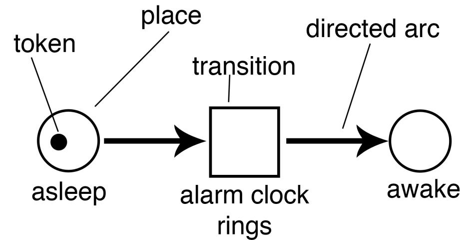

Figure 3.2: A simple example of a Petri net.

In Figure 3.2, a very simple example of a Peti net is shown. The first component of a Petri net is the *place*. It is a representation for things such as an actual place or a state. In Figure 3.2, there are two places that represent the states of a person in the morning; Before the alarm clock rings the person is asleep and afterwards he is awake.
The second component is the *transition*. E.g., it represents an event or an action. To get from one place to another, it is mandatory to go through a transition. In order to show which transitions are directly reachable from a place, *directed arcs* point from the place to the transitions. The directed arcs also indicate the places that will be reached from a transition by pointing from this transition to these places. In Figure 3.2, the event of the ringing alarm clock is the transition that has to happen to reach the place 'awake' from the place 'asleep'.
All places from which an arc is pointing to one transition are called *input places* of this transition. All places to which an arc is pointing from one transition are called *output places* of this transition.
The last component, on which Petri nets are built, is the *token*. For instance, it represents a product or a person. Tokens can only be inside of places and one place is able to hold an arbitrary number of tokens. In Figure 3.2, the token represents the person that is initally asleep.
A transition is *enabled* and thereby is able to *fire* if and only if every input place contains at least one token. If the transition is fired, one token from every input place is removed and one token is added to every output place. Thus, in Figure 3.2, when the transition is fired the token at the place 'asleep' is removed and a token at the place 'awake' is added, representing the person's waking up process.

High-level Petri Nets

The description of high-level Petri nets also leans on the work of van der Aalst and van Hee [AH04]. They argue that the **color, time and hierarchical extension** are necessary to overcome some limitations of classical Petri-nets [AH04].
The **color extension** enables tokens to be identified by assigning a tuple of values. The

transformation can access this information and can act differently depending on the values. This also means that a transition does not have to add one token in each output place when it is fired. Instead an arbitrary number of tokens can be added to each output place and the values of the tokens are also determined by the transition. Furthermore, transitions are extended to support preconditions that have to be met to enable a transition.
The **time extension** modifies the behavior of tokens. Tokens are available to transactions only if their timestamp value is not bigger than the current time. Moreover, a token that is added by a transaction can set this timestamp and thereby set delays. The extension dictates FIFO order for tokens and transitions.
The **hierarchical extension** introduces the possibility to represent a part of the Petri net as a single block that is called a process. The process contains the part of the Petri net that it represents, but itself can be partly or fully constructed of subprocesses.

Extended Workflow Nets

Van der Aalst argues in [Aal96] and together with Hofstede in [AH05] that high-level Petri nets have a high suitability to model workflows, but also mention three limitations [AH05]. Firstly, it is not easily possible to create multiple instances of a part of a high-level Petri-net if the number of instances is not known in advance. Secondly, advanced synchronization patterns are sometimes needed to synchronize parallel branches, which is hard to achieve in high-level Petri-nets. Finally, it is necessary to be able to remove tokens from the net if the specific procedure is canceled. This means removing a token from wherever it currently is in the net. Modeling a high-level Petri-net that is able to do that can result in a very confusing graphical representation [AH05]. For a detailed descriptions of these limitations we refer the interested reader to [AH05]. In order to overcome these limitations van der Aalst and Hofstede introduce *extended workflow nets (EWF-nets)* [AH05].
An EWF-net consists of set of Conditions C that includes the input conditions i as well as the output condition o, the set of tasks T, the flow relation F, *split* and *join* are functions that defines the split and join behavior of each task, *rem* is a function that specifies how tokens can be removed of parts of an EWF-net and *nofi* is a function that defines the instantiation behavior of each task by specifying values such as the minimum number and the maximum number of instances.
The term ‘condition’ is replacing the term ‘place’ and ‘task’ substitutes ‘transition’.
The specification of *composite tasks* [AH05] provides the functionality for the hierarchical extension by acting as (sub)processes. They additionally enable reuse of processes.
YAWL offers a graphical representation with which the control-flow can be designed (Figure 3.3).

Worklet Extension

The worklet extension enables YAWL to specify tasks that can be carried out in different ways, i.e., each way is specified in a worklet. One worklet has to be chosen to carry out the task at run-time. It is also possible to create a new worklet even during run-time

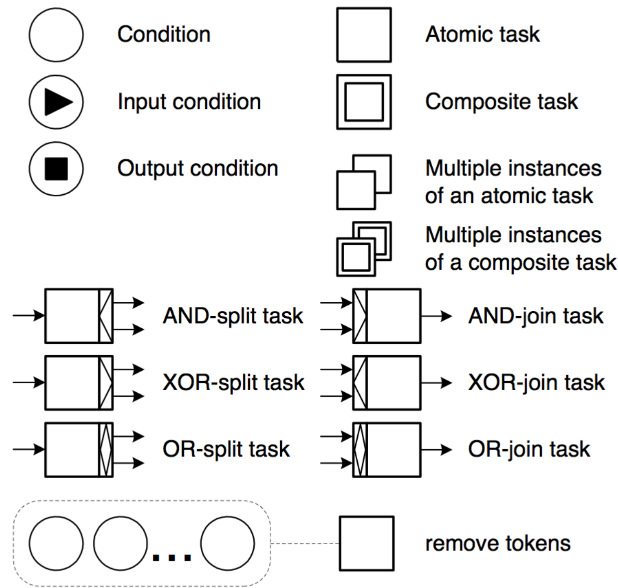

Figure 3.3: The graphical representation of all elements of which a YAWL control-flow can be constructed. [AH05]

[AHEA06]. Thus, the worklet extension is congruent with flexibility by underspecification [SMRM07, SMR+08b, SMR+08a].

Conclusion

YAWL is built on a special adaption and enhancement of high-level Petri nets that have a high suitability for being used as a workflow language [Aal96, AH05]. Through the support of many workflow patterns [AHKB03, Wor, AH05] flexibility by design is deeply supported.
Flexibility by change is partly achieved because it is allowed to modify the process model, but no process instances at run-time [SMR+08a].
The worklet extension [AHEA06] adds support to flexibility by underspecification to YAWL.
Flexibility by deviation is not supported by YAWL [SMR+08a]. Furthermore, it is also possible to add support for declarative workflows through the Declare Service [ACRH09], but because this means including a completely different workflow language, we do not consider it being a feature of YAWL.
In the context of understandability YAWL has the advantage of being based on Petri nets that are graphically representable while having a mathematical respectively semantical background.
YAWL is an example for a workflow language with the main focus on flexibility by design and underspecification [SMR+08a].

3.1.3 FLOWer

Paul Athena's FLOWer is a workflow language that is built on the case-handling paradigm [AB01, ASW03, AWG05]. The paradigm is keeping the focus on the product of the workflow. It does this through the definition of *activities*, *roles*, *plans*, *subplans*, *data objects* and *forms*. Although tasks are always called activities in FLOWer, we will use the terms interchangeably. Whereas the definition of tasks is already given (see Section 2.1.1), all the other terms need to be specified.
A role is defining which permissions a person has on a specific task. There are several roles one can have, e.g., the *skip role*, the *redo role* or the *execution role*. This means that a person who has the skip role is allowed to skip the task (but does not have to skip it).
A plan can consist of tasks, roles, data objects and forms. It is the process model of FLOWer. A subplan is a plan within a plan. When a process designer finishes to model a plan, it can be used as a single component of a new plan. Thus, it can be embedded into the new plan as a singe block like any task. Still, when this block is executed at run-time, the whole plan of the block has to be executed.
A subplan can be static or dynamic whereupon the latter one has the ability to be instantiated multiple times at run-time. Therefore, it needs the following attributes: *Expansion name*, *Minimum instances*, and *Max expansions* [AWG05]. They act as the name of the instance and the lower and upper limit of the number of instances.
A data object can be used to set pre- and postconditions on a task, meaning that a task can not be started or finished unless the particular condition holds. A condition demands a special value from a data object. Thus, these data objects are called *mandatory* for these tasks. In addition, data objects can be *restricted* to tasks and therefore, they can only be accessed within these tasks [AWG05]. Data objects that are not linked to any task are called *free* and can be accessed all the time while the process instance is running.
Forms are documents that present data objects [AWG05]. Forms have to be linked to a task from which they shall be accessible.

The control flow is structured through the succession relation. Tasks support pre- and postconditions. One main characteristic of FLOWer is that if a condition of a task does not hold, instead of blocking the flow, the task is bypassed [AWG05]. Also, every task of the workflow is enhanced to be an AND-join and an AND-split, enabling parallelism and, together with conditions, choice [AWG05].
The task life cycle needs to be adjusted to be able to support the re-doing and bypassing of tasks. Therefore, there are six states in which a task can be: *initial*, *ready*, *running*, *completed*, *skipped* and *passed* [AWG05]. The transitions between these states are straightforward.
Not only the tasks need a life cycle, but also the data objects. There are three states in which a data object can be: *undefined*, *defined* and *unconfirmed* [AWG05]. The 'unconfirmed' state exists to support the re-doing of tasks. When the task that sets the data object, is finished again, the data object switches back to the state 'defined' [AWG05].

In order to give an better insight, an schematic representation of an example workflow in FLOWer is presented in Figure 3.4 . The workflow in Figure 3.4 defines how a bread should be baked in a bakery. However, it is strongly simplified for the sake of shortness. The example

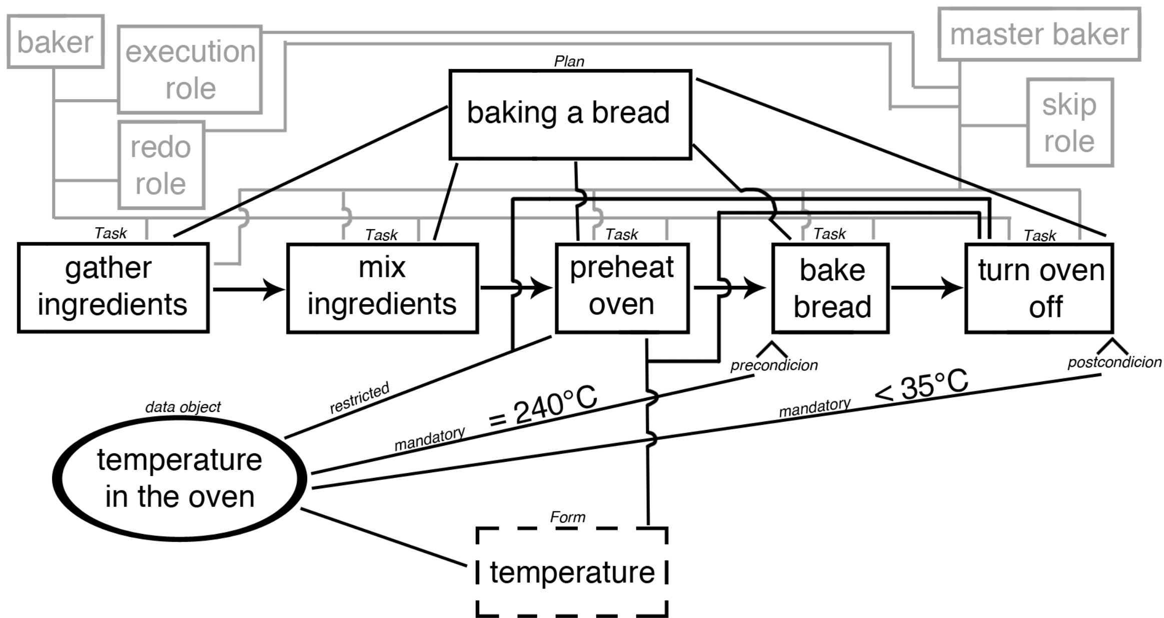

Figure 3.4: A schematic example of a FLOWer workflow, inspired by [AWG05].

includes five tasks that are ordered sequentially with the succession relation. The first task is to gather the ingredients. Afterwards, they have to be mixed. Then, the oven should be preheated because the dough should not be put into the oven before it reaches the temperature of 240°C degrees. Therefore, a data object for the temperature is needed. The temperature can be set by the task 'preheat oven' through a form. It is also restricted to this task and the last task, because one should not change the temperature during the baking process. For simplicity, the temperature is assumed to change instantly in the oven when changed through the form.
Since 240°C degrees are needed before the dough can be put into the oven, a precondition is specified on the 'bake bread' task. After the baking process has finished, the oven should be turned off, which is achieved through the same form as before. After that, the oven should have a temperature below 35°C depending on the room temperature that is assumed to be below 35°C degrees. Thus, a postcondition on the last task checks the temperature in the oven.
The example also includes bakers and a master baker of the bakery. The bakers can take the role of executing or re-doing a task, whereas the master baker is additionally allowed to skip a task. All the role related objects are colored gray.

Conclusion

FLOWer is a workflow language that does not only focus on the control flow view but also on the data view. Through tasks with pre- and postconditions that depend on the presence and the values of data objects and the additional restriction of data objects to specific tasks, FLOWer explicitly enhances the language to support data-driven workflows. A disadvantage is that it is possible to create tasks that will always be bypassed by not putting necessary data objects into the corresponding form, which means that the tasks are not allowed to access the data object (unless it is a free data object) [AWG05].
Although FLOWer is based on the imperative paradigm, it is extended by the assignment of different roles to tasks. The skip, redo and execute roles define who is allowed to carry out the particular action. By supporting the skipping and re-doing of tasks, FLOWer gains flexibility by deviation [SMR+08a]. However, the re-doing of one task means that all subsequent tasks have to be performed (or at least skipped) again. FLOWer also supports flexibility by design through the support of a data-driven workflows and support of workflow patterns [AHKB03, Wor] like choice and parallelism [SMR+08a]. Flexibility by underspecification and flexibility by change are not supported [SMR+08a]. Thus, FLOWer is an example for a workflow language with the main focus on flexibility by deviation [SMR+08a].

3.1.4 ADEPT

ADEPT [RD98, RRD03, RRD04b, DR09, ADEa, ADEb] is a workflow language that addresses the problem that it is not possible to change the process instance in strict imperative workflow languages.
Directed graphs are the basis of ADEPT (see Section 2.2). To be able to model more sophisticated workflows, ADEPT extends the definition of directed graphs.

ADEPT Graphs

ADEPT Graphs are defined as follows [RD98]:

$$G = (N, E, S, D, DF)$$

Whereupon N replaces V and defines tasks instead of vertices. S is the set of *services*. A service describes a procedure that is either started when the task execution finishes or when a task, it is connected to, shall start to execute whereupon the procedure of the service has to be finished before the task execution starts. D are the data objects and DF are the links that define which tasks have permission to access a data object.
To support synchronization of parallel branches Reichert and Dadam [RD98] define a special type of edge: The *synchronization edge* that points from one task of a branch to one task of a parallel branch and leads to the synchronization of the connected tasks. It solves access conflicts to data objects.

The tasks of the graph can adopt different roles, such as acting as an AND-split, AND-join, OR-split or an OR-join [RD98]. Furthermore, loops are supported by ADEPT through edges which point "backwards". These edges have a condition associated with them that defines when this edge will be used [RD98]. The only restriction is that if a loop starts within another loops, the backward pointing edge of the inner loop has to end within the outer loop [RD98], i.e., they are not allowed to overlap.

Change

In ADEPT, it is possible to change an process instance at run-time. This is called an *ad-hoc change* [RD98]. It is possible to insert a task, delete a task or change the sequence of tasks at run-time [RD98]. ADEPT also supports changing the process model and applying changes to process instances that stay valid after applying the change [RRD03]. To support this, it is necessary to check for correctness and validity of the newly created process instance. How this is achieved, is out of the scope of this work. There exists much work about process model and instance checking, such as [RRD04a, RRD04b, RRD03, RD98]. For an extended survey of change patterns and how ADEPT supports them, we refer to [WRRM08].

Conclusion

The basis of ADEPT is built on the traditional imperative workflow paradigm by setting the order of the tasks through the succession operation. Through the support of AND-splits, AND-joins, OR-splits and OR-joins and all reasonable combinations of joins with splits [RD98], ADEPT gains support for workflow patterns [AHKB03, Wor] like parallelism and choice [SMR+08a]. Thus, it obtains more flexibility by design than strict imperative workflow languages but not as much as YAWL (see Section 3.1.2) [SMR+08a], which is specificially modeled to widely support Flexibility by Design.
ADEPT does not support flexibility by deviation or flexibility by underspecification [SMR+08a]. This can be compensated by the support of flexibility by change. ADEPT allows for adaption of the process instance and process model at run-time.
Therefore, flexibility by deviation is not necessarily needed because, instead of deviating from the workflow, the process instance can be changed on-the-fly. The same arguments can be used to address the missing support of flexibility of underspecification.
However, the effort of changing the whole process might be much higher. Moreover, the permissions, given to the user, should be considered. By executing a process with underspecified tasks or with the permission to deviate, gives very specific restrictions on what one can do. Achieving the same flexibility through change means that the same person gets the permission to alter the process instance, which may or may not be the kind of freedom he should have when carrying out his work.
Although this is a huge restriction on mimicing the behaviour of deviation and underspecification, it is at least possible to do so whereas it is not possible or at least far more complicated to imitate flexibility by change with flexibility by deviation or underspecification.
Because ADEPT is built on graph theory, it has a easy-to-learn graphical representation,

which makes it simple to understand and manage process models. ADEPT is an example for a workflow language with the main focus on flexibility by change [SMR+08a, WRRM08].

3.2 Declarative Workflow Languages

Declarative workflow languages receive much attention in research [All83, ASSR93, DHM+96, ASE+96, HLS+99, RF02, WLB03, AP06, LSPG06, PSA07, MPVDAP08, Pes08, APS09, DDHS09, JI09, FMR+09a, FMR+09b, Wag10, HM10b, HM10a, LUW10, UELW10, DDTS11, BDF+12, DDTS12]. There are several fundamentally different declarative workflow languages. Therefore, it becomes useful to classify them.

3.2.1 Classification of Declarative Workflow Languages

In the following we will separate constraint-based languages from those that are not built on constraints. Furthermore, we categorize the constraint-based languages according to the logic on which they are built. Some use the linear temporal logic (LTL) others use computational tree logic (CTL) and some use none of these.
In the remainder of this section we will present the following languages: Object-oriented declarative Workflows (OODW) [DDHS09, DDTS11, DDTS12], Freeflow [DHM+96], Intertask dependencies [ASSR93, ASE+96] and Declare [AP06, PSA07, MPVDAP08, Pes08, APS09, Dec]. They are classified as shown in Figure 3.5.

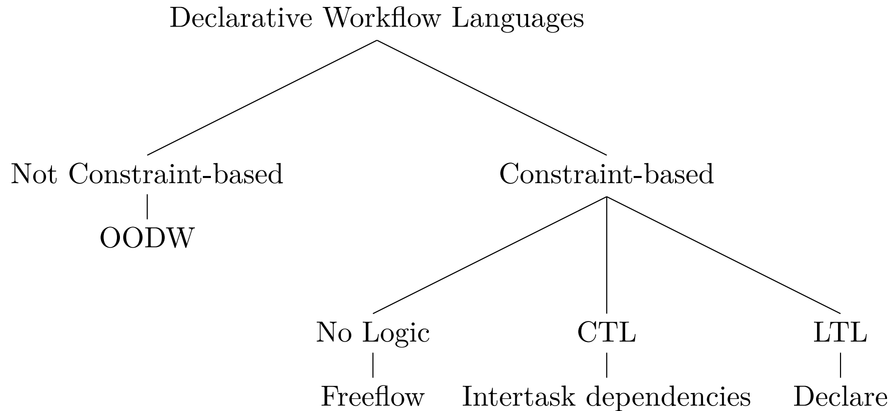

Figure 3.5: Classification of declarative workflow languages.

3.2.2 Object-Oriented Declarative Workflows

Object-oriented declarative workflows [DDHS09, DDTS11, DDTS12] are based on the idea of *active objects*. Therefore, active objects are introduced in the following.

Active Objects

Active objects represent tasks and processes. They are integrated into a database that is also responsible for managing them. The four most important features are presented in the following [DDTS12].

The *firecondition* defines when the active object will be executed. It is similar to a precondition except that it triggers the execution instead of allowing it.

The *endcondition* stops the execution of the active object to which it belongs if the condition evaluates to true.

The *executioncode* defines what is done when the firecondition becomes true.

The *endcode* will be executed if the endcondition forces the object to abort its execution.

Active objects support inheritance. Furthermore, they can be structured hierarchically. An active object can have several sub-objects that can be executed during the execution of the super-object only. The sub-objects have their own fireconditions, which means that they are executed when the super-object is being executed and the firecondition holds at the same time. The super-object can finish only if all sub-objects have finished and its execution code has finished.

The Database

The database contains all active objects and all other resources. This means everything is available in a single data structure.
The database can be accessed and altered with a query language. This makes the approach very flexible because active objects are treated like any other object in the database. All operations, such as addition and deletion, can be performed dynamically. The fireconditions and endconditions are checked cyclically.

Conclusion

Object-oriented declarative workflows do not use a formal logic. Therefore, the process designer does not have to learn one. However, the paradigm of fireconditions and endconditions are closely related to programming and thus, programming experience might help understanding object-oriented declarative workflows.
The database system come with some advantages and some disadvantages. The database supports change easily because active objects are simply new objects in the database and can be altered like everything else. Furthermore, everything, the active objects and all other resources, are contained in one data structure. This can simplify management because everything can be accessed with the database query language.
On the downside, this means that the workflow designer has to have a good understanding of databases and the database query language. In addition, a database does not have a trivial

graphical representation except the table representation of the database. Thus, it is not easy to understand what a process will do by looking at a table. This also holds for the maintenance of the process. Compared to Declare (see Section 3.2.5), which has graphical representations for important and often used constraints and has the ability to add new ones, object-oriented declarative workflows clearly have drawbacks.
Furthermore, the cyclic checking mechanism of fireconditions and endconditions can cause high performance costs. In the case of a large database, it is possible that it takes a relative long time between the periodical checks of one condition. Thus, the evaluation can be delayed. Additionally, the correctness of a process model can not simply be checked. This is due to the fact that object-oriented declarative workflows are not built on a formal logic. Furthermore, every task is defined as a separate active object. The dependencies between these objects are not easily detectable. Therefore, verification and correctness can not be granted.

3.2.3 Freeflow

Freeflow [DHM+96] is a constraint-based workflow language that is not built on a formal logic, such as LTL or CTL. Freeflow is based on a specific life-cycle for tasks (Figure 3.6).

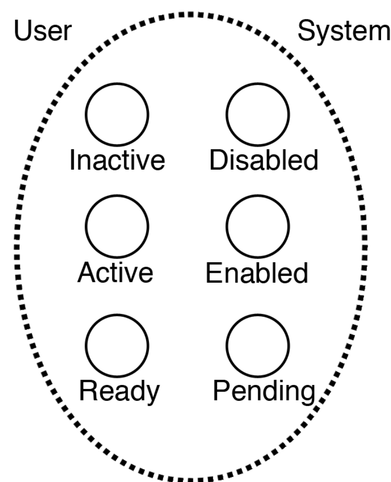

Figure 3.6: Freeflow's task life cycle. [DHM+96]

The states are separated into system states and user states whereupon only the user can switch between user states and only the system can switch between system states. However, it is allowed to switch only to a state that is not deactivated by a constraint.
A constraint is defined as an interconnection of two task states whereupon one state is depending on the other. This means, if a state A_S of task A is depending on a state B_S of task B, state A_S can be entered only if task B_S evaluates to *true*. Every task state can thereby depend on every of the six states of another task.

For some highly used relations between tasks, Dourish et al. implemented higher-level templates [DHM+96] whereupon all templates are also defined by constraints between two task states.
In Freeflow it is allowed to violate a constraint. However, before the violation happens a warning will tell the user that he is about to break a constraint.
Freeflow does not support any model checking techniques, it only checks if a constraint is violated when a task is about to be executed. Also, it does not check for *dead-ends* at run-time. A dead-end denotes a state from which it is not possible to reach a final, satisfying state. Therefore, the responsibility lies with the workflow designer and the workflow user.

Conclusion

Freeflow is based on an easily understandable paradigm. Thus, it is possible to comprehend the full technique of it in a short amount of time. On the downside, this comes with some limitations. Constraints are limited to two task states. Therefore, it is not possible to specify that one task C is allowed to be executed after A or B.
This problem arises because Freeflow is not based on formal logic. This means Freeflow is generally more limited than logic based approaches like Declare (see Section 3.2.5) or intertask dependencies (see Section 3.2.4).
Also, it is not always obvious how to achieve a specific behavior with Freeflow's constraints. Although Dourish et al. implement some relations as templates [DHM+96], it can be complex to specify the right constraints manually if the desired template is not available.
Furthermore, Freeflow does not support model checking and has no formal basis on which it could be achieved. Instance checking is also not available except for checking if a task that is about to be executed, will violate a constraint.

3.2.4 Intertask Dependencies

Although 'intertask dependencies' is a very general term, we use it to identify the workflow language presented by Attie et al. [ASSR93, ASE+96]. The workflow language is based on CTL. Therefore, CTL is shortly presented in the following.

CTL

Computation tree logic (CTL) [EH82, EH86] considers all states of the model that can happen after the initial state. This is represented by the initial node s_0 and nodes directly following s_0 on different branches.These states are followed by all states that can happen after that and so on. The result is the structure of a tree. In the following the states will be represented by tasks meaning that this is the state in which the task is successfully executed. Consider three tasks a, b and c whereupon s_0 is the initial state. If there are no rules on the task order, every order is allowed. The result can be viewed in Figure 3.7 whereupon all concurrent executions of tasks are excluded in this example for the sake of shortness.

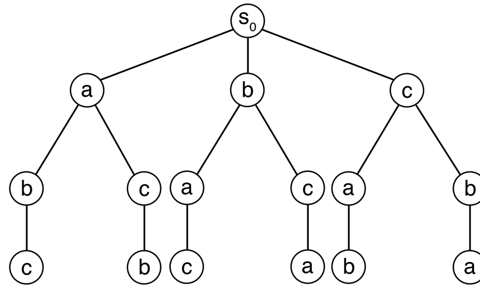

Figure 3.7: Example CTL tree

CTL makes it possible to restrict the allowed order of tasks. If the following rule is specified, the tree is reduced to that in Figure 3.8 where the gray branches represent the orders that are not possible any more.

$$AX[c]$$

The equation states that c have to hold in the immediate successor state, which, in this example, simply means that c has to be executed successfully in the next state.

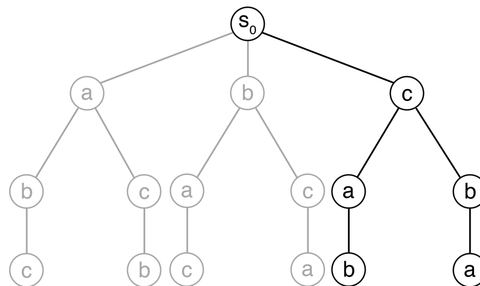

Figure 3.8: Example tree restricted through a CTL rule

CTL formulas can be constructed of the following symbols:

A states that the formula related to A has to hold for every branch that is reachable from the current state (for ***all*** branches),

E states that the formula related to E has to hold for at least on branch that is reachable from the current state (there ***exists*** a branch),

$\boldsymbol{F}[\boldsymbol{\psi}]$ defines that ψ ***finally*** has to hold (for one state at least),

$\boldsymbol{G}[\boldsymbol{\psi}]$ defines that ψ has to hold ***always*** from "now" (the current state) on,

$\boldsymbol{X}[\boldsymbol{\psi}]$ defines that ψ has to hold for the ***next*** state,

$\boldsymbol{\phi U \psi}$ defines that ϕ has to hold at least ***until*** ψ holds and ψ has to hold finally,

$\boldsymbol{\phi W \psi}$ defines that ϕ has to hold at least until ψ holds and ψ is not guaranteed to hold finally (***weak until***),

$\boldsymbol{\neg\psi}$ is the ***negation*** of ψ,

$\boldsymbol{\phi \wedge \psi}$ is the ***logical conjunction*** of ϕ and ψ,

$\boldsymbol{\phi \vee \psi}$ is the ***logical disjunction*** of ϕ and ψ.

This is nether the minimal set nor the full list of all symbols of CTL, but it gives a comprehensible overview on what can be done in CTL and additionally, the missing symbols can be expressed though those that are given above, e.g., the implication $a \rightarrow b = \neg a \vee b$. For an extended description of CTL, we refer the interested reader to [EH82, EH86, ASSR93, ASE$^+$96] whereupon Emerson and Halpern examine the differences of CTL and LTL in [EH86].

The Task Life Cycle

Instead of using the rough approach in which one transition represents the whole execution of a task, intertask dependencies use a finer grained approach (Figure 3.9).

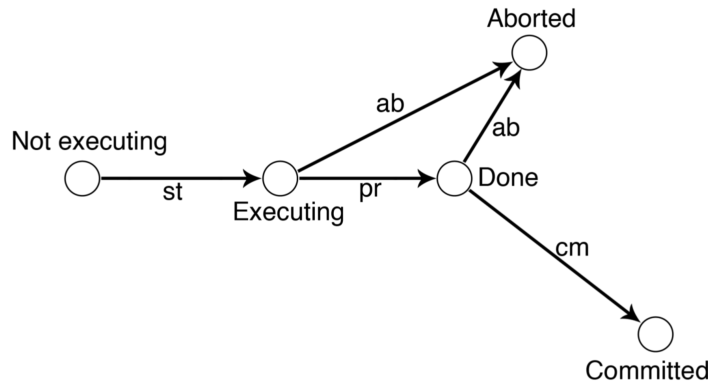

Figure 3.9: Life cycle of a task for intertask dependencies [ASSR93]

All the events of the task life cycle can be used in CTL formulas acting as constraints that have to be followed. The CTL formulas are automatically transformed into finite state automata that enforce the constraints [ASSR93, ASE$^+$96]. Therefore, it is possible to add and remove dependencies at run-time whereupon the removal is trivial and the addition can cause a

violation.

Conclusion

Intertask dependencies are based on CTL through which they gain a formal basis. This allows to transform the dependencies into finite state automata [ASSR93, ASE+96], which enables the power to enforce the dependencies. This means the expressiveness of CTL is available. However, a process designer has to understand CTL and therefore, intertask dependencies are not intuitively usable. For a comparison to an LTL based workflow language, read the Section 3.2.5.

3.2.5 Declare

Declare [AP06, PSA07, MPVDAP08, Pes08, APS09, Dec] is a workflow language that is based on LTL. Therefore, LTL is shortly presented in the following.

LTL

Linear temporal logic (LTL) considers the linear sequence of events. The sequence is temporarily ordered. This means there is one initial state that is followed by another. This one is also followed by another and so on. Thus, LTL does not define rules on all possible orders of states like CTL. Instead, LTL specifies rules on the actual state sequence that will occur. Although it has a great overlap with CTL, there are expression that can be formulated in CTL only and likewise, there are expressions that can be formulated in LTL only [EH86]. The following list of symbols, of which a LTL formula can be constructed, gives an idea of these differences and overlaps

$\Diamond\psi$ defines that ψ ***finally*** has to hold (for one state at least),

$\Box\psi$ defines that ψ has to hold ***always*** from "now" (the current state) on,

$\circ\psi$ defines that ψ has to hold for the ***next*** state,

$\phi\mathcal{U}\psi$ defines that ϕ has to hold at least ***until*** ψ holds and ψ has to hold finally,

$\phi\mathcal{W}\psi$ defines that ϕ has to hold at least until ψ holds and ψ is not guaranteed to hold finally (***weak until***),

$\neg\psi$ is the ***negation*** of ψ,

$\phi \wedge \psi$ is the ***logical conjunction*** of ϕ and ψ,

$\phi \vee \psi$ is the ***logical disjunction*** of ϕ and ψ.

This is nether the minimal set nor the full list of all symbols of LTL, but it gives a comprehensible overview on what can be done in LTL and additionally, the missing symbols can be expressed though those that are given above, e.g., the implication $a \rightarrow b = \neg a \vee b$. For a further description of LTL and a comparison to CTL, we refer the interested reader to [EH86].

Constraints

Constraints are defined through LTL-formulas and they can contain an arbitrary number of tasks.

$$\diamond a$$

Whereupon a is a task. This formula specifies that a has to be successfully executed in the next state.
Declare specifies a task life cycle of three states: *started*, *completed* and *canceled* [Pes08]. The formulas are specified on the life cycle of the task. Applying this to the example above, this results in the following.

$$\diamond a_{started}$$

This formula specifies that a has to be started successfully in the next state.
However, specifying simple constraints can generate complex LTL-formulas. Therefore, it would be useful to be able to reuse an already specified constraint. Declare implements such a mechanism [Pes08]. A graphical representation can be attached to a constraint and the LTL-formula is saved in a generic form [Pes08]. Thus, the graphical representation of a constraint can be applied to graphical representations of tasks. Through this the generic form of the LTL-formula will automatically fill in the tasks for the generic placeholders. Declare includes many predefined constraints (Figure 3.10).

Constraints can be in three states at run-time: *satisfied*, *temporarily violated* or *permanently violated* [Pes08]. When a constraint gets permanently violated, the process instance execution fails.

Verification of Process Models

There are several techniques to translate LTL formulas into automata [GPVW96, GH01, GL02, BKRS12] whereupon all these techniques have their roots in the model checking theory (see Section 2.3). Declare utilizes this for verification [Pes08].
If a task is executed, a transition of the automaton is triggered. However, this is only possible if the execution of this task does not violate any constraint permanently. So, the automaton ensures the correctness of the process instance. Furthermore, it verifies the process model,

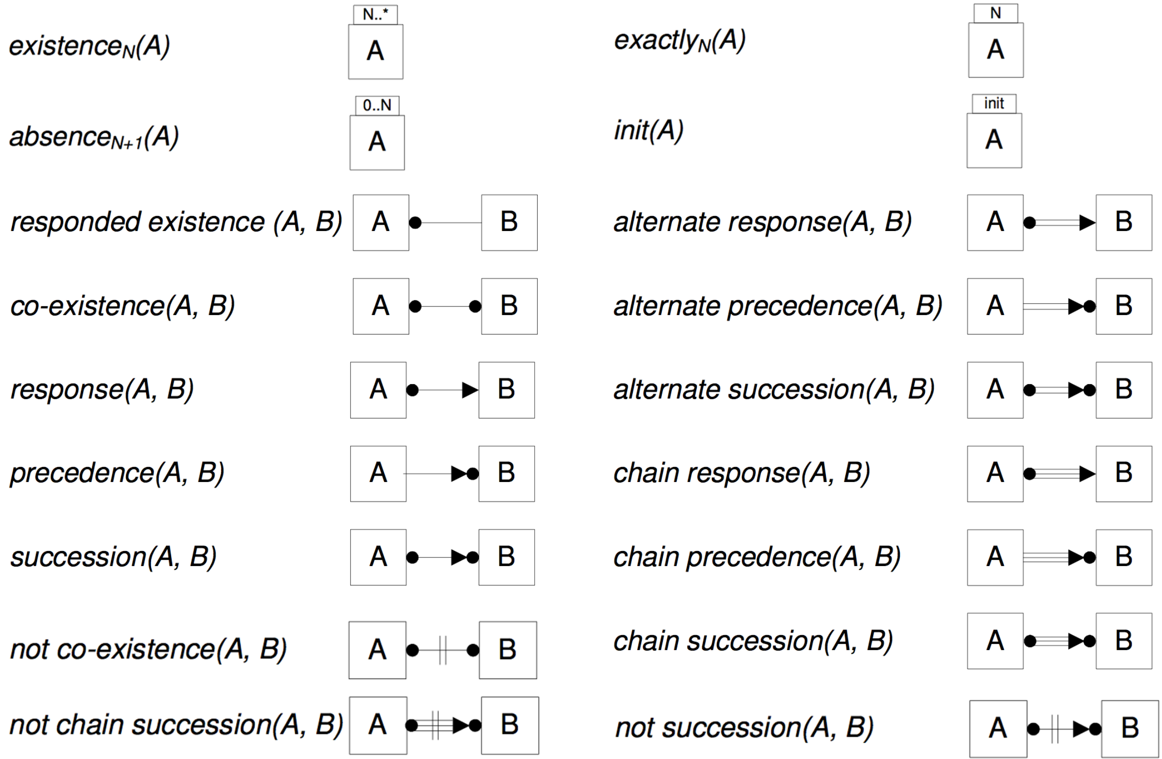

Figure 3.10: The predefined graphical representations of Declare constraints. [Pes08]

because if it is not possible to perform all mandatory tasks without the permanent violation of a constraint, the automaton would be empty [Pes08].
This technique enables Declare to support ad-hoc changes. A change can be applied if no constraint gets violated permanently. Furthermore, it is possible to apply changes to the process model. The evolutionary change will be applied to all new instances and all of the currently running instances that will not become permanently violated because of the change.

Conclusion

Declare gives the possibility to use the whole power of LTL. It is intuitively usable through predefined constraints. However, it is necessary to understand and to be able to specify LTL-formulas if more than the predefined constraints shall be used.
Furthermore, Declare is a workflow language that strongly supports verification and correctness. On the downside, Declare has no support for concurrency of events. Also, there is no possibility to explicitly specify time. For example, it is not possible to define that task b has to start five minutes after task a has finished.
However, in [Wag10, LUW10] it is suggested how these extensions could be implemented.

4 System Model and Problem Statement

In this chapter, the system model (Section 4.1) and the problem that this work is addressing (Section 4.2) will be presented. The system model consist of the network model (Section 4.1.1) and the workflow model (Section 4.1.2). The network model section describes requirements and assumption that are associated to the system. Then the workflow model section states which workflow language is used in order to be able to increase the robustness in a distributed environment. Finally, the problem that this work is addressing, will be stated.

4.1 System Model

In order to increase the robustness in a distributed environment, the system replicates the services and distributes them to different nodes, which execute the replicas. Therefore, we need to define the network model in which such a system works. Afterwards, we define the workflow model because the workflows are the services the nodes have to execute.

4.1.1 Network Model

The system consist of a set of nodes, which are connected by a network. This means that all nodes are able to send to and receive messages from other nodes. All nodes can fail temporarily or permanently like in real-world networks and pervasive systems. Furthermore, each node is able to run and execute workflows locally. The nodes are also able to invoke a service by sending messages to other nodes. This means that a node can generate and distribute workflows to other nodes, which will executed the received workflows.

4.1.2 Workflow Model

The workflow language that is used for this work, is Declare (see Section 3.2.5) with a small modification. In Declare, time is not handled explicitly, but through the specification of constraints. The constraints restrict the order in which the tasks are allowed to be executed. The execution order can be viewed as a chain of time slots including the tasks that happen in the specific time slots.
Consider a workflow consisting of the tasks a, b, c and the order $a \rightarrow b \rightarrow c$, which means that first a will be executed, then b and finally c. The corresponding mapping to time slots can be viewed in Figure 4.1.

Time slot	1	2	3
Tasks	a	b	c

Table 4.1: The example workflow $a \rightarrow b \rightarrow c$ mapped to time slots.

In [Wag10, LUW10] extensions of Declare (see Section 3.2.5) are presented, which will be used to enable concurrency of tasks. Therefore, a workflow consisting of the tasks d, e, f and the order $d\&e \rightarrow f$, which means that first d and e will be executed concurrently and afterwards f will be executed. The corresponding mapping to time slots can be viewed in Figure 4.2.

Time slot	1	2
Tasks	d	f
	e	

Table 4.2: The example workflow $d\&e \rightarrow f$ mapped to time slots.

Thus, the model is a declarative workflow language, namely Declare (see Section 3.2.5), with the additional support of concurrent tasks.
Node that a task happens at a time slot and does not have a separately handled start and ending. Declare is defined to have a starting point of a task and an ending point of a task. Both are constraint by LTL formulas. We use the LTL formulas to specify when a task is allowed to happen and if and how often it has to happen, but not differentiate between start and ending, i.e., the whole task happens at a time slot.

4.2 Problem Statement

In order to be able to increase the robustness on the workflow level by workflow replication, we need to solve the following problems.

1. We need to define a robustness metric to be able to evaluate the robustness of a set of workflows.
2. Because the process model of the deployed workflow language is specified by LTL-formulas, we need to find an algorithm that generates the workflow replicas, which will be distributed to different nodes of the network, from the LTL specification.
3. We need to find a method to select k workflows out of all generated flows such that the robustness is maximized according to the metric.

All workflow executions that satisfy all mandatory constraints of the declarative process model are called workflow specializations. We need to find the most robust set of workflow specializations of size k, so that k different nodes will each execute a different workflow specializations. This means that these workflow specializations have to be created. Checking if a workflow violates any constraint is a time and memory consuming process (see Section 2.3) and also strongly related to creating workflow specializations.

Furthermore, to choose k workflows from n workflow specializations means that we have to solve a combinatorial problem.
The last to two steps of the problem can be combined into one by generating only the k flow specifications that have the highest robustness according to the robustness metric, which is difficult because then the knowledge about the robustness of the workflow is needed before it is generated or during its generation.
In conclusion, this means that a small declarative process model can lead to many workflow specializations, i.e., a large output. The creation of these satisfying flows can result in long run-times and huge memory consumption. The second part, the finding of the most robust set of size k has long run-times itself, but depends on the output of the first part, which makes the performance problem even worse. The alternative is to find a way of evaluating the robustness of the workflow before or during its generation to be able to generate only the set of workflows that has the highest robustness rating.
Therefore, we have to find a way of solving all these problems to be able to approximate the most robust set within reasonable time.

5 Identification and Generation of Robust Workflow Specializations

In this work, the goal is to increase the robustness of the distributed execution of workflows. This is usually achieved by replicating the workflow and executing the exact same behavior on different nodes.
The idea is to construct the process model with a declarative workflow language and to exploit the declarative specification in order to increase the robustness. In specific, a constraint-based workflow language is used (see Section 4.1) and therefore, the tasks can be ordered differently or even replaced by alternative tasks to fulfill the goal of the workflow. Thus, it is possible to generate workflow replicas which differ at run-time but achieve the same goal. Executing these replicas on different nodes increases the robustness of the system because the replicas are decoupled in respect of time and hardware dependencies. To evaluate how much they differ and therefore, how much they increase the robustness, a metric is presented in Section 5.1. This metric allows us to select the most robust set of workflows. The brute-force strategy of generating and selection workflow specializations is described and analyzed in Section 5.2. How a model checking technique is used to generate different workflow orders and can be combined with simulated annealing or the robustness approximation heuristic to overcome the limitation of the brute-force approach is presented in Section 5.3.

5.1 Robustness Metric

In order to increase the robustness, a metric has to be defined that specifies the robustness. The robustness increases the more two workflow replicas differ. To measure how much replicas differ, we introduce the ***workflow specialization distance*** $D_{WS}(w_1, w_2)$ whereupon w_1 and w_2 denote different workflow specializations. A workflow specialization is a workflow for which the task order is fixed, i.e., there are only one way to execute this flow. The terms workflow replica and workflow specialization are used interchangeably in the following.
We define a function $M(x_y, w)$ that returns the time slot(s) in which a task x_y is executed in a workflow specialization w or ∞ if x does not appear in w. If x happens multiple times in w, $M(x_y, w)$ returns the set of all time slots in which x occurs. A set will only be returned if the sublabel $y = \emptyset$ because the sublabels are explicitly defined to ensure this (see below). When x without a sublabel is given to the function (which is equal to the case that x_y with $y = \emptyset$ is given to the function), the function returns all time slots in which a task x_z occurs, no matter what the value of z is.

If x_y with $y \neq \emptyset$ is given to the function, the function only returns the time slot in which x_y occurs.

$$(5.1)\quad M(x_y, w) = \begin{cases} \text{time slot of task } x_y \text{ in specialization } w & \text{, if } y \neq \emptyset \wedge x_y \in w \\ \text{time slot(s) of task } x \text{ in specialization } w & \text{, if } y = \emptyset \wedge x \in w \\ \infty & \text{, otherwise} \end{cases}$$

Time slot	1	2	3
w_{s_1}	a	b	a
w_{s_2}	b	c	a

Table 5.1: Workflow specialization examples

Consider the two example workflow specializations w_{s_1} and w_{s_2} of Table 5.1. $M(a, w_{s_1})$ will return the set $\{1, 3\}$. $M(a, w_{s_2})$ will return 3.
If a task exist more than once in workflows w_1 and w_2, sublabels that unambiguously identify each occurrence of the task in both workflow have to be added. The first occurrence will get the sublabel 1 and the second one the sublabel 2 and so on. Node that the sublabes are only added for calculating the robustness workflow distance. After the calculation, the sublabels will be removed again.
Considering the workflow specializations of Table 5.1, this leads to the workflows of Table 5.2.

Time slot	1	2	3
w_{s_3}	a_1	b_1	a_2
w_{s_4}	b_2	c	a_3

Table 5.2: Workflow specialization of Table 5.1 after adding sublabels

Consider the two example workflow specializations w_{s_3} and w_{s_4} of Table 5.2. Still, $M(a, w_{s_3})$ will return the set $\{1, 3\}$ but $M(a_2, w_{s_3})$ will only return 3.
We define the function $T_S(w)$ that returns the number of time slots the workflow specialization w needs to finish. Additionally, we define a constant s_{max} that is set to the number of time slots the workflow specialization with the most time slots needs to finish. Consider w_x is the workflow specialization that needs the most time slots to finish, then $s_{max} = T_S(w_x)$.
After the definition of Equation 5.1, a function has to be defined that returns the distance of one task x_y in two workflow specializations w_1 and w_2. For simplicity, we assume that the time slots are synchronized for the two replicas. However, this is not required. Although the robustness metric is not as expressive as in synchronous systems, our approach is also applicable to asynchronous system.

(5.2)

$$D_T(x_y, w_1, w_2) = \begin{cases} |M(x_y, w_1) - M(x, w_2)| & \text{, if } x_y \in w_1 \wedge \infty > |M(x_y, w_1) - M(x, w_2)| > 0 \\ |M(x_y, w_2) - M(x, w_1)| & \text{, if } x_y \in w_2 \wedge \infty > |M(x_y, w_2) - M(x, w_1)| > 0 \\ s_{max} & \text{, if } M(x, w_1) = \infty \vee M(x, w_2) = \infty \\ -1 & \text{, otherwise} \end{cases}$$

Because of the way sublabels are added, x_y only occurs multiple times in a workflow if $y = \emptyset$ when calling $M(x_y, w)$ (Equation 5.1), i.e., no sublabel is added to x. In case a set is returned, the operation will be applied to every element of the set. The result will be another set. The minimal element will be chosen as the result of $D_T(x_y, w_1, w_2)$ (Equation 5.2). It is chosen because the smaller the distance of a task in two different specializations (using the time slots as the distance criteria), the more it decreases the robustness. Therefore, the minimal element has to be selected.
Consider the two example workflow specializations w_{s_3} and w_{s_4} of Table 5.2. $D_T(a_3, w_{s_3}, w_{s_4})$ will lead to the calculation of the distance between a_1 of w_{s_3} and a_3 of w_{s_4} and also to the calcualation of the distance between a_2 of w_{s_3} and a_3 of w_{s_4}. This means that the result is the set $\{2, -1\}$. The smallest element is -1 and therefore, the result of $D_T(a_3, w_{s_3}, w_{s_4})$ will be -1.
The result of $D_T(c, w_{s_3}, w_{s_4})$ is s_{max} (which is 3 in this case).

If the result of $M(x, w_1) = \infty$ or $M(x, w_2) = \infty$, the task x occurs in only one of the replicas. This means, the task is an alternative to another task except the task is optional and is additional work, which is handled explicitly (see below). Alternative tasks have an infinite distance to each other because only one of them occurs in one workflow specialization. However, infinity is not a good value to measure the distance because the distances of all tasks define the workflow specialization distance $D_{WS}(w_1, w_2)$ (Equation 5.3). The result is not very expressive if one task alone can determine the output of the function. Therefore, s_{max} is used. This guarantees that an alternative tasks has a better rating than the greatest possible distance of tasks in any two workflow specializations.
If the result is equal to 0, the task is executed within the same time slot in the different workflow specialization. We specify that this case evaluates to -1 instead of 0. This behavior is explained after the definition of the workflow specialization distance.
To get the workflow specialization distance, the distances of all task that occur in workflow specialization w_1 or w_2 have to be added. $w_1 \cup w_2$ denotes the union of all tasks of w_1 and w_2.

(5.3) $$D_{WS}(w_1, w_2) = \sum_{x_y \in w_1 \cup w_2} D_T(x_y, w_1, w_2)$$

Described in words, Equation 5.3 is doing the following.
Firstly, it iterates over all tasks of all time slots of w_1 and searches the nearest time slot in w_2

in which the same task happens. It calculates the distance and adds all the distances of all tasks. If a task does not happen in w_2, it adds s_{max}.
Then the same is done, while iterating over all tasks of all time slots of w_2 and searching in w_1. Both results will be added and that is the result of $D_{WS}(w_1, w_2)$ (Equation 5.3).
Consider the two example workflow specializations w_{s_3} and w_{s_4} of Table 5.2 again. The calcualtion of $D_{WS}(w_{s_3}, w_{s_4})$ is the following.

$$\begin{aligned} D_{WS}(w_{s_3}, w_{s_4}) &= \sum_{x_y \in w_{s_3} \cup w_{s_4}} D_T(x_y, w_{s_3}, w_{s_4}) \\ &= D_T(a_1, w_{s_3}, w_{s_4}) + D_T(b_1, w_{s_3}, w_{s_4}) + D_T(a_2, w_{s_3}, w_{s_4}) \\ &\quad + D_T(b_2, w_{s_3}, w_{s_4}) + D_T(c, w_{s_3}, w_{s_4}) + D_T(a_3, w_{s_3}, w_{s_4}) \\ &= 2 + 1 + (-1) + 1 + 3 + (-1) = 5 \end{aligned}$$

In order to explain the behavior of evaluating to -1 instead of 0 (Equation 5.2), we use an example workflow of three tasks d, e and f which can be executed in any order. Consider the three workflow specializations: $d \to e \to f$ (I), $f \to e \to d$ (II) and $f \to d \to e$ (III).

	Evaluate to 0	Evaluate to -1
$D_{WS}(I, II)$	8	6
$D_{WS}(I, III)$	8	8

Table 5.3: Comparison between evaluating tasks at same time slot in different workflow specializations to 0 or to -1

The workflow specialization distances of this example (see Table 5.3) show that two workflow specializations, in which more tasks are moved (I, III), would not be preferred over two specializations, in which the tasks are less moved (I, II). This means, the situation of the task that happens at the same time slot in different workflow specializations, which is the worst case for this task, is not weighted accordingly.

However, there is still another thing to consider. The case of a workflow specialization that has more tasks than the one with the fewest tasks, has to be handled. Given that w_y is the workflow with the fewest tasks, $t_{min} = |w_y|$ is the minimum number of tasks needed to reach the goal of the declarative workflow. We define, that the absolute value of a workflow is the number of tasks that it will execute if it finishes its execution without any failure.
Node that the workflow with the fewest tasks is not necessarily the flow that needs the fewest time slots to finish.

$$L(w_1, w_2) = ((|w_1| - t_{min}) + (|w_2| - t_{min})) \cdot (s_{max} + 1) \tag{5.4}$$

Consider the two example workflow specializations w_{s_3} and w_{s_4} of Table 5.2 and assume there is another workflow specialization that needs only two tasks and one time slot to finish. Then,

for these three flow specializations $s_{max} = 3$ and $t_{min} = 2$. So, the result of $L(w_{s_3}, w_{s_4})$ is the following.

$$L(w_{s_3}, w_{s_4}) = ((|w_{s_3}| - t_{min}) + (|w_{s_4}| - t_{min})) \cdot (s_{max} + 1)$$
$$= ((3-2) + (3-2)) \cdot (3+1) = 2 \cdot 4 = 8$$

The result of this equation has to be subtracted from the workflow specialization distance (Equation 5.3) because otherwise a longer workflow would increase the robustness rating. Through the subtraction, any additional task decreases the robustness by $s_{max} + 1$. The maximum gain of a single task is s_{max} and hence, decreasing the robustness by $s_{max} + 1$ ensures that the robustness is eventually decreased. The result is the ***extended workflow specialization distance*** $D_{EWS}(w_1, w_2)$.

$$(5.5) \quad D_{EWS}(w_1, w_2) = D_{WS}(w_1, w_2) - L(w_1, w_2)$$

Node that after the calculation of $D_{EWS}(w_1, w_2)$ (Equation 5.5) the sublabels are removed again. They only exist for the calculation of the robustness distance.
Consider the example of above with the two example workflow specializations w_{s_3} and w_{s_4} of Table 5.2 and assume that there is still a third workflow specialization that needs only two tasks and one time slot to finish. Then the outcome of $D_{EWS}(w_{s_3}, w_{s_4})$ is the following.

$$D_{EWS}(w_{s_3}, w_{s_4}) = D_{WS}(w_{s_3}, w_{s_4}) - L(w_{s_3}, w_{s_4}) = 5 - 8 = -3$$

The number of replicas is not fixed to two. More replicas can further increase the robustness of the system. Therefore, the robustness metric has to take this into account. The robustness of a set of workflow specializations S_{WS} of size $k = |S_{WS}|$ is calculated through adding all extended workflow specialization distances $D_{EWS}(w_1, w_2)$ (Equation 5.5) of all possible pairs w_1, w_2 of used workflow specializations S_{WS}. The extended workflow specialization distance is a commutative property, which means $D_{EWS}(w_1, w_2) = D_{EWS}(w_2, w_1)$. Therefore, only one value has to be taken into account.

$$(5.6) \quad R(S_{WS}) = \sum_{(w_1 \in S_{WS})} \sum_{(w_2 \in S_{WS} | w_1 \leq w_2)} D_{EWS}(w_1, w_2)$$

This is the robustness metric that is used in this work. The higher the value, the more robust the set.

5.2 Brute-Force Approach

In order to calculate the robustness of a set of workflow specializations, the set of workflow specializations has to be generated. Furthermore, we want the most robust set that can be created.The brute-force approach is to generate all possible workflow specializations and after that, calculating the robustness of all possible sets of a fixed size k whereupon k specifies the number of workflow specializations and thereby, the numbers of replicas that will be executed on different nodes. Then, the most robust set is selected based on the calculated robustness values.
The brute-force approach is presented and analyzed in the remainder of this section.

5.2.1 Constructing All Workflow Specializations

Algorithmus 5.1 Splitting tasks

```
 1: procedure CONSTRUCT WORKFLOW SPECIALIZATIONS
 2:     R = ∅ ^a
 3:     for all t ∈ T do
 4:         if t can be executed more than once then
 5:             Split t into x_min mandatory tasks and x_opt optional tasks ^b
 6:             Add all new x_min + x_opt tasks to R
 7:         else
 8:             Add t to R
 9:         end if
10:     end for
11:     H = ∅ ^c
12:     RECURSIVE CONSTRUCTION(R, H)
13: end procedure
```

[a] R is a mutable array
[b] x_{min} corresponds to the specified minimum of task instances and $x_{min} + x_{opt} = x_{max}$ corresponds to the specified maximum of task instances.
[c] H is a mutable array

In Algorithm 5.1, we split the tasks that can (and sometimes have to) be executed more than once into tasks that can be executed only once (Algorithm 5.1) whereupon T is the set of all tasks of the workflow. The minimum number of executions x_{min} of a task get converted into x_{min} mandatory tasks, i.e., x_{min} tasks that have to be executed exactly once. The maximum number of executions x_{max} of a task helps calculating the number of optional tasks x_{opt} because $x_{opt} = x_{max} - x_{min}$. If no maximum is set for the task, the maximum is set to a constant x_c before the algorithm is executed. Thus, x_{max} optional tasks are added. An optional tasks has to be executed once or not at all. Therefore, the result is a set of tasks that can be executed only once. This makes it easier to construct all possible workflow specializations.
To be able to apply the constraints correctly and calculate the correct robustness distance between the flows, the tasks have to be recognizable as a child of the task that was split. This

is achieved by keeping the label of the task and adding a sublabel that identifies the child whereas the constraints apply to the label only. This way, the constraints does not have to be edited or replicated. The sublabel also has to identify the task as an optional or mandatory task. The robustness will be calculated without consideration of these sublabels (but add their own).
Consider the LTL-formula $\diamond a \wedge \neg(\diamond(a \wedge \circ(\diamond(a \wedge \circ(\diamond a)))))$ that means that a has to be executed once, but not more than three times. It is split into three separate a tasks that get the sublabels $1m$, $2o$ and $3o$. a_{1m} is a mandatory task whereas a_{2o} and a_{3o} are optional tasks.
After splitting the tasks, a history array H is instantiated. It will be used to store the workflow execution order that is currently investigated. Technically, H is an array of (time-slot) arrays of tasks.
The construction is called recursively (Algorithm 5.2) and with each call H is checked if it is a workflow specialization. If not the next time slot is checked for which of the remaining tasks can happen. For each combination of tasks that can happen at the time slot, the history array H will be replicated and the task combination will be added at the end of the array as a new time slot and then given to the next call of *Recursive Construction*. This way, the history array H contains a workflow specialization, when H ends in a satisfying state.
If we talk about the current workflow execution in the following, we mean the task order that is stored in H (of the current instance of the procedure).

W_S is the set of valid workflow specializations and is saved as an mutable array. It is an array of data structures that are constructed like H. R is the set of tasks that are left for execution, i.e., that are not executed yet since every task can be executed only once. In detail, the procedure *Recursive Construction* (Algorithm 5.2) works as follows.
It receives a workflow execution H and a set of tasks R that were not being executed so far. Initially R contains all tasks and H is empty. First the procedure checks if the received flow execution violates any constraints permanently. If it does, the procedure terminates because then there is no possibility to construct a workflow execution that satisfies all constraints by adding more time slots.
If it does not violate any constraint permanently, the execution proceeds with checking if the flow execution H ends in a satisfying state, which means that R contains only optional tasks and all constraints are satisfied. If it does end in a satisfying state, then H is a workflow specialization and the procedure adds H to W_S and terminates.
If H does not end in a satisfying state, all possible combinations of tasks that can happen at this time slot have to be generated. Firstly, all tasks that can not happen at this time slot will be removed from R and added to container C. Tasks that can not happen at this time slot are tasks that have preconditions (specified by constraints) that do not hold. For instance, task a is only allowed to be executed if b has been executed before. If b was not executed so far (which can be checked in H), a can not happen at this time slot.
There are several preconditions, which can be derived from the constraints of Declare (see Section 3.2.5). We do not discuss them in detail, because this is a speed-up that does not effect the worst-case run-time of the algorithm.
The remaining tasks in R have to be used to produce every possible combination of them. Every possibility to take one task, every possible combination of taking two tasks and so forth. This is achieved by defining a variable n that is set to the number of tasks that R contains.

Algorithmus 5.2 Construction of all possible workflow specializations

```
1: procedure RECURSIVE CONSTRUCTION(R, H)
2:     if H does not violate any constraints permanently then
3:         if H ends in a satisfying state then
4:             Add H to W_S
5:         else
6:             Select all tasks of which one precondition does not hold in R [a]
7:             Remove them from R and add them to C
8:             n = |R|
9:             a[n] = {1, ..., 1} // a is an array of binary values. It determines which tasks are
10:                                    // selected to execute in the current state.
11:            while a > 0 do                  // This considers a to be a binary number.
12:                S = ∅ // It will contain all tasks which will be executed in the current state.
13:                H_n = H     // Generate a new history which can be used for recursive calls.
14:                C_n = C
15:                for (k = 0; k < n; k++) do
16:                    if a[k] == 1 then
17:                        Add R[k] to S
18:                    else
19:                        Add R[k] to C_n
20:                    end if
21:                end for
22:                Add S to H_n
23:                RECURSIVE CONSTRUCTION(C_n, H_n)
24:                DECREMENT ARRAY(a, n − 1)
25:            end while
26:        end if
27:    end if
28: end procedure

29: procedure DECREMENT ARRAY(a, x)
30:     if a[x] == 0 then
31:         a[x] = 1
32:         DECREMENT ARRAY(a, x − 1)
33:     else
34:         a[x]− = 1
35:     end if
36: end procedure
```

[a] In this context preconditions mean that constraints like after, chain succession, etc. can not be met and therefore, the corresponding task can not be executed in the current state.

Then an array of binary values of size n is instantiated. All elements of the array are set to 1. The array will act like an binary number. Within a loop it will be decremented by 1 (by calling *Decrement Array*) until the value of the binary number is equal to 0.
Before decrementing, there is another loop within the loop. It loops over all elements of the array and all elements of R at once. If the ith element of the array is equal to 1 the task will be selected to happen at this time slot, otherwise it will be added to C_n, which gathers all elements that did not happen so far. C_n also contains alll elements of C. So the inner loop selects which elements happen at this time slot based on the binary number (represented by the array). This concludes the inner loop.
A replicate H_n of H is initialized and the currently selected tasks are added as a new time slot at the end of H_n. Then the construction calls itself recursively with the parameters C_n and H_n.
The outer loop constructs all combinations of tasks. After all combinations are constructed and given to their independent next recursive call, the procedure terminates.
The procedure *Decrement Array* (Algorithm 5.2) simply decrements a binary number, which is represented as an array of binary values, by one when called with $n-1$ and an array of size n.
The construction method of Algorithm 5.2 has a drawback. It can not construct workflows that contain a time slot in which no task is executed. If this case would be treated, the procedure would run infinitely because it would have to call the case of executing no task at each time slot and the case of executing no task in a time slot can happen infinite times in a row.
In our experience, the case of an empty time slot rarely occurs in practice but still can possibly happen.

5.2.2 Calculating the Table

In order to get the best set of workflow specializations of the size k from all possible workflow specializations, the extended workflow specialization distances (DEWS, Equation 5.5) for every possible combination of two workflow specializations is calculated and saved in a table (Algorithm 5.3). This means that it does not have to be recalculated for every possible set of workflow specializations of size k but can be read from memory.

The function *Calculate DEWS* is the implementation of Equation 5.5.
The procedure *Calculate All DEWS* first iterates over all workflow specializations to find t_{min} and s_{max} and then simply calls *Calculate DEWS* with all possible combinations of workflow specializations and stores the results in the table P. Thus, the table P includes all extended workflow specialization distances. It is evaluated only above the diagonal because of the commutative behavior of the extended workflow specialization distance (see Section 5.1).

5.2.3 Finding the Most Robust Set

Having the results of Algorithm 5.1, 5.2 and 5.3, it is now possible to calculate a robustness value for all sets of workflow specializations of size k. Out of these sets, the one with the highest robustness has to be selected. This is done in Algorithm 5.4. Moreover, the algorithm returns all workflow specializations as a set if k greater or equal to the number of workflow

Algorithmus 5.3 Calculation of all extended workflow specialization distances

```
1: procedure CALCULATE ALL DEWS(W_S)
2:     t_min = ∞
3:     s_max = −∞
4:     for (x = 0; x < W_S.length; x++) do
5:         if t_min > W_S[x].taskCount then t_min = W_S[x].taskCount
6:         end if
7:         if s_max < W_S[x].timeSlotCount then s_max = W_S[x].timeSlotCount
8:         end if
9:     end for
10:    for (x = 0; x < W_S.length; x++) do
11:        for (y = x + 1; y < W_S.length; y++) do
12:            P[x][y] = CALCULATE DEWS(W_S[x], W_S[y], t_min, s_max)
13:        end for
14:    end for
15: end procedure

16: function CALCULATE DEWS(w_1, w_2, t_min, s_max)
17:    p = 0                                   // Robustness value of w_1 and w_2
18:    for (i = 0; i < w_1.length; i++) do
19:        r = s_max                           // Robustness rating of the current task
20:        for (j = 0; j < w_2.length; j++) do
21:            if (w_1[i] == w_2[j]) ∧ (|i − j| < r) then
22:                if i ≠ j then r = |i − j|
23:                else r = −1
24:                end if
25:            end if
26:        end for
27:        p+ = r
28:    end for
29:    for (i = 0; i < w_2.length; i++) do
30:        r = s_max                           // Robustness rating of the current task
31:        for (j = 0; j < w_1.length; j++) do
32:            if (w_2[i] == w_1[j]) ∧ (|i − j| < r) then
33:                if i ≠ j then r = |i − j|
34:                else r = −1
35:                end if
36:            end if
37:        end for
38:        p+ = r
39:    end for
40:    if w_1.length > t_min then p− = (w_1.length −t_min) · (s_max + 1)
41:    end if
42:    if w_2.length > t_min then p− = (w_2.length −t_min) · (s_max + 1)
43:    end if
44:    return p
45: end function
```

specializations. If there are more workflow specializations than k, the most robust will be calculated as follows.
We initialize an array of binary values with all elements set to 0. The number of workflow specializations determines the size of the array, i.e., the size of the array is equal to the number of flow specializations.
After this, the first k elements of the array are set to 1. All elements that are equal to 1 represent the workflow specializations that will be part of the current set. All elements that are equal to 0 will not be part of the current set. This means that if the ith element of the array is equal to 1, the ith workflow specialization is part of the set. The selection is done by looping over the binary array and the array of workflow specializations at once.
The selection is enclosed in another loop that loops over all possibilities to set k elements to 1 and all others to 0 in the binary array. Within this loop every selection of flow specializations is evaluated for their robustness by the function *Robustness Of Set* of Algorithm 5.5. If its robustness rating is higher than the previously found highest rated set, it will be saved in W_k and the robustness in r_k.
Then, the binary array will be changed by procedure *Next Set* such that it will select the next set of size k and all possibilities will be generated once, when calling it until it has produced the last possibility.
Then the outer loop finishes and the most robust set will be returned.

The function *Robustness Of Set* of Algorithm 5.5 is the implementation of Equation 5.6 with the enhancement of reading the robustness distance from the saved table P instead of calculating it every time it is called.
The procedure *Next Set* of Algorithm 5.5 generates the next possibility of having k elements that are equal to 1. A example clarifies what the procedure is doing.
Consider $k = 3$ and a binary array of size 5. Table 5.4 shows how procedure *Next Set* modifies the array.

Count of calling *Next Set*	The binary array				
0	1	1	1	0	0
1	1	1	0	1	0
2	1	1	0	0	1
3	1	0	1	1	0
4	1	0	1	0	1
5	1	0	0	1	1
6	0	1	1	1	0
7	0	1	1	0	1
8	0	1	0	1	1
9	0	0	1	1	1

Table 5.4: Explanatory example for *Next Set* with $k = 3$ and a binary array of size 5.

The behavior of Table 5.4 is achieved by *Next Set* of Algorithm 5.5 through doing the following. The element of the array that is nearest to the end of the array and is equal to 1, but has

Algorithmus 5.4 Select the set of workflow specializations of size k with the highest robustness value.

1: **function** SELECT MOST ROBUST SET(W_S, k)
2:   **if** $|W_S| \leq k$ **then return** W_S // This means the biggest set possible is returned.
3:   **else**
4:     a[$|W_S|$] = $\{0, ..., 0\}$ // a is a array of binary values. It determines which workflow
5:     // specializations of W_S are selected to form a set of size k.
6:     **for** i in $0..k-1$ **do**
7:       $a[i] = 1$
8:     **end for**
9:     $W_k = \emptyset$ [a] // The most robust set of size k
10:     $r_k = -\infty$
11:     $r_i = 1$
12:     **while** $r_i == 1$ **do**
13:       $W_c = \emptyset$ [b] // Will form a set of size k of which the robustness will be calculated
14:       $W_p = \emptyset$ [c] // Saves the position the workflow specialization has in W_S
15:       **for** i in $0..a.$length-1 **do** // Add k workflow specializations to W_c
16:         **if** $a[i] == 1$ **then**
17:           Add $W_S[i]$ to W_c
18:           Add i to W_p
19:         **end if**
20:       **end for**
21:       r_c = ROBUSTNESS OF SET(W_p)
22:       **if** $r_c > r_k$ **then**
23:         $W_k = W_c$
24:         $r_k = r_c$
25:       **end if**
26:       **if** the last k elements of a are equal to 1 **then**
27:         $r_i = 0$
28:       **else**
29:         NEXT SET(a)
30:       **end if**
31:     **end while**
32:     **return** W_k
33:   **end if**
34: **end function**

[a] W_k is a mutable array
[b] W_c is a mutable array
[c] W_p is a mutable array

Algorithmus 5.5 Helper functions for Algorithm 5.4.

```
1:  function RobustnessOfSet(W_p)
2:      r = 0
3:      for (x = 0; x < W_p.length; x++) do
4:          for (y = x + 1; y < W_p.length; y++) do
5:              r += P[W_p[x]][W_p[y]]
6:          end for
7:      end for
8:      return r
9:  end function

10: procedure NextSet(a)            // e.g. gets a = {1,1,0,0}, produces a = {1,0,1,0}
11:     i = a.length−2
12:     if a[a.length−1] == 1 then
13:         a[a.length−1] = 0
14:         c = 1
15:         while i ≥ 0&&a[i] == 1 do
16:             c += 1
17:             a[i] = 0
18:             i−−
19:         end while
20:         while i ≥ 0&&a[i] == 0 do
21:             i−−
22:         end while
23:         while c > 0 do
24:             i++
25:             a[i] = 1
26:             c−−
27:         end while
28:     else
29:         while i > 0&&a[i] == 0 do
30:             i−−
31:         end while
32:         a[i] = 0
33:         a[i + 1] = 1
34:     end if
35: end procedure
```

at least one element that is equal to 0 that is even nearer to the end of the array than the 1 that is selected. Consider the selected 1 to be the ith element. Then, the ith element is set to 0 and the $i+1^{\text{th}}$ element is set to 1. Furthermore, all elements that are equal to 1 and are in a higher position than $i+1$ are counted and stored in a variable c. All positions of the array higher than $i+1$ are set to 0. Afterwards, the elements $i+2$ up to $i+1+c$ (including $i+1+c$) are set to 1 if $c > 0$.

5.2.4 Run-time Complexity of the Brute-Force Approach

In this section, we will investigate the complexity of the described algorithm. Every part of the algorithm is analyzed on its own.

Algorithm 5.1 implements a loop over all tasks and splits them. The tasks and the constraints are the input of size n. Therefore, the algorithm has a worst-case run-time of $O(n)$. This does not include line 12, which calls Algorithm 5.2.

The significant parts of Algorithm 5.2 are the loop (lines 11-24) and the recursive call within. The tasks and the constraints are the input of size n. The worst case for this algorithm is that there are no constraints because then every order of the tasks is allowed. Thus, the input is considered to be constructed of tasks only. The loop is executed 2^n times because a task can be either executed or not executed in the current state and there are n tasks. Thus there are 2^n possible combinations of executing tasks in the current time slot.
Every time the loop is executed, the procedure is called recursively. Thus, the loop will be called again with the remaining tasks with at least one less task. This means it is executed maximally 2^{n-1} times. Within the loop the procedure is called recursively again and so forth. This results in the run-time that can be viewed in Equation 5.7.

$$\prod_{k=0}^{n} 2^k = 2^{\sum_{k=0}^{n} k} = 2^{\frac{n^2}{2}+\frac{n}{2}} < 2^{n^2} \text{ for } n \geq 1 \quad (5.7)$$

Therefore, Algorithm 5.2 has a worst-case run-time of $O(2^{n^2})$.

The input of Algorithm 5.3 are all workflow specializations. Node that this means n has a new meaning. In this algorithm, there is a double loop over all workflow specializations and within a call of the function *Calculate DEWS*. The function itself has a double loop over all tasks of the workflow specializations. The nesting of those four loops lead to a worst-case run-time of $O(n^4)$.

The input of Algorithm 5.4 are all found workflow specializations. The significant part of the algorithm is the loop (line 11-26). It is executed $\binom{n}{k}$ times because this the number of

possibilities to choose k flows out of n workflow specializations. This leads to the run-time that can be viewed in Equation 5.8.

$$(5.8)\quad \binom{n}{k} = \frac{n!}{k! \cdot (n-k)!} < n!$$

Thus, Algorithm has a worst-case run-time of $O(n!)$.

5.2.5 Mapping to Existing Problems

Since Algorithm 5.1 and 5.3 can be executed in polynomial time, the main interest lies with Algorithm 5.2 and 5.4.

Algorithm 5.2 generates all workflow specializations from a LTL-formula. Sistla and Clarke proved in [SC85] that satisfiability of LTL is a PSPACE-complete problem and therefore, finding satisfying traces, i.e., workflow specializations, is a PSPACE-complete problem.

Algorithm 5.4 evaluates the robustness of all possible sets of size k. It uses the previously generated robustness distance table (Algorithm 5.3). Thus, the problem that is solved, is to find the most robust set of size k out of n workflow specializations. This can be mapped to the *maximum edge-weighted clique problem*, which is NP-hard [AGKW07]. It is defined as follows.
Given a graph $G = (V, E, f)$ (for basics of graphs see Section 2.2) whereupon f is function that assigns a weight to every edge, i.e., $f : E \to \mathbb{R}$. A clique is a complete subgraph, which means that every node of the clique is connected to all other nodes of the clique by an edge. Find the clique in which the sum of all of its edges is maximal and the number of nodes of the clique is less of equal to k whereupon $k \in \mathbb{N}$ [AGKW07].
Treating the workflow specializations as the nodes of a fully connected graph and using the robustness distance table as the adjacency matrix, creates a graph. Thus, the robustness distances act as weights between the nodes of the workflow specializations. This means in the newly constructed graph, finding the most robust set of size k is equal to search for the maximum edge-weighted clique of size k.
Binary quadratic programming can be applied to the maximum edge-weighted clique problem to approximate it via optimization [AGKW07]. The robustness distance table is a symmetric $n \times n$-matrix Q and we search for a vector $v \in \{0, 1\}^n$, such that the outcome of Equation 5.9 is maximal.

$$(5.9)\quad f(v) = \frac{1}{2} v^T Q v$$

If the element i of vector v is equal to 1, the workflow specialization of column i in the robustness distance table is part of the set. If it is 0, it is not. By adding the constraint that

the sum of all elements of vector v has to be k, the set size is restricted to k. Therefore, the finding of the most robust set of size k can be solved by solving the maximum edge-weighted clique problem or by finding the best vector for the quadratic programming approach.
The analysis of the brute-force approach and the mapping to existing problems allows us to find and extend existing techniques to overcome the shortcomings of the brute-force generation of workflow specializations, which is the content of the remainder of this chapter.

5.3 Model Checking Supported Generation of Workflow Specializations

Model checking theory can be used to translate LTL-formulas into automata [Cou99, GH01, GL02, DLP04]. This automaton accepts all words over an alphabet Σ that satisfies the LTL-formulas.
For declarative workflow languages, which are based on LTL, the theory can be applied by using the tasks instead of an alphabet and thereby, the automaton accept all possible workflow executions that do not violate any constraint [Pes08].
Compared to the brute-force generation, the model checking approach has several advantages. Firstly, it is able to construct workflow specializations in which time slots exist that are empty, i.e., in which no task is executed.
Secondly, model checking has been researched deeply and techniques that deal with the problems of model checking, have been proposed [GPVW96, CGP99, Cou99, GH01, ST03, RRD04b, DLP04, RV07, Pes08, BK08, UELW10, FHR11]. This means that using a model checking approach has a widely researched basis and is enhanced in terms of run-time and memory usage.
We use SPOT [SPO, Cou99, DLP04] to generate the automaton from the LTL formulas. SPOT [DLP04, SPO] is based on the paper of Couvreur [Cou99]. Futhermore, SPOT implements techniques of Thirioux [Thi02] and Sebastiani and Tonetta [ST03].
SPOT [DLP04, SPO] is a tableau-based algorithm using *binary decision diagrams* (BDD) [Ake78] for implementation. It takes the formula and looks up which rule of the tableau (see Table 5.5) has to be applied.

Then it performs what is specified by the rule and looks up the next step. This procedure repeats until there is nothing more to do. From the expansion of the formula, the automaton can be easily constructed. For a more detailed description, we refer the interested reader to [Cou99, DLP04].
There are several alternatives that are able to generate automata from LTL-formulas [RV07]. There are two main reasons why we chose SPOT [DLP04, SPO].
Firstly, according to Rozier and Vardi [RV07], it has a good performance and is the only tool which is more than a prototype.
Secondly, it is able to translate directly from LTL-formulas into a *Transition-based Generalized Büchi Automata* (TGBA) [Cou99, GL02] from which we can derive workflow specializations. A TGBA is a Büchi automaton with generalized acceptance conditions attached to the transitions. Also, the events, which, in our case, are tasks, are attached to transitions instead of being

fromula	1st child	2nd child
$\{\Gamma, \neg\top\}$	$\{\Gamma, \bot\}$	
$\{\Gamma, \neg\bot\}$	$\{\Gamma, \top\}$	
$\{\Gamma, \neg\neg\psi\}$	$\{\Gamma, \psi\}$	
$\{\Gamma, \psi \wedge \phi\}$	$\{\Gamma, \psi, \phi\}$	
$\{\Gamma, \psi \vee \phi\}$	$\{\Gamma, \psi\}$	$\{\Gamma, \phi\}$
$\{\Gamma, \neg(\psi \wedge \phi)\}$	$\{\Gamma, \neg\psi\}$	$\{\Gamma, \neg\phi\}$
$\{\Gamma, \neg(\psi \vee \phi)\}$	$\{\Gamma, \neg\psi, \neg\phi\}$	
$\{\Gamma, \neg \circ \psi\}$	$\{\Gamma, \circ\neg\psi\}$	
$\{\Gamma, \phi\mathcal{U}\psi\}$	$\{\Gamma, \psi\}$	$\{\Gamma, \phi, \circ(\phi\mathcal{U}\psi), \mathcal{P}\psi\}$
$\{\Gamma, \neg(\phi\mathcal{U}\psi)\}$	$\{\Gamma, \neg\psi, \neg\phi\}$	$\{\Gamma, \neg\psi, \circ\neg(\phi\mathcal{U}\psi)\}$

Table 5.5: Example tableau of expansion rules of Couvreur [Cou99].

attached to the nodes. An example of a TGBA can be viewed in Figure 5.2. For further descriptions on the TGBA, we refer the interested reader to [Cou99, GL02]. Because of the direct translation into TGBAs from which the flow specializations can be read, it is possible to extent SPOT to support pruning (see Section 5.3.2).

5.3.1 Simulated Annealing

We are able to generate all workflow specializations with the model checking approach but still need to find the most robust set out of all generated flow specializations. As described in Section 5.2.5, the combinatorial problem can be approximated by optimization via binary quadratic programming. Simulated annealing [KV83] is a well known heuristic for solving optimization problems. Katayama and Narihisa [KN01] applied it to binary quadratic programming with a technique of Merz and Freisleben [MF02]. Because the maximum edge-weighted clique problem is a binary quadratic programming problem and simulated annealing is a widely accepted optimization heuristic, we implemented the promising approach of Katayama and Narihisa [KN01] to approximate the most robust set. The workflow specializations that the automaton of the model checking approach produces are the input of the SA algorithm.
However, our implementation of the SA approach differs slightly (compared to Katayama and Narihisa [KN01]) because of slightly different requirements. We shortly present the technique of Katayama and Narihisa [KN01] and discuss our modifications in the following.

Solving Binary Quadratic Programming Problems with Simulated Annealing

Katayama and Narihisa [KN01] use a 1-opt technique in their implementation. This means, that they have a vector $v \in \{0, 1\}^n$ and they switch a single element of the vector from 1 to 0 or from 0 to 1. n corresponds to the size of the n-by-n matrix of the quadratic programming problem.

If an element i of the vector is equal to 1, then the element of the ith row (and column, which is the same) of the matrix is chosen, otherwise (if the i is equal to 0) it is not selected. Translated into our case, this means that if the ith element of the vector is equal to 1, the ith workflow is part of the current set.
As stated above, one element at a time of the vector is flipped. After the flipping, the gain (compared to the vector before the flipping) is checked. The gain g of flipping the currently selected candidate of the vector is calculated according to a formula suggested by Merz and Freisleben [MF02] and a given evaluation metric.
If $g > 0$, the flipping is accepted. Otherwise, it is accepted with a possibility of $e^{\frac{g}{T}}$ whereupon T is initially given to the function and then constantly lowered to lower the possibility of accepting worse results.
Because the first element of the vector that has a positive gain will be accepted, the elements should be checked for their gain in a random order. To do this, a random permutation from 1 to n is generated, which defines the order in which the elements will be flipped. Additionally, Katayama and Narihisa [KN01] calculate a vector that includes all gains the flipping of each element has, i.e., the ith element of this vector is the gain, the flipping of the ith element of vector v will generate. Corresponding to the random permutation, the gain of flipping the particular element is checked.
This whole procedure is enclosed in a while-structure. It counts how many times in a row the result has not been getting better, i.e., $g \leq 0$. If this counter is greater or equal to a given parameter $TermCount$, the loop stops. Otherwise it takes the result from the last run and starts the above described procedure of flipping again with a reduced T. Another given parameter $TFactor$ is used to lower T as follows: $T = T \cdot TFactor$.
In Conclusion, the above described *simulated annealing function* gets a start vector v, a start value for T which is called T_{init}, a value for $TFactor$ and $TermCount$. Based on these, the function iteratively improves the outcome of the quadratic programming problem by changing the vector v. With a decreasing possibility it accepts worse results to overcome local minima.
The simulated annealing heuristic is started by the manual setting of several values: T_{init}, $TFactor$, $TermCount$, $SACount$ and $StartTFactor$.
Afterwards, a random v is generated and the simulated annealing is started.
A for-loop is executed $SACount$ times, in which the *simulated annealing function* is called with parameters v, T_{init}, $TFactor$ and $TermCount$. It returns a vector, which is calculated as described above, and the result is saved in v. Then T_{init} is recalculated by multiplying it with $StartTFactor$.
After the for-loop has finished, the approximation of vector v is returned, which concludes the simulated annealing calculations.
Katayama and Narihisa use the following values for the variables in their evaluation: $T_{init} = 0.3 \cdot n$, $TFactor = 0.99$, $TermCount = 10$, $SACount = 2$ and $StartTFactor = 0.8$. Furthermore, they execute the whole process 30 times and take the best result. We use these values in our evaluation as well and also start the process with 30 randomly chosen vectors with k elements that are equal to 1; the other elements are equal to 0.

Modifications to the Simulated Annealing Approach

The approach of Katayama and Narihisa [KN01] can be used to find the maximum edge-weighted clique but we search the maximum edge-weighted clique of size k. Therefore, it is not sufficient to generate a totally random vector and to flip only one element in the vector. The vector v has to have exactly k elements that are equal to 1 otherwise it would not select a set of size k. This means the 1-opt approach must be adapted. When a 0 is flipped a 1 has to be flipped at the same time and vice versa.
This effects the permutation of the flipping. Instead of creating a permutation of size n, a permutation of size $(n - k) \cdot k$ has to be created. Each element of the new permutation is a pair of values of which the first determines which 0 of the vector is flipped and the second determines which 1 of the vector is flipped.
This means that the inner loop of the simulated annealing function will be executed $(n - k) \cdot k$ times maximally.
Furthermore, we do not calculate all gains in advance, but on demand. This means that we do not create a second vector that saves all gains, but calculate the gain when it is needed.

Conclusion

Simulated annealing is a well known and widely used heuristic for approximating optimization problems. It generates results with a good run-time/performance trade-off with correctly chosen parameters.
The model checking approach of translating LTL-formulas to automata and deriving all possible workflow specializations combined with the SA approach should speed up the calculations enormously. The results will have a reduced quality because they will be approximated but still should have reasonable quality.
Translating the formula to an automaton and extracting all possible workflow specializations is still a PSPACE-complete problem. Therefore, this solution does not reduce the memory consumption because still all workflow specializations will be generated and saved in memory. This means that we have an approach that should be able to generate all workflow specializations and to approximate the most robust set in reasonable time but still have not solved the memory consumption problem for many workflow specializations and huge automata. To solve this problem, we introduce a new technique, which will be presented in the next section.

5.3.2 Automaton Expansion Enhanced by Pruning

While the model checking approach has clear advantages to the brute-force generation (see above), it still suffers from long run-times and strong memory consumption. The simulated annealing approach addresses the run-times of the combinatorial problem but does not solve the memory problems. To overcome these limitations, we introduce pruning to the automaton construction process.
The automaton will be generated with a tableau-based method (see above). The tableau rules that lead to two childs are responsible for splitting of new branches during the expansion.

Therefore, one node of the graph can lead to several different nodes. The idea is to expand only those nodes further that have a high possibility to return workflow specializations that will form a set of workflow specializations with a high robustness. This is done by expanding all current nodes and all the branches that occur during expansion until all direct successor nodes of these nodes are developed. This step is refered to as one *iteration* (Figure 5.1).
After this, all newly retrieved workflow executions are evaluated against each other to retrieve the robustness distance table with a heuristic (see below). Node that the workflow executions do not have to be accepted by the automaton because the expansion is not finished and it might be necessary to expand further to reach an accepting transition. Moreover, it is possible that a branch does not lead to any accepting transition. We will refer to this as a *dead end*.
Consider two tasks a and b and the LTL-formula $\diamond b \wedge \neg b \mathcal{U} a \wedge \square(\neg a \vee \neg b)$. It means that b has to happen at least once and a has to happen before b and that a and b never happen concurrently. Because of $\diamond b = \top \mathcal{U} b$ the second last line in Table 5.5 applies and one child with would be created, in which p would hold. Because a can not happen concurrently and has to hold before b holds, this is a dead end. In this case, the transition and the node, the transitions is leading to, would not be created in the first place because the conflict is obvious within the formula, but there are cases with more complex LTL-formulas that lead to conflicts that only become obvious within the next iteration or even a few iterations later.
Having calculated the robustness table, the most promising workflow executions are selected to expand their nodes in the automaton further (unless they are already accepted by the automaton) (Figure 5.1). This is done after every iteration. Therefore, the state space is smaller and the algorithm speeds up the automaton expansion.

Consider that in Figure 5.1 the transition from node 1 to node 5 would be a transition that fulfills all acceptance conditions. This would make the workflow $b \rightarrow c$ a workflow specializations. Consider that no other transition fulfills all acceptance conditions. If the workflow specializations $b \rightarrow c$ is one of the three most robust flows, which is determined by the heuristic, only two nodes have to be expanded maximally. The satisfying flow does not have to be expanded because it already fulfills all acceptance conditions.
This behavior, together with expanding only parts of the automaton, leads to fewer memory consumption and faster run-times depending on how many flows are selected to expand their nodes further. The variable that defines how many of the highest rated workflows will be followed is called k_{Flows}.
Choosing a high value for k_{Flows} may lead to the full expansion and would save no time or memory. Moreover, it will introduce a overhead because the heuristic will evaluate all workflows after each iteration. Therefore, the heuristic has to be chosen such that the overhead will be minor but the evaluation still gives reasonable quality when the automaton is not fully expanded (due to a smaller value of k_{Flows}). How this is achieved, is presented in the following.

Robustness Evaluation for Pruning

The robustness can not be evaluated like before because it is not possible to get the workflow specialization with the most time slots to finish (to determine s_{max} (see Section 4.1.2)) and

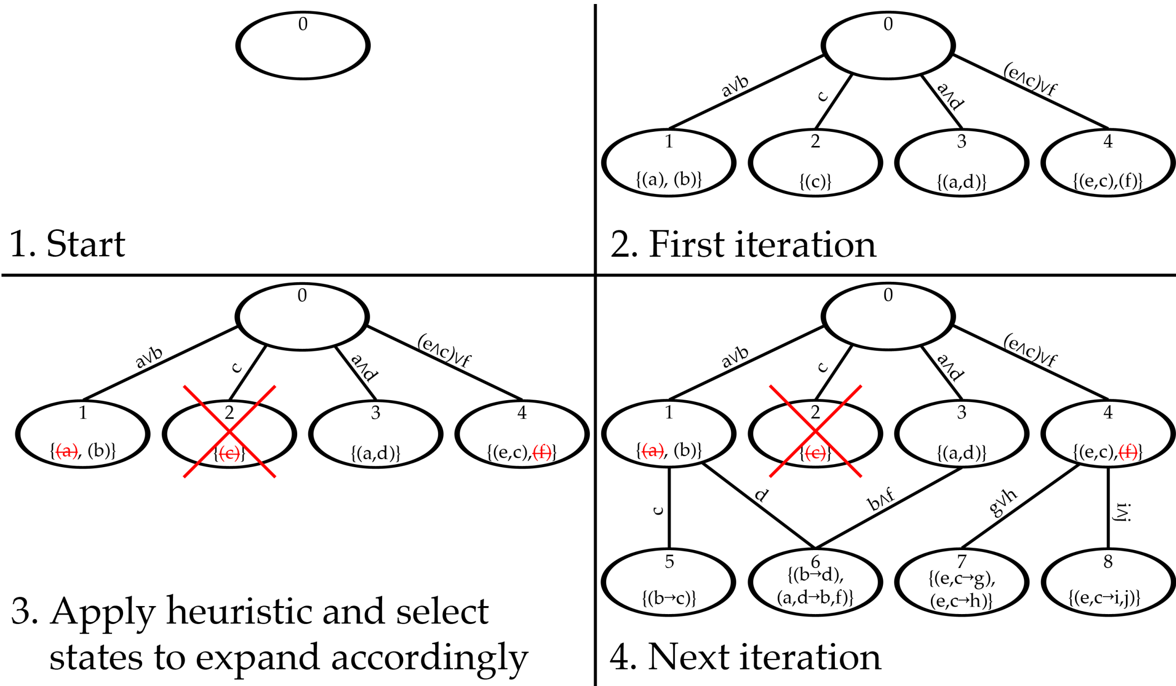

Figure 5.1: This is an example that shows how pruning is applied while expanding an automaton. After each iteration the three most robust workflows are selected to expand their nodes further, i.e., $k_{Flows} = 3$.

the workflow with the fewest tasks (to determine t_{min} (see Section 4.1.2)) without calculating all workflow specializations.
The value of s_{max} is sufficiently replaced by the number of time slots of the workflow execution with the most time slot of all currently available workflow executions. However, it is not obvious how to replace t_{min}. This means that $D_{EWS}(w_1, w_2)$ (Equation 5.5) can not be used. The alternative is to use $D_{WS}(w_1, w_2)$ (Equation 5.3).
Unfortunately, the use of $D_{WS}(w_1, w_2)$ (Equation 5.3) implies some limitations. Additional tasks improve the robustness rating. This would mean that the workflow is better, the more optional tasks are executed. Fortunately, the automaton prevents this. The label of a transition only includes the tasks that are mandatory to happen and those that are not allowed to happen at this time slot to follow this branch. Every task that is not included in the label is allowed to happen without violating any constraint, but does not have to. Hence, it is an optional task. We simply do not add any optional task because they obviously would decrease the robustness value of $D_{EWS}(w_1, w_2)$ (Equation 5.5). There are three exceptions to this.
In Figure 5.2, the first case can be observed looking at the transition from node 2 to node 1. If $\neg a$ does not hold, i.e., it is executed in this time slot, b and c have to be executed. Because we do not execute tasks that are not included in the label, the label will reduce to $y \vee (y \wedge b \wedge c)$ and therefore, the second case will be removed by our algorithm.
Secondly, there are transitions that loop on one state. We do not use these transitions either except they fulfill all acceptance conditions.

Finally, it is possible to use a transition after going through a transition that fulfills all acceptance conditions, i.e., add more tasks to a workflow execution that would be accepted already. We never do this. All three cases occur in Figure 5.2 and are greyed out because they are not used. The accepting transition that loops on node 1 is grayed out because of the third case, meaning node 1 can not be reached with an unaccepting workflow. Otherwise, the transition would not be grayed out.

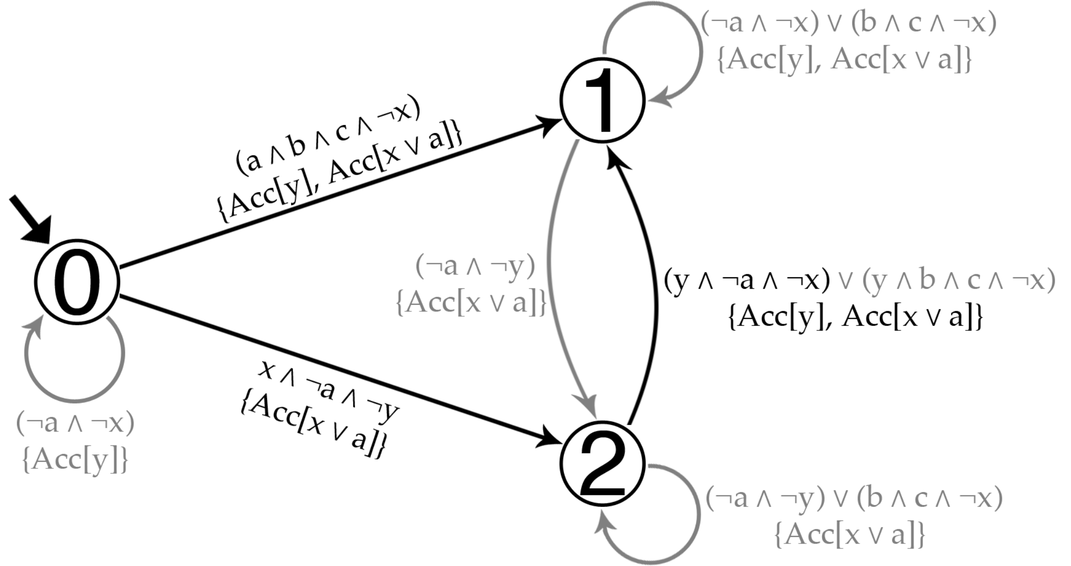

Figure 5.2: The automaton that is generated by SPOT from the formula $\diamond(x \vee a) \wedge \Box(a \rightarrow (b \wedge c)) \wedge \Box(x \rightarrow \neg a) \wedge \Box(x \rightarrow \diamond y) \wedge \Box(x \rightarrow \neg y)$. It has two acceptance conditions Acc$[y]$ and Acc$[x \vee a]$.

Unfortunately, optional tasks are not the only way to get a workflow with extra tasks. Consider the example of Figure 5.2. Executing a, b, c means that only one time slot is needed to finish the workflow successfully, but there is a workflow $x \rightarrow y$ that needs two time slots, but has fewer tasks. After the first iteration all three states are generated, but state 1 and 2 have no outgoing transitions. This means, a, b, c (transition form state 0 to 1) is already known as a workflow specialization whereas x (transition form state 0 to 2) does not fulfill all acceptance conditions and at this step it is unknown how many task this branch needs to fulfill all acceptance conditions. Moreover, the branch could be a dead-end. We defined our heuristic to approach this problem.

The Robustness Approximation Heuristic

The heuristic should be able to decide which intermediate workflows are worth following and which to prune during the expansion of the automaton. It also should have a low run-time complexity. This means that using the brute-force checking of all sets of size k after each iteration would not be sufficient as a heuristic. Therefore, we introduce a the **r**obustness **ap**proximation **h**eurisitc (RAPH).
Firstly, the robustness distance table has to be calculated. As mentioned above, it is not possible to use $D_{EWS}(w_1, w_2)$ (Equation 5.5), because the workflow specialization with the fewest task (t_{min}) is unknown in advance and even might never be found due to the pruning of the automaton. Moreover, it is not obvious how to apply it to intermediate flows. Thus, satisfying workflows and intermediate workflows are treated differently. t_{min} is separated into t_{minS} for satisfying workflows and t_{minI} for intermediate workflows. t_{minS} is determined by the number of tasks of the workflow with the fewest tasks no matter whether the workflow with the fewest tasks is intermediate or satisfying. t_{minI} is determined by the number of tasks of the satisfying workflow with the fewest tasks.
With these new values, $L(w_1, w_2)$ (Equation 5.4) has to be modified whereupon S_I is the set of all intermediate workflows and S_S is the set of all satisfying workflows.

$$(5.10)\quad L_{mod}(w_1, w_2) = L_{mod2}(w_1) + L_{mod2}(w_2)$$

$$(5.11)\quad L_{mod2}(w) = \begin{cases} (|w| - t_{minS})) \cdot (s_{max} + 1) & \text{, if } w \in S_S \\ (|w| - t_{minI})) \cdot (s_{max} + 1) & \text{, if } w \in S_I \wedge |w_1| \geq t_{minI} \\ 0 & \text{, otherwise} \end{cases}$$

Having the these equations, $D_{EWS}(w_1, w_2)$ (Equation 5.5) has to be modified as follows.

$$(5.12)\quad D_{EWSmod}(w_1, w_2) = D_{WS}(w_1, w_2) - L_{mod}(w_1, w_2)$$

With $D_{EWSmod}(w_1, w_2)$ (Equation 5.12) the behavior of $D_{EWS}(w_1, w_2)$ (Equation 5.5) is imitated by the heuristic. A flow with more tasks gets a penalty for every additional task that it needs in comparison to the workflow with the fewest tasks. This means, a pair of two workflows gets the penalty for both workflows and will not be preferred over a pair with fewer tasks. Satisfying flows are judged by the flow with the fewest tasks.
To prevent satisfying flows from being pruned through intermediate workflows with fewer tasks that might turn out to be worse when they are followed, all found satisfying flows will be kept. Intermediate workflows will be evaluated by the satisfying flow with the fewest tasks except they have fewer tasks, then they do not get any penalty at all.
On the one hand, the intermediate workflows that have fewer tasks than already found satisfying

flows have an advantage because a penalty is applied to the satisfying flows. On the other hand, the satisfying flows will be selected by the heuristic if they have a better robustness value and this leads to an expansion of fewer nodes in the automaton.
The heuristic will return the same results like $D_{EWS}(w_1, w_2)$ (Equation 5.5) if all intermediate tasks have more tasks then the satisfying workflow with the fewest task.

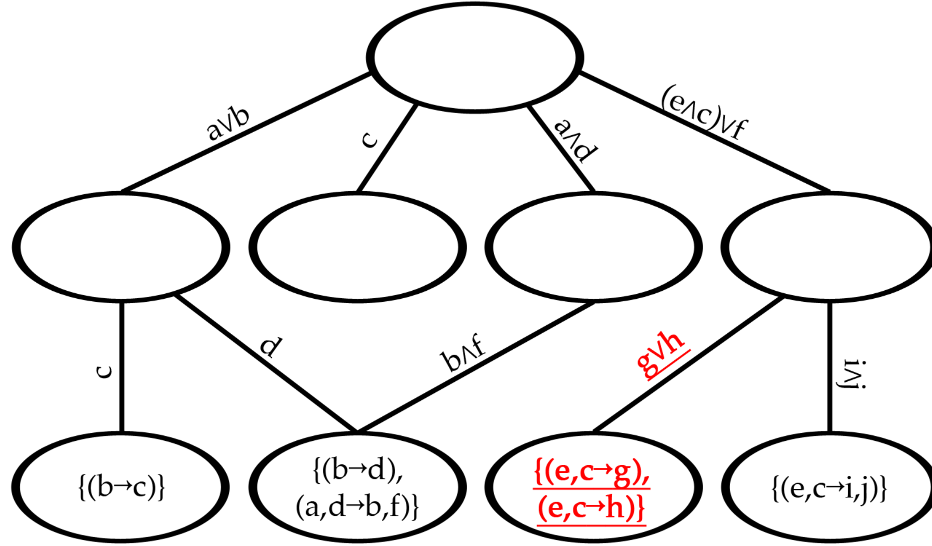

Figure 5.3: This is an example automaton during the expansion (between two iterations, when the heuristic is applied). The underlined, red labels denote accepting transitions and workflow specializations.

In Figure 5.3, there are six workflows. Four are intermediate workflows and two are workflow specializations, i.e., they are accepted by the automaton. Thus, $S_I = \{(b \to c), (b \to d), (a, d \to b, f), (e, c \to i, j)\}$ and $S_S = \{(e, c \to g), (e, c \to h)\}$.
The workflow with the fewest tasks is either $b \to c$ or $b \to d$. Therefore, $t_{minS} = 2$. The satisfying workflow with the fewest tasks is either $e, c \to g$ or $e, c \to h$. Therefore, $t_{minI} = 3$. $s_{max} = 2$ because all workflows need two time slots.
If we select three flows to be followed after each iteration ($k_{Flows} = 3$), then in each iteration three nodes are expanded maximally. Selecting two or three flows of the same node reduces the amount of nodes that will be expanded in the next iteration step.
When one of the workflow specializations is selected, only two nodes are expanded maximally. If both are selected, only one node is expanded maximally (and in this case, it would be exactly one).
The two intermediate workflows $b \to c$ and $b \to d$ will not get a penalty through $L_{mod2}(w)$ (Equation 5.11) because they have fewer tasks than the workflow specialization with the fewest tasks.

The two workflow specializations $e, c \rightarrow g$ and $e, c \rightarrow h$ will get a penalty because they are satisfying flows and there are intermediate flows with fewer tasks. This ensures that intermediate workflows with fewer tasks are preferred. Because the intermediate workflows most likely need more tasks to become workflow specializations and because the heuristic is not equal to the real robustness, all found workflow specializations will be kept until the expansion finishes. Afterwards, we apply the brute-force approach of finding the most robust set, out of all found workflow specializations. Because of the pruning with a correctly chosen k_{Flows}, there will be few and the brute-force algorithm will have a short run-time.
The two intermediate workflows $a, d \rightarrow b, f$ and $e, c \rightarrow i, j$ will get a penalty because they have more tasks than the workflow specialization with the fewest tasks. This is done because expanding the nodes of these workflows only can add more tasks although they are already longer than found workflow specializations. Thus, their rating is lowered by $L_{mod2}(w)$ (Equation 5.11), which results in a lower possibility that their nodes will be expanded further.
This example shows that $D_{EWSmod}(w_1, w_2)$ (Equation 5.12) is defined such that only workflows with a reasonable rating are being followed. With this definition the whole robustness distance table can be calculated. Table 5.6 is showing the robustness distance table for the workflows of Figure 5.3.

	$b \rightarrow c$	$b \rightarrow d$	$a, d \rightarrow b, f$	$e, c \rightarrow g$	$e, c \rightarrow h$	$e, c \rightarrow i, j$
$b \rightarrow c$	-	2	7	5	5	7
$b \rightarrow d$	2	-	5	7	7	9
$a, d \rightarrow b, f$	7	5	-	8	8	10
$e, c \rightarrow g$	5	7	8	-	-6	-4
$e, c \rightarrow h$	5	7	8	-6	-	-4
$e, c \rightarrow i, j$	7	9	10	-4	-4	-

Table 5.6: Robustness distance table for workflows of Figure 5.3 according to $D_{EWSmod}(w_1, w_2)$ (Equation 5.12).

The calculation of the robustness distance table has a worst-case run-time complexity of $O(n^4)$, considering the input consists of n intermediate and satisfying workflows. This table is needed in order to calculate anything, as it is the underlying metric. Furthermore, n will be kept as low as possible through pruning. Thus, this step does not introduce a considerable overhead.
In the next step, the best workflows have to be found, so the nodes corresponding to them can be further expanded while all the others are being pruned, i.e., they will not be expanded.
The column of a workflow in the table includes all distances to all other workflows. Therefore, the column has a lot information about this flow. This is the reason, we use the column of the workflow as the information on which RAPH evaluates the quality of the flows.
Taking only the highest value of the row to evaluate the workflow is a strong hint that the flow is part of the most robust set because a high value means that taking the workflow and the one to which it has the high distance (which is likely to happen, if they have the greatest robustness values out of all workflows) will increase the robustness of the set by a great amount.
The second possibility is to add the $k - 1$ highest values of the column of the workflow and use

the result as the robustness measure. Consider that the workflow is part of the most robust set. The workflow forms the set with $k-1$ other flows. Therefore, $k-1$ values of its column take part in determining the robustness of the set.
The third possibility is to add all values of the column because the value is a metric for the distance to all other flows. This means that no matter which workflows will be chosen for the set, the robustness should always be good. Moreover, taking only 1 or $k-1$ of the highest values into account like above ignores the fact that the highest values of a column might not be part of the most robust set even if the workflow is. Consider the workflows A, B and C. A might have a big robustness distance to B and B might also have a big robustness distance to C. A and C could have such a small or even a negative robustness distance to each other that the set $\{A, B, C\}$ would be a bad choice (assuming $k = 3$). In that case it might be better to use an other available (less rated) flow.
In order to be to use all cases and even steps in between, we define the following.
A parameter m that is given to the heuristic determines how many values of the column of the flow will be taken into account. The m highest values of the row will be added and the result is the robustness approximation of the workflow. This means that m is set to 1, $k-1$ or n for the above described cases.
A second parameter k_{Flows} determines how many workflows will be followed in the automaton. Thus, the nodes of the k_{Flows} highest rated flows, according to the sum of the m highest values, will be further expanded.
The problem that the followed workflow can be a dead-end is approached by the the parameter k_{Flows}. It should always be chosen $\geq k$. The greater k_{Flows} is, the less reaching a few dead-ends matter. Furthermore, choosing k_{Flows} bigger increases the chance of finding a more robust set of size k because more of the automaton will be expanded. On the downside, it means that the performance benefit will be reduced.

	$m = 1$	$m = k - 1$	$m = n$
$b \to c$	7	14	26
$b \to d$	9	16	30
$a, d \to b, f$	10	18	38
$e, c \to g$	8	15	10
$e, c \to h$	8	15	10
$e, c \to i, j$	10	19	18

Table 5.7: Evaluating the quality of the workflows of Table 5.6 with different values for m.

In Table 5.7, the results for rating the flows according to the above described way are given. From this table, the flows that will be followed further will be selected. The selection depends on the second parameter k_{Flows}.
Choosing $k_{Flows} = 3$, $m = 1$ and $m = k - 1$ will both select $(b \to d)$, $(a, d \to b, f)$ and $(e, c \to i, j)$ as the most robust workflows.
$m = n$ selects $(b \to c)$, $(b \to d)$ and $(a, d \to b, f)$ as the most promising workflows.
If $k_{Flows} = 6$, all flows will be followed no matter what value m has. In practice this means, when $k_{Flows} \geq n$, the heuristic does not have to calculate anything, but simply returns the

instruction to follow all flows.
If $k_{Flows} < n$, still all current nodes of the automaton can be expanded because one node can contain more than one flow. Only one flow of every node has to be selected after each iteration to receive a fully expanded automaton in the end. This means that there is no run-time gain for the expansion part but the algorithm will return less workflow specializations, which leads to a speed-up when applying the brute-force selection of the most robust set. Also less workflow specializations will be in the memory.
The fact that the full automaton can be expanded when $k_{Flows} < n$ emphasizes the need for a fast heuristic. RAPH decides in $O(n)$ which workflows will be followed, taking the robustness distance matrix as the input n. Considering that the matrix has to be constructed, it still guarantees a worst-case run-time of $O(n^4)$, taking the n workflows as the input.

6 Evaluation

The SA approach and RAPH that were presented in Section 5.3.1 and in Section 5.3.2, are evaluated in this chapter. In order to this, we implemented an random LTL-formula generator and use SPOT [SPO, Cou99, DLP04] to translate LTL-formulas into automata as described in Section 5.3. SPOT [SPO, Cou99, DLP04] is an open source library; freely availible at [SPO]. SPOT is able to generate an automaton from a LTL-formula that accepts all traces which do not violate the LTL-formula, i.e., it accepts all valid executions of a workflow (see Section 5.3). We implement a simulated annealing technique [KN01, MF02] to solve the combinatorial problem of finding the most robust set (see Section 5.3.1). For the second approach, we also adapted and extended SPOT in order to support pruning (see Section 5.3.2). This means that it is possible to generate an automaton which will not be fully expanded. RAPH (see Section 5.3.2) decides which nodes are worth a further expansion.
In the remainder of this chapter, we search for the optimal value for the parameter m (see Section 5.3.2) of RAPH. Afterwards, we investigate the quality and run-time of our result in comparison to the model checking approach with the brute-force selection of the most robust set. We will also compare these results to the SA technique. Finally, we use LTL-formulas that result in many workflow specializations, which means that the brute-force checking of all of them would be insufficient. Thus, we evaluate RAPH only against the SA approach.

6.1 Measurements

In all figures of this chapter, all values are normalized, meaning that the run-time of RAPH and SA are divided by the run-time of expanding the full automaton and checking all workflow specializations for the most robust set. The robustness is divided by the robustness of the true most robust set. This is done in Section 6.1.1 and Section 6.1.2. In Section 6.1.3, in which the brute-force selection of the most robust set becomes insufficient, the results of RAPH are normalized by the values the SA approach returns. This means that all results are expressed in percent.
In all measurements of Section 6.1.2 and Section 6.1.3 a *Full Expansion* measurement is included. It shows how much time is needed to expand the full automaton and read out all workflow specializations without checking or approximating which will form the most robust set.
Only LTL-formulas which resulted in at least two workflow specialization (expanding the full automaton) were evaluated because if there are less, no robustness value can be calculated. We also do not investigate LTL-formulas which result in more than 500 workflow specializations

when expanding the full automaton and $k = 3$. In case $k = 4$, we do not investigate LTL-formulas which result in more than 200 workflow specializations. We do this because the checking of all workflow specializations to find the most robost set has an exponential run-time behavior (depending on the length of the input, which are the workflow specializations). It would result in very long measurement times and therefore, is avoided here.
In the plots, there are only discrete steps of k_{Flows}. To make the figures easier to investigate, we shifted some functions in x-direction. Thus, if a function has x-values, which are not whole number, its actual x-values are the rounded values of the x-value of the function. Furthermore, the measurements are taken from discrete steps, but to be able to easily see the associated values, we link them with lines even if only whole numbers are possible for k_{Flows}.

6.1.1 Finding the Optimal m for RAPH

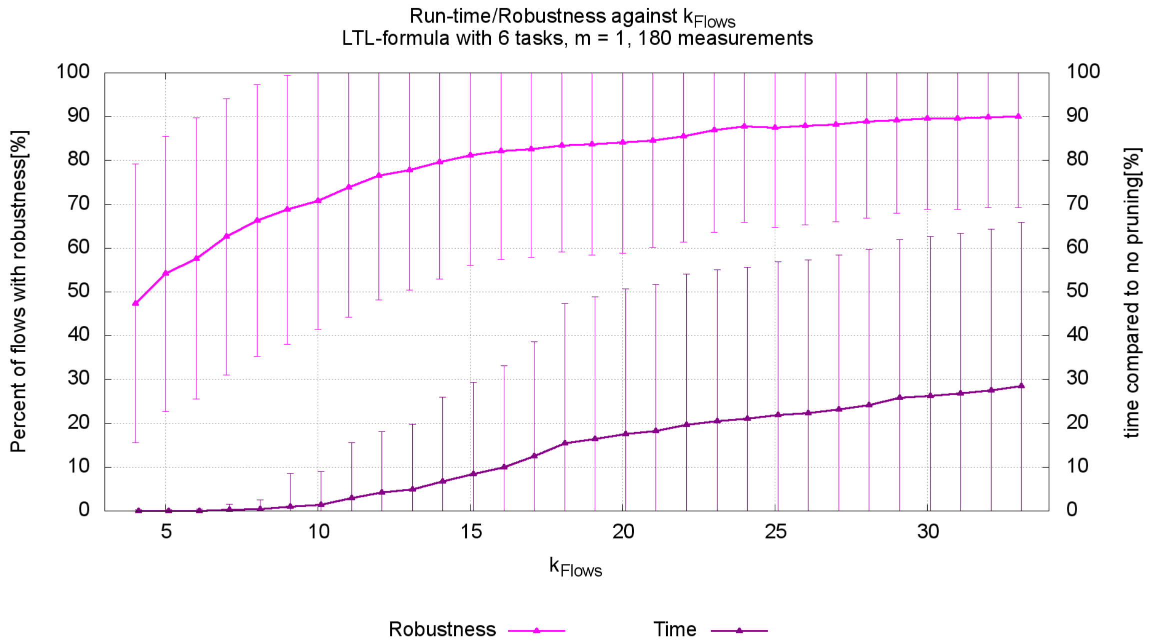

Figure 6.1: Finding the optimal m ($m = 1$, $k = 4$)

Parameter m determines how many (of the highest) values of the column of a workflow in the robustness distance table are added to assign a robustness value to the workflow (see Section 5.3.2).
The most reasonable choices to set m seem to be either 1, $k - 1$ or n. To find the optimal m, we investigate LTL-formulas which result in 100-200 workflow specializations and search for a set of size $k = 4$. We do this, because $k = 3$ would mean that setting $m = 1$ and setting $m = k - 1$ would be a minor difference. The reason we do not include results below 100 workflow specializations is that when few tasks are minimally needed and the full automaton returns few workflow specializations, even pruning with a small value of k_{Flows} often leads to

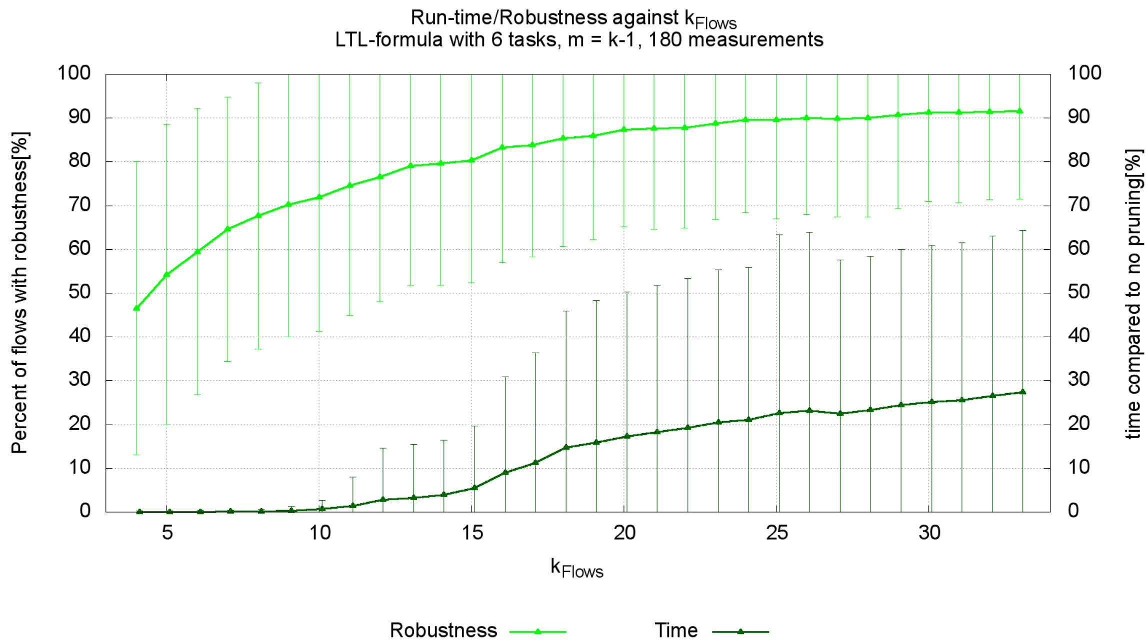

Figure 6.2: Finding the optimal m ($m = k - 1$, $k = 4$)

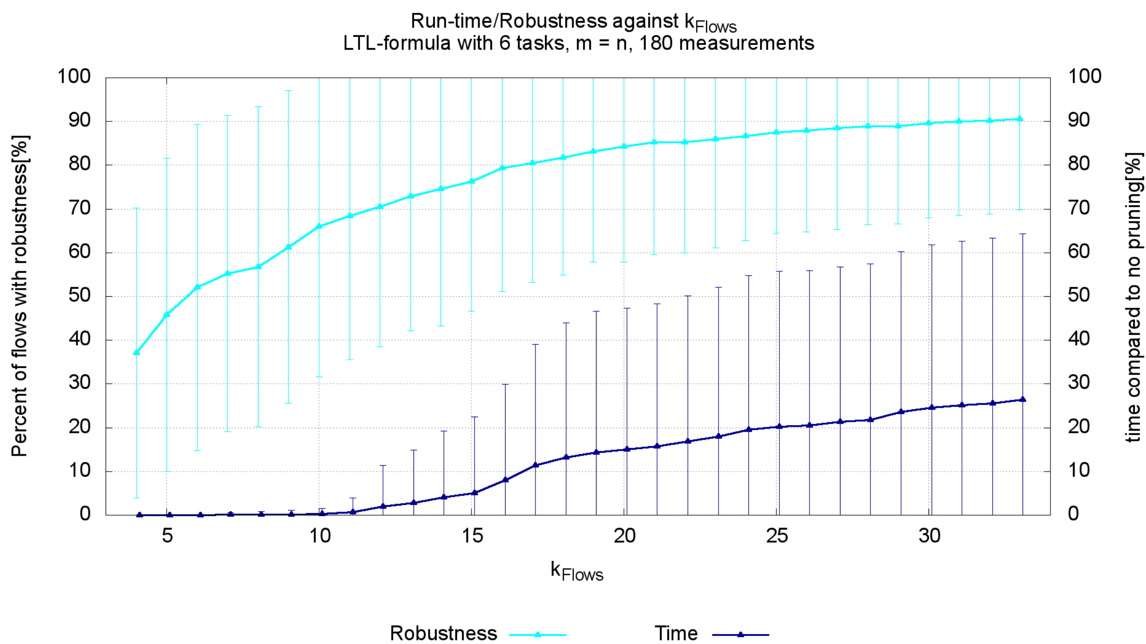

Figure 6.3: Finding the optimal m ($m = n$, $k = 4$)

the full automaton expansion. This means that there is no gain from pruning but we want to find the optimal m when pruning has a gain.
The maximal amount of workflow specializations is set to 200 because this means when all sets of size $k = 4$ are checked for their robustness, already $\binom{200}{4} = 64.684.950$ possible sets exist. The random LTL-formula generator always included six tasks in the formula, which means that at least one task is minimally needed to satisfy the formula.

Figure 6.1, 6.2 and 6.3 basically have the same run-time behavior. Looking at the robustness measurements reveals that $m = n$ (Figure 6.3) has lower quality than the other two, especially at low values for k_{Flows}. $m = 1$ (Figure 6.1) and $m = k - 1$ (Figure 6.2) are very alike regarding the robustness. So both, run-time and robustness are almost the same. Any small difference can be coincidentally. To decide which one we choose, we looked at the lowest value of k_{Flows}. $m = 1$ has around 48% and $m = k - 1$ has around 46% of the robustness. This is why we use $m = 1$.

6.1.2 Performance of RAPH and SA with Few Workflow Specializations

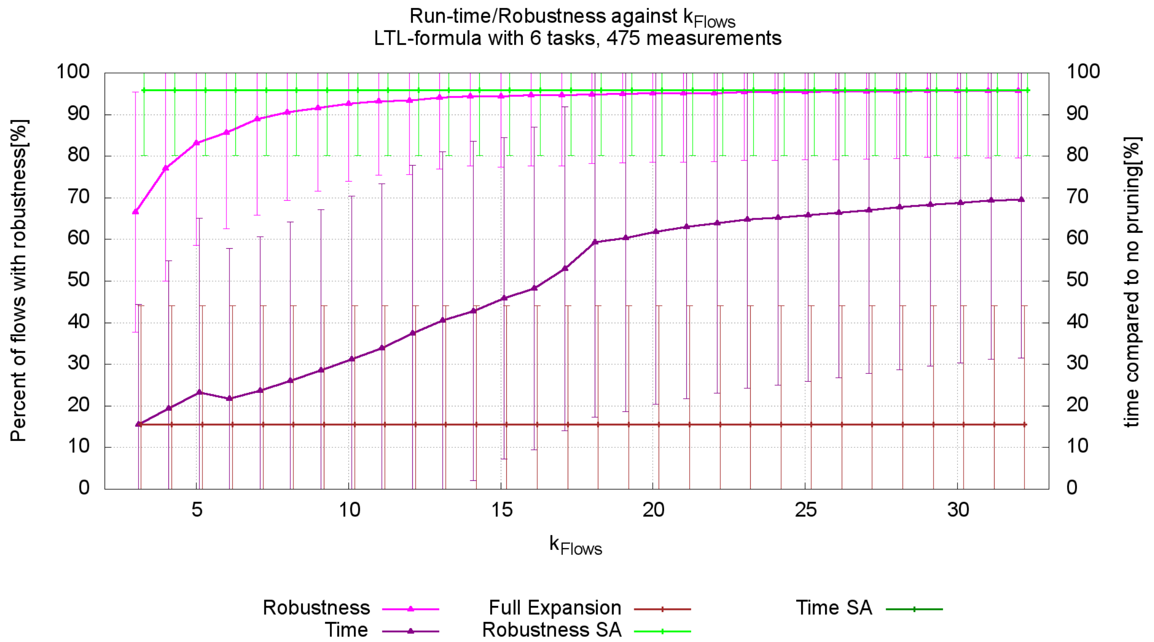

Figure 6.4: Measure for LTL-formulas that resulted in 2-500 workflow specializations ($k = 3$). The time of SA is 340.294% on average and the standard deviation is 600.109%.

For evaluation, we compare the results of RAPH (Section 5.3.2) to the SA approach (see Section 5.3.1). Moreover, both are evaluated against generating all workflow specializations via model checking and using the brute-force selection of the most robust set. To be able to get the results in reasonable time, we only investigate LTL-formulas that result in up to 500

workflow specializations when expanding the full automaton. At least two flows are needed to calculate a distance robustness table, which is why we chose it to be the minimum. We chose $k = 3$. Also, to get a reasonable run-time.
The random LTL-formula generator created formulas that included six tasks. Node that the SA approach has no parameters that we adjust, so the result will be always the same, no matter what k_{Flows} on the x-axis is. The same is true for the *Full Expansion* measurement.
In Figure 6.4, all measurements are combined and the average and the standard deviation are plotted. The SA approach does not seem applicable because of its run-time whereas RAPH saves time (on average) when approximating the most robust set.
The SA approach is usually applied when it is not efficient to use the brute-force method. Thus, we filter out all measurements below 200 workflow specializations (Figure 6.5).

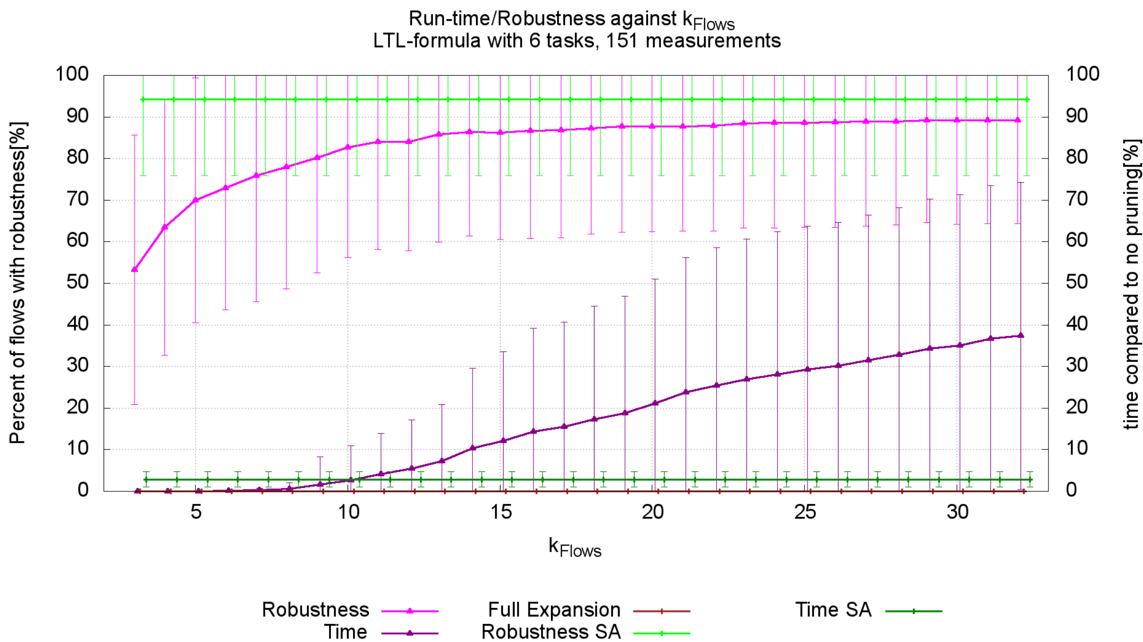

Figure 6.5: LTL-formulas resulting in 200-500 flow specializations ($k = 3$)

In Figure 6.5, it gets obvious that the SA approach gets better, compared to the brute-force checking, the more workflow specializations there are. It has an average of about 95% of the possible robustness and only about 3% of the run-time.
Compared to RAPH, it always returns a better robustness on average, no matter what value k_{Flows} is set to. Comparing the run-time, RAPH is able to perform better when k_{Flows} has a lower value than 10. To perform better than SA, the specific value of k_{Flows} depends on the number of workflow specializations of the full expansion of the automaton and the time needed for the full expansion. Also, it depends on the parameters of the SA approach (see Section 5.3.1). Thus, the value $k_{Flows} < 10$ is no general statement but it shows that RAPH is faster than the SA approach when choosing the right value for k_{Flows}.
The robustness reaches around 80% of the robustness of the true most robust set before the

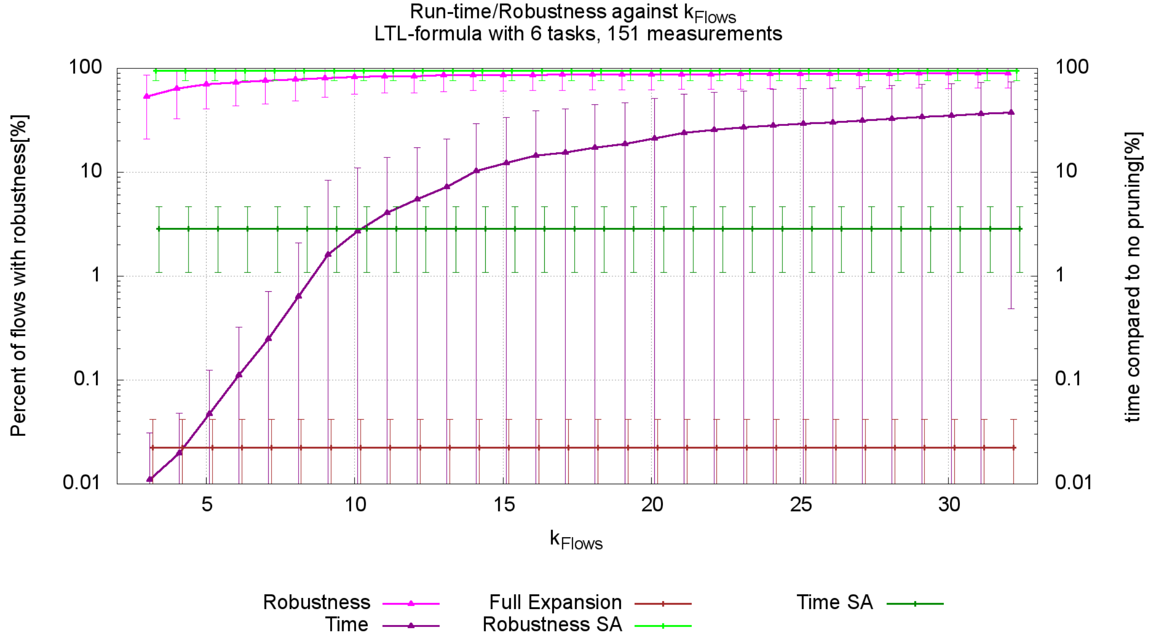

Figure 6.6: Measurement of Figure 6.5 plotted with logarithmic scale ($k = 3$)

run-time takes more time than the SA approach.
In Figure 6.6, it is easy to see that RAPH with pruning averagely performs even better than expanding the full automaton and reading out all possible workflow specializations for $k_{Flows} = 3$ and $k_{Flows} = 4$. This means, that the heuristic is finished before the simulated annealing calculation is able to start. The average robustness value for $k_{Flows} = 3$ is around 54% and for $k_{Flows} = 4$ is around 64% and both have a high standard deviation. Still, the results have a reasonable average quality for the performance.
The plots above are all accumulations of all results, no matter how much tasks the workflow with the fewest tasks (t_{min}) has for the LTL-formula. Figure 6.7 and 6.8 show more differentiated results.

Interestingly, around the point where the time line of RAPH is intersecting the line of the time of the SA approach, RAPH starts to develop a big standard deviation and also seems to climb more steeply.
Furthermore, it seems that a rising value of t_{min} results in a less steep gradient. More tasks mean that the automaton is more likely to be bigger, which results in greater run-times of the full expansion. Following the same amount of k_{Flows} will be relatively faster, but is also more likely to give less robust results because the percentage of found workflows specializations decreases. Therefore, k_{Flows} depends on the value of t_{min}, which is unfortunate because t_{min} is unknown with our approach. There are two possibilites that approach this problem.
Firstly, it is possible to expand the full automaton and calculate t_{min}. Then, our approach can be applied. Secondly, it is possible to estimate the value. This can be achieved by counting all independent $\diamond x$ statements wherupon x can be any task. The counted number

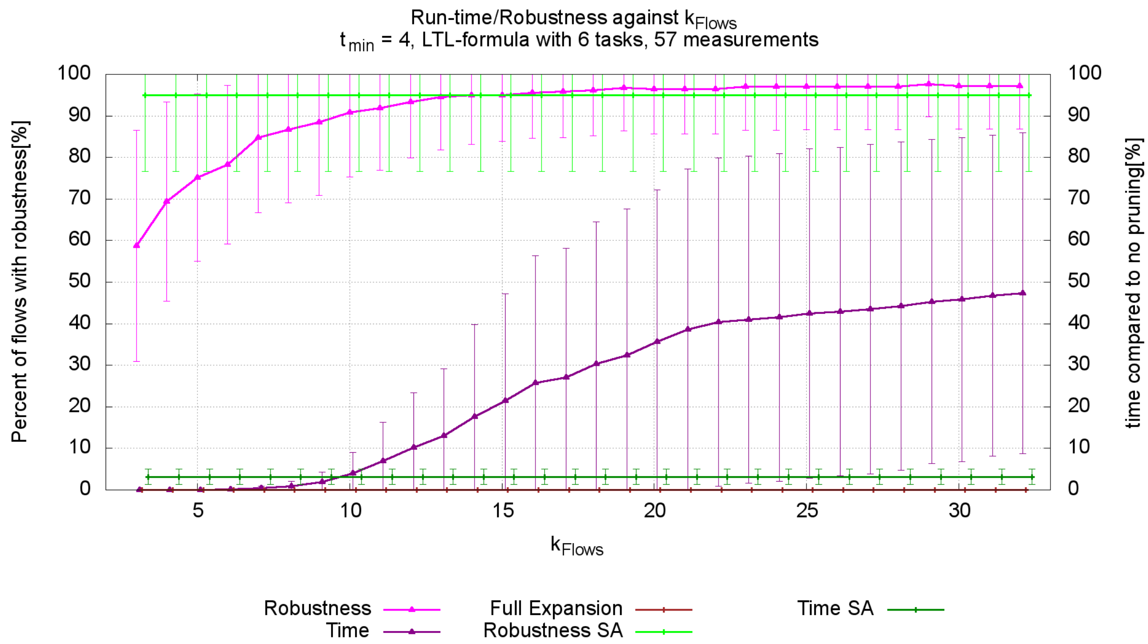

Figure 6.7: LTL-formulas resulting in 200-500 flow specializations ($k = 3, t_{min} = 4$)

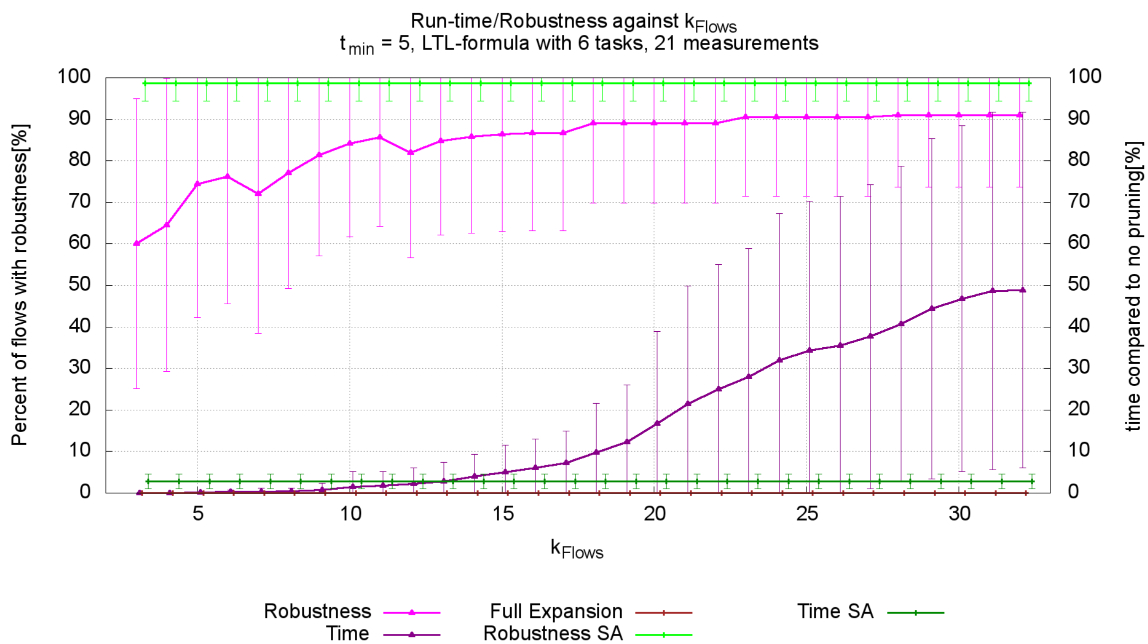

Figure 6.8: LTL-formulas resulting in 200-500 flow specializations ($k = 3, t_{min} = 5$)

is the estamination. For a better approximation, the existence constraints of Declare (see Section 3.2.5) can be used. This is rough estimation but may be more sufficient with very huge automata or time critical applications. Moreover, because only existence constraints are used for it, it always returns the right value or a value that is to low. Therefore, we can guarantee that when we chose k_{Flows} depending on t_{min} correctly, that the run-time will not be longer because of the estimation (because grater t_{min} leads to bigger k_{Flows}, which results in a longer run-time). Unfortunately, we can not guarantee anything for the robustness, when approximating t_{min} this way.

6.1.3 Performance of RAPH and SA with Many Workflow Specializations

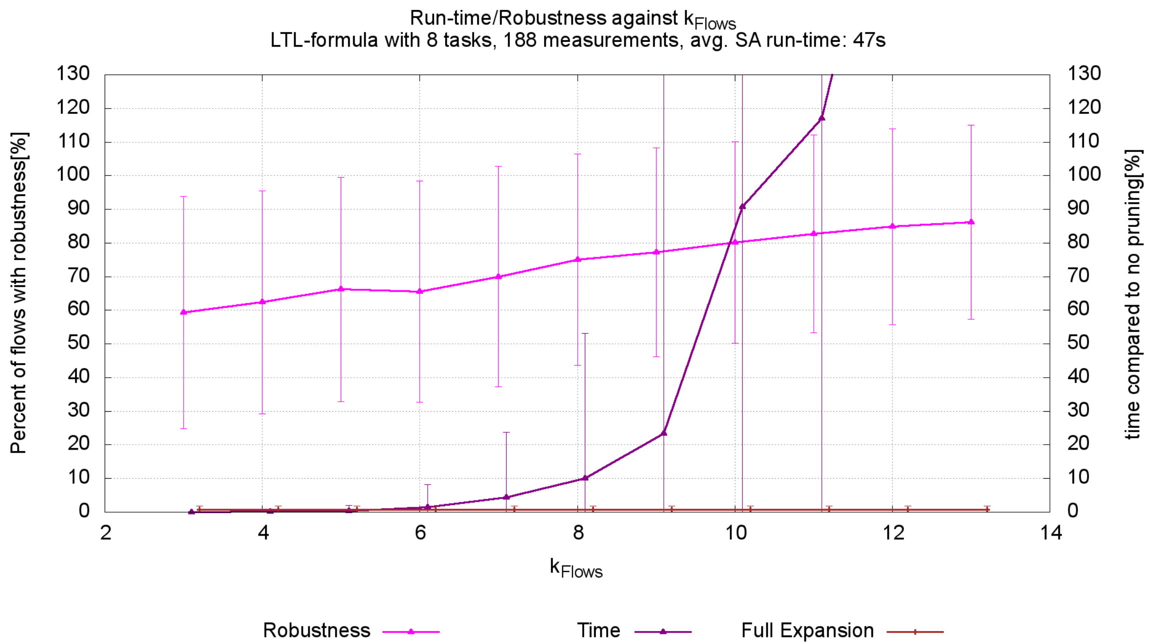

Figure 6.9: LTL-formulas resulting in 500-2500 flow specializations ($k = 3$)

The main application of RAPH are LTL-formulas that result in many workflow specializations because then the checking of all sets for the most robust one becomes insufficient because of the run-time. To evaluate RAPH in this domain, we compare it to the SA approach. Firstly, we run the SA algorithm as described in Section 5.3.1 and then we run RAPH and normalize it by dividing the result with the SA result.

In Figure 6.9, the results from Section 6.1.2 are confirmed. We investigated only formulas which resulted in 500-2500 workflow specializations and $k = 3$. In the following, when we talk about the result, the average is meant.

RAPH is able to perform faster than the SA approach but has decreased robustness results. RAPH does not reach the same robustness as the SA heuristic, even if k_{Flows} is set to a value that the calculation takes longer than the SA approach.

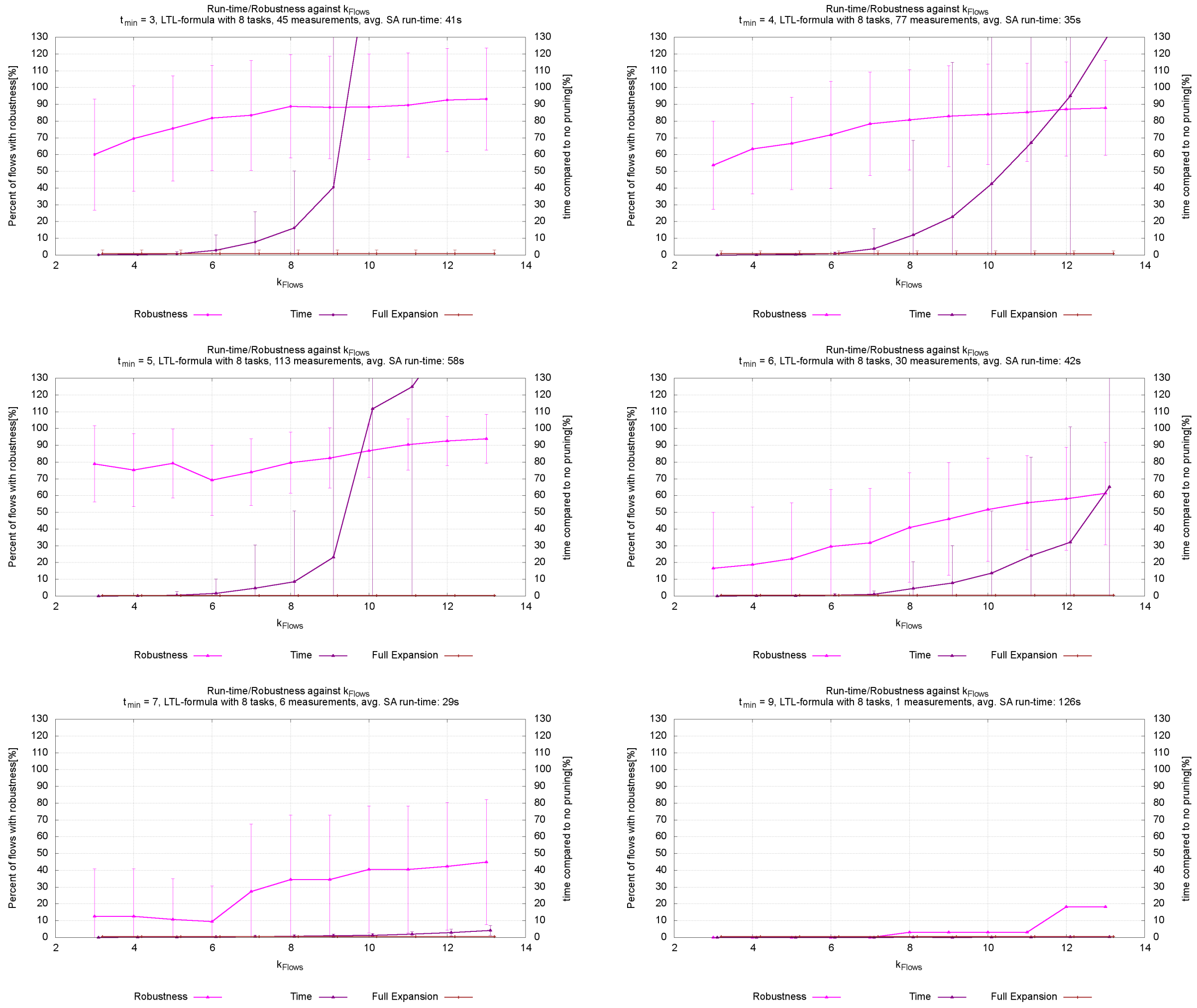

Figure 6.10: These figures show the run-time and robustness plotted against k_{Flows} with the different t_{min} LTL-formulas that result in 500-2500 workflow specializations ($k = 3$).

Figure 6.10 confirms that a greater values for t_{min} result in decreased percental run-times and robustness. The run-times and robustness also increase slower when increasing k_{Flows}. This emphasizes the need for choosing this parameter correctly.
To find a formula to calculate k_{Flows}, we need to investigate higher values for k_{Flows} when t_{min} is also high. Therefore, we do some measurements with LTL-formulas constructed of eight tasks that result in 500-2500 workflow specializations and $t_{min} \geq 7$.

In Figure 6.11 and 6.12, show the trend of Figure 6.10 continues. The run-time and robustness decreases a fixed k_{Flows} when t_{min} increases.
To be able to calculate a k_{Flows} depending on t_{min} we gather all results (from Figure 6.11, 6.12 and 6.10) that have at least 40 measurements for one t_{min}. We chose the

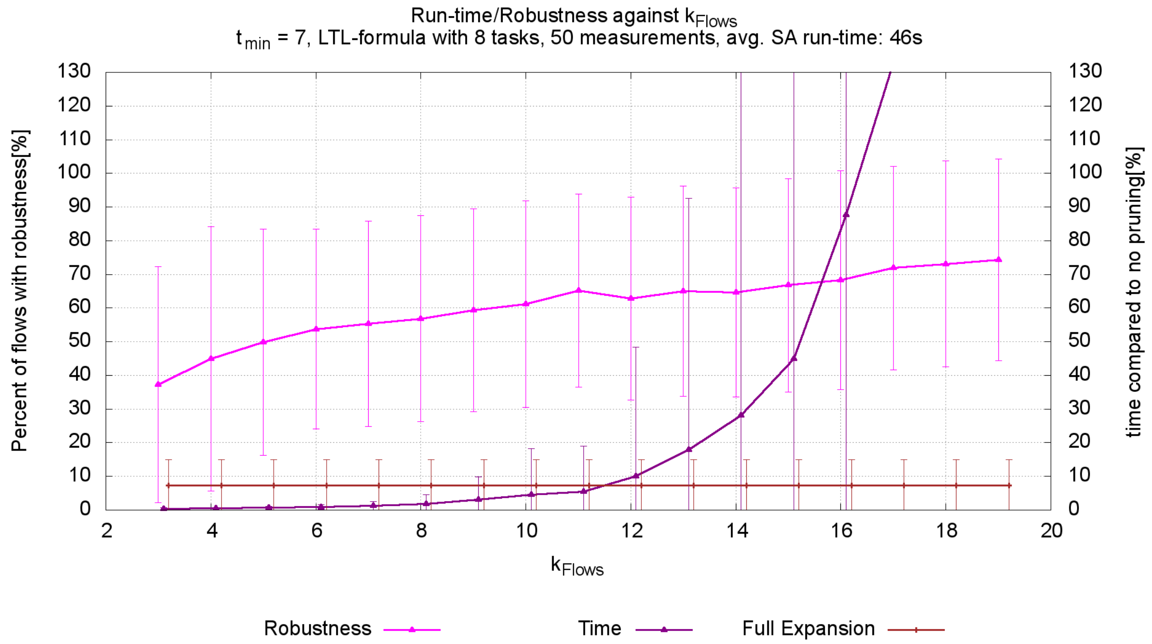

Figure 6.11: LTL-formulas resulting in 500-2500 flow specializations ($k = 3$)

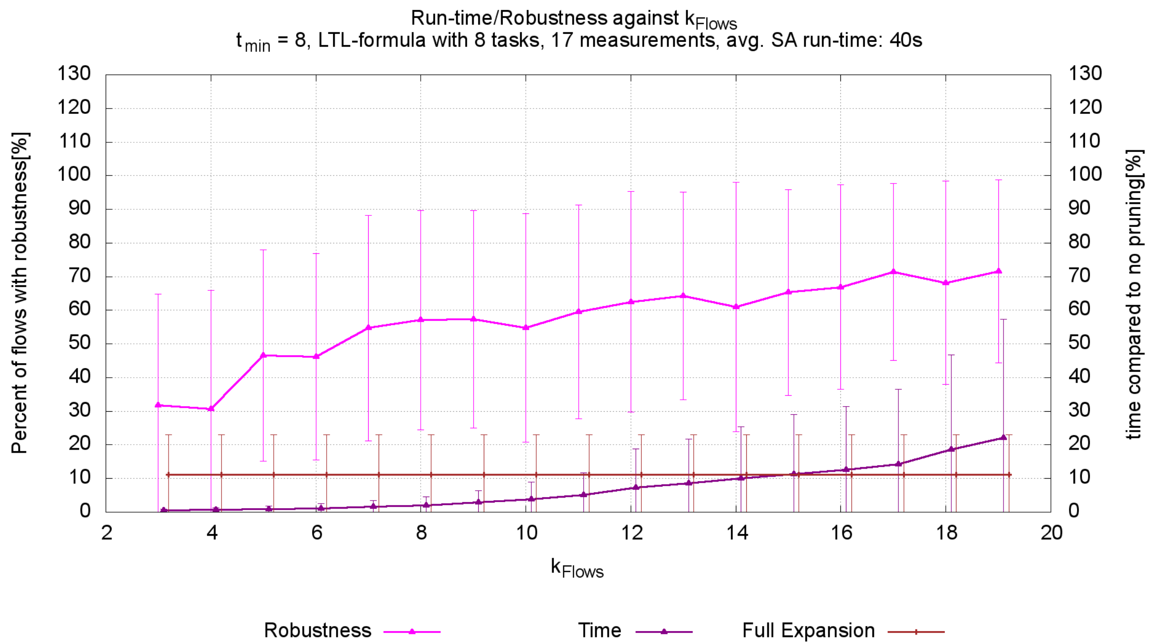

Figure 6.12: LTL-formulas resulting in 500-2500 flow specializations ($k = 3$)

point of 70% of the SA approach of the robustness to be optimal. It has usually a very low run-time and therefore, seems to be a reasonable trade-off. The read values can be viewed in Figure 6.13.

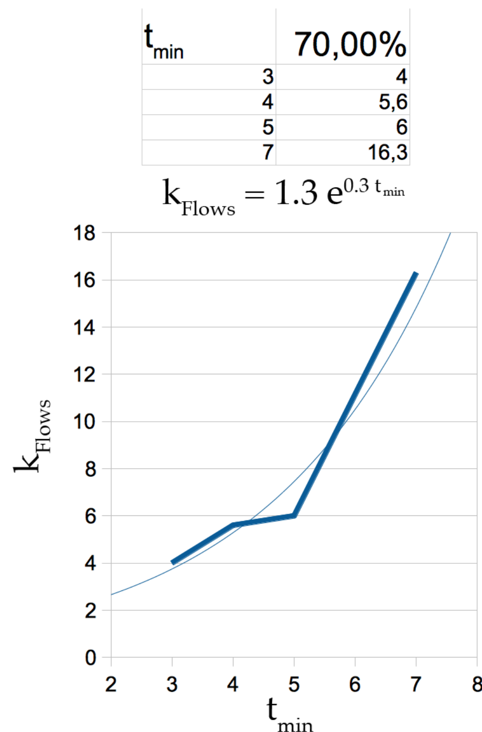

Figure 6.13: Calculating the dependency of k_{Flows} on t_{min}

For $t_{min} = 5$, we chose $k_{Flows} = 6$ instead of $k_{Flows} = 3$ because when increasing k_{Flows}, the robustness also increases, which is not true $k_{Flows} = 3$. Therefore, $k_{Flows} = 6$ seems to be the more stable decision.
From the values, we derive a exponential function $k_{Flows} = 1.3 \cdot e^{0.3 \cdot t_{min}}$. It allows us to calculate k_{Flows}, if we know t_{min}.

6.2 Discussion

RAPH gives sufficiently robust sets within reasonable time. Compared to the SA approach, there are some advantages and disadvantages. On the downside, RAPH returns less robust sets on average with any sensible value for k_{Flows}. Also, k_{Flows} has to be chosen correctly. We recommend $k_{Flows} = 1.3 \cdot e^{0.3 \cdot t_{min}}$. As stated above, t_{min} can be calculated or approximated by analyzing the LTL-formula.
It could be argued that the SA approach has even more parameters that have to be chosen, but we used the values that Katayama and Narihisa [KN01] used themselves and we assume them to be nearly optimal for a trade-off between run-time and quality.

The pros of RAPH are that it improves the run-time hugely and it is able to perform faster than SA ever could, no matter what the parameters of SA are set to. That is because through pruning, the expansion time is reduced so much that the whole process is faster than expanding the full automaton and reading out all workflow specializations. Moreover, RAPH performs faster than the SA approach up to a certain value of k_{Flows}. At the same time, the memory consumption is reduced by pruning. Therefore, RAPH can deal with LTL-formulas that result in automata that has to much states to apply the SA approach.

7 Conclusion and Future Work

To increase the robustness of workflows in a distributed environment, we assign different workflows with the same goal to different nodes in the network. By using a LTL-based workflow language like Declare, we can exploit its declarative nature to generate differently structured imperative workflows for the different nodes. Using alternative and differently ordered tasks increases the robustness because when each node receives a different workflow that achieves the same goal, the possibility of failures is reduced by decoupling the replicas in respect of time and hardware dependencies.
To generate the workflows and select the ones that will increase the robustness most, we apply model checking theory to the LTL-formula. This allows us to translate the LTL-formula into an automaton that accepts all valid flows. To find the most robust set of workflows all sets have to be evaluated for their robustness. Both the generation of the workflows and the finding of the most robust set are expensive calculations regarding their run-time.
Our first solution is to approximate the most robust set of workflows by applying a slightly modified version of a simulated annealing technique of Katayama and Narihisa [KN01]. This speeds up the second part of the problem.
Our second solution is to extend the model checking approach by pruning and therefore, present a heuristic named RAPH that is able to decide during the expansion, which nodes of the automaton should be expanded further, i.e., which nodes will lead to workflows that will be part of a robust set. The heuristic has a parameter k_{Flows} that sets how many intermediate workflows shall be followed further by expanding the nodes of these workflows in the automaton. In doing so, the amount of found workflows that satisfy the LTL-formula is reduced and leads to a faster selection of the most robust set of out of all candidates when evaluating all of them for their robustness.
To evaluate RAPH, we compare it to the full expansion of the automaton with the brute-force checking of all sets for the most robust one. The results show a performance gain that increases with the amount of satisfying workflows, combined with a reasonable quality of the found set. We also compared RAPH to the simulated annealing approach. RAPH is able to perform faster with the correctly tuned parameter k_{Flows} than the SA approach. Though, it does not provide the same quality on average. However, RAPH is applicable to LTL-formulas that lead to small automata and few workflow specializations whereas the SA approach then takes longer than the brute-force method. Moreover, RAPH has a reduced memory consumption due to pruning and therefore, is applicable to LTL-formulas that results in bigger automata. However, the performance gain and the quality of results strongly depends on choosing k_{Flows} correctly. Therefore, we provide a recommendation how it should be set.
We conclude that the SA approach can be used to save time and receive good quality. RAPH can be applied to approximate the most robust set with reasonable performance and works even faster than the SA approach while having a lower memory consumption, but delivers

reduced quality of robustness. This means that RAPH can be applied to cases in which SA becomes unattractive because of its run-time and memory consumption when expanding the full automaton.

Future Work

The robustness can be increased further by supporting continuous time as an extension to the discrete model. The tasks are often not of equal length and therefore, start and end should be separate events in the time line, like in Declare (see Section 3.2.5). This would also increase the generated automaton by a huge amount. Integrating task time length into pruning to keep the automaton smaller and only follow traces that not only fulfill the LTL-formula, but also the task length, should be investigated.
Introducing continuous time would also require an adjustment of the robustness metric because it is completely specialized to support discrete time slots. This would result in a great benefit to any real-world example in which the run-time of each task can be measured or is known in advance.

Bibliography

[Aal96] W. M. P. van der Aalst. Three Good reasons for Using a Petri-net-based Workflow Management System. In S. Navathe, T. Wakayama, editors, *Proceedings of the International Working Conference on Information and Process Integration in Enterprises (IPIC'96),*, pp. 179–201. 1996. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.1056. (Cited on pages 24 and 25)

[AB01] W. M. P. van der Aalst, P. J. S. Berens. Beyond workflow management: product-driven case handling. In *Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work*, GROUP '01, pp. 42–51. ACM, New York, NY, USA, 2001. doi:10.1145/500286.500296. URL http://doi.acm.org/10.1145/500286.500296. (Cited on pages 20 and 26)

[ACRH09] M. J. Adams, S. Clemens, M. L. Rosa, A. H. ter Hofstede. YAWL : power through patterns. In A. K. A. de Medeiros, B. Weber, editors, *BPM '09 : Business Process Management Conference*. CEUR-WS.org, Ulm, Germany, 2009. URL http://eprints.qut.edu.au/29113/. (Cited on page 25)

[ADEa] URL http://www.uni-ulm.de/in/iui-dbis/forschung/projekte/abgeschlossene-projekte/adept1.html. (Cited on pages 20 and 28)

[ADEb] URL http://www.uni-ulm.de/in/iui-dbis/forschung/projekte/abgeschlossene-projekte/adept2.html. (Cited on pages 20 and 28)

[AGKW07] B. Alidaee, F. Glover, G. Kochenberger, H. Wang. Solving the maximum edge weight clique problem via unconstrained quadratic programming. *European Journal of Operational Research*, 181(2):592 – 597, 2007. doi:10.1016/j.ejor.2006.06.035. URL http://www.sciencedirect.com/science/article/pii/S0377221706004759. (Cited on page 57)

[AH04] W. van der Aalst, K. van Hee. *Workflow Management: Models, Methods, and Systems*, volume 1 of *MIT Press Books*. The MIT Press, 2004. URL http://ideas.repec.org/b/mtp/titles/0262720469.html. (Cited on pages 13 and 23)

[AH05] W. van der Aalst, A. H. M. T. Hofstede. YAWL: yet another workflow language. *Information Systems*, 30:245–275, 2005. URL http://www.sciencedirect.com/science/article/pii/S0306437904000304. (Cited on pages 20, 22, 24 and 25)

[AHEA06] M. Adams, A. Hofstede, D. Edmond, W. Aalst. Worklets: A Service-Oriented Implementation of Dynamic Flexibility in Workflows. In R. Meersman, Z. Tari, editors, *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, volume 4275 of *Lecture Notes in Computer Science*, pp. 291–308. Springer Berlin Heidelberg, 2006. doi:10.1007/11914853_18. URL http://dx.doi.org/10.1007/11914853_18. (Cited on pages 20 and 25)

[AHKB03] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, A. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14:5–51, 2003. doi:10.1023/A:1022883727209. URL http://dx.doi.org/10.1023/A:1022883727209. (Cited on pages 21, 22, 25, 28 and 29)

[Ake78] S. Akers. Binary Decision Diagrams. *Computers, IEEE Transactions on*, C-27(6):509 –516, 1978. doi:10.1109/TC.1978.1675141. (Cited on page 58)

[All83] J. F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983. doi:10.1145/182.358434. URL http://doi.acm.org/10.1145/182.358434. (Cited on page 30)

[AP06] W. M. P. van der Aalst, M. Pesic. DecSerFlow: Towards a Truly Declarative Service Flow Language. *Lecture Notes in Computer Science : Web Services and Formal Methods, Volume 4184, 2006*, pp. 1–23, 2006. URL http://dx.doi.org/10.1007/11841197_1. (Cited on pages 30 and 36)

[APS09] W. van der Aalst, M. Pesic, H. Schonenberg. Declarative workflows: Balancing between flexibility and support. *Computer Science - Research and Development*, 23:99–113, 2009. doi:10.1007/s00450-009-0057-9. URL http://dx.doi.org/10.1007/s00450-009-0057-9. (Cited on pages 19, 20, 30 and 36)

[ASE+96] P. C. Attie, M. P. Singh, E. A. Emerson, A. Sheth, M. Rusinkiewicz. Scheduling workflows by enforcing intertask dependencies. *Distributed Systems Engineering*, 3(4):222, 1996. URL http://stacks.iop.org/0967-1846/3/i=4/a=003. (Cited on pages 30, 33, 35 and 36)

[ASSR93] P. C. Attie, M. P. Singh, A. P. Sheth, M. Rusinkiewicz. Specifying and Enforcing Intertask Dependencies. In *Proceedings of the 19th International Conference on Very Large Data Bases*, VLDB '93, pp. 134–145. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. URL http://dl.acm.org/citation.cfm?id=645919.672807. (Cited on pages 30, 33, 35 and 36)

[ASW03] W. van der Aalst, M. Stoffele, J. Wamelink. Case handling in construction. *Automation in Construction*, 12(3):303 – 320, 2003. doi:10.1016/S0926-5805(02)00106-1. URL http://www.sciencedirect.com/science/article/pii/S0926580502001061. (Cited on pages 20 and 26)

[AWG05] W. M. van der Aalst, M. Weske, D. Grünbauer. Case handling: a new paradigm for business process support. *Data Knowl. Eng.*, 53(2):129–162, 2005. doi:10.1016/j.datak.2004.07.003. URL http://dx.doi.org/10.1016/j.datak.2004.07.003. (Cited on pages 20, 26, 27 and 28)

[BDF+12] J. Brzeziński, A. Danilecki, J. Flotyński, A. Kobusińska, A. Stroiński. ROsWeL Workflow Language: A Declarative, Resource-oriented Approach. *New Generation Computing*, 30:141–164, 2012. doi:10.1007/s00354-012-0203-y. URL http://dx.doi.org/10.1007/s00354-012-0203-y. (Cited on page 30)

[BK08] C. Baier, J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008. URL http://dl.acm.org/citation.cfm?id=1373322. (Cited on pages 14 and 58)

[BKRS12] T. Babiak, M. Kretínský, V. Rehák, J. Strejcek. LTL to Büchi Automata Translation: Fast and More Deterministic. *CoRR*, abs/1201.0682, 2012. URL http://arxiv.org/abs/1201.0682. (Cited on pages 10, 15 and 37)

[Bly99] J. Blythe. Decision-theoretic Planning. *AI Magazine*, 20(2), 1999. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.26.6568. (Cited on pages 15 and 16)

[BM08] J. A. Bondy, U. S. R. Murty. *Graph Theory*. Graduate Texts in Mathematics, Vol. 244. Springer, 2008. URL http://www.springer.com/new+%26+forthcoming+titles+%28default%29/book/978-1-84628-969-9?cm_mmc=Google-_-Book%20Search-_-Springer-_-0. (Cited on page 14)

[CGP99] E. M. Clarke, O. Grumberg, D. A. Peled. *Model Checking*. The MIT Press, 1999. URL http://www.worldcat.org/isbn/0262032708. (Cited on pages 10, 14 and 58)

[Cou99] J.-M. Couvreur. On-the-fly Verification of Linear Temporal Logic. In J. Wing, J. Woodcock, J. Davies, editors, *FM'99 — Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pp. 253–271. Springer Berlin Heidelberg, 1999. doi:10.1007/3-540-48119-2_16. URL http://dx.doi.org/10.1007/3-540-48119-2_16. (Cited on pages 10, 15, 58, 59 and 71)

[DDHS09] M. Dabrowski, M. Drabik, P. Habela, K. Subieta. *Object-oriented Declarative Workflow Management System*. Institute of Computer Science Polish Academy of Sciences, 2009. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.156.5843&rep=rep1&type=pdf. (Cited on page 30)

[DDTS11] M. Dabrowski, M. Drabik, M. Trzaska, K. Subieta. Prototype of Object-Oriented Declarative Workflows. In N. Nguyen, C.-G. Kim, A. Janiak, editors, *Intelligent Information and Database Systems*, volume 6591 of *Lecture Notes in Computer Science*, pp. 47–56. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-20039-7_5. URL http://dx.doi.org/10.1007/978-3-642-20039-7_5. (Cited on page 30)

[DDTS12] M. Dabrowski, M. Drabik, M. Trzaska, K. Subieta. Business Process Modeling in Object-Oriented Declarative Workflow. In *ICSEA 2012, The Seventh International Conference on Software Engineering Advances*, pp. 141–146. 2012. URL http://www.thinkmind.org/index.php?view=article&articleid=icsea_2012_6_10_10231. (Cited on pages 30 and 31)

[Dec] URL http://www.win.tue.nl/declare/. (Cited on pages 30 and 36)

[DHM+96] P. Dourish, J. Holmes, A. MacLean, P. Marqvardsen, A. Zbyslaw. Freeflow: mediating between representation and action in workflow systems. In *Proceedings of the 1996 ACM conference on Computer supported cooperative work*, CSCW '96, pp. 190–198. ACM, New York, NY, USA, 1996. doi:10.1145/240080.240252. URL http://doi.acm.org/10.1145/240080.240252. (Cited on pages 30, 32 and 33)

[DLP04] A. Duret-Lutz, D. Poitrenaud. SPOT: an extensible model checking library using transition-based generalized Büchi automata. In *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004. (MASCOTS 2004). Proceedings. The IEEE Computer Society's 12th Annual International Symposium on*, pp. 76 – 83. 2004. doi:10.1109/MASCOT.2004.1348184. URL http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=1348184&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D1348184. (Cited on pages 10, 58 and 71)

[DR09] P. Dadam, M. Reichert. The ADEPT Project: A Decade of Research and Development for Robust and Flexible Process Support - Challenges and Achievements. *Computer Science - Research and Development*, 23(2):81–97, 2009. URL http://dbis.eprints.uni-ulm.de/486/. (Cited on pages 20 and 28)

[EH82] E. A. Emerson, J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, STOC '82, pp. 169–180. ACM, New York, NY, USA, 1982. doi:10.1145/800070.802190. URL http://doi.acm.org/10.1145/800070.802190. (Cited on pages 33 and 35)

[EH86] E. A. Emerson, J. Y. Halpern. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, 1986. doi:10.1145/4904.4999. URL http://doi.acm.org/10.1145/4904.4999. (Cited on pages 33, 35, 36 and 37)

[FHR11] S. Föll, K. Herrmann, K. Rothermel. PreCon: expressive context prediction using stochastic model checking. In *Proceedings of the 8th international conference on Ubiquitous intelligence and computing*, UIC'11, pp. 350–364. Springer-Verlag, Berlin, Heidelberg, 2011. URL http://dl.acm.org/citation.cfm?id=2035646.2035680. (Cited on page 58)

[FMR+09a] D. Fahland, J. Mendling, H. Reijers, B. Weber, M. Weidlich, S. Zugal. Declarative vs. Imperative Process Modeling Languages: The Issue of Maintainability. In B. Mutschler, R. Wieringa, J. Recker, editors, *1st International Workshop on Empirical Research in Business Process Management (ER-BPM'09)*, pp. 65–76. Ulm, Germany, 2009. URL http://metrik.informatik.hu-berlin.de/grk-wiki/index.php/Declarative_vs._Imperative_Process_Modeling_Languages:

_The_Issue_of_Maintainability. (LNBIP to appear). (Cited on pages 19 and 30)

[FMR+09b] D. Fahland, J. Mendling, H. Reijers, B. Weber, M. Weidlich, S. Zugal. Declarative vs. Imperative Process Modeling Languages: The Issue of Understandability. In B. Mutschler, R. Wieringa, J. Recker, editors, *1st International Workshop on Empirical Research in Business Process Management (ER-BPM'09)*, pp. 65–76. Ulm, Germany, 2009. URL http://metrik.informatik.hu-berlin.de/grk-wiki/index.php/Publications:Declarative_versus_Imperative_Process_Modeling. (LNBIP to appear). (Cited on pages 19, 20 and 30)

[GH01] D. Giannakopoulou, K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, pp. 412–416. IEEE Computer Society, Washington, DC, USA, 2001. URL http://dl.acm.org/citation.cfm?id=872023.872506. (Cited on pages 10, 37 and 58)

[GL02] D. Giannakopoulou, F. Lerda. From States to Transitions: Improving Translation of LTL Formulae to Büchi Automata. In *Proceedings of the 22nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems*, FORTE '02, pp. 308–326. Springer-Verlag, London, UK, UK, 2002. URL http://dl.acm.org/citation.cfm?id=646220.682186. (Cited on pages 10, 37, 58 and 59)

[GN90] N. Gupta, D. Nau. Optimal Block's World Solutions are NP-Hard. Technical report, Computer Science Dept., Univ. of Maryland, 1990. (Cited on page 16)

[GPVW96] R. Gerth, D. Peled, M. Y. Vardi, P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pp. 3–18. Chapman & Hall, Ltd., London, UK, UK, 1996. URL http://dl.acm.org/citation.cfm?id=645837.670574. (Cited on pages 10, 37 and 58)

[HHJ+99] P. Heinl, S. Horn, S. Jablonski, J. Neeb, K. Stein, M. Teschke. A comprehensive approach to flexibility in workflow management systems. *SIGSOFT Softw. Eng. Notes*, 24(2):79–88, 1999. doi:10.1145/295666.295675. URL http://doi.acm.org/10.1145/295666.295675. (Cited on pages 19 and 21)

[HLS+99] R. Hull, F. Llirbat, E. Siman, J. Su, G. Dong, B. Kumar, G. Zhou. Declarative workflows that support easy modification and dynamic browsing. *SIGSOFT Softw. Eng. Notes*, 24(2):69–78, 1999. doi:10.1145/295666.295674. URL http://doi.acm.org/10.1145/295666.295674. (Cited on page 30)

[HM10a] T. Hildebrandt, R. R. Mukkamala. Distributed Dynamic Condition Response Structures. In *Pre-proceedings of InternationalWorkshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES 10)*. 2010. URL http://www.itu.dk/people/rao/rao_files/dcrsplacescamredver.pdf. (Cited on page 30)

[HM10b] T. T. Hildebrandt, R. R. Mukkamala. Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs. In K. Honda, A. Mycroft, editors, *PLACES*, volume 69 of *EPTCS*, pp. 59–73. 2010. URL http://dblp.uni-trier.de/db/series/eptcs/eptcs69.html#abs-1110-4161. (Cited on page 30)

[HTD90] J. Hendler, A. Tate, M. Drummond. AI planning: systems and techniques. *AI Mag.*, 11(2):61–77, 1990. URL http://dl.acm.org/citation.cfm?id=87205.87211. (Cited on pages 15 and 16)

[JI09] H. Jamil, A. Islam. The Power of Declarative Languages: A Comparative Exposition of Scientific Workflow Design Using BioFlow and Taverna. In *Services - I, 2009 World Conference on*, pp. 322 –329. 2009. doi:10.1109/SERVICES-I.2009.46. (Cited on page 30)

[KN01] K. Katayama, H. Narihisa. Performance of simulated annealing-based heuristic for the unconstrained binary quadratic programming problem. *European Journal of Operational Research*, 134(1):103 – 119, 2001. doi:10.1016/S0377-2217(00)00242-3. URL http://www.sciencedirect.com/science/article/pii/S0377221700002423. (Cited on pages 59, 60, 61, 71, 81 and 83)

[KV83] S. Kirkpatrick, M. P. Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983. (Cited on page 59)

[LSPG06] R. Lu, S. Sadiq, V. Padmanabhan, G. Governatori. Using a temporal constraint network for business process execution. In *Proceedings of the 17th Australasian Database Conference - Volume 49*, ADC '06, pp. 157–166. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2006. URL http://dl.acm.org/citation.cfm?id=1151736.1151753. (Cited on page 30)

[LUW10] F. Leymann, T. Unger, S. Wagner. On designing a people-oriented constraint-based workflow language. 2010. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2010-13&engl=1. (Cited on pages 30, 38 and 40)

[MF02] P. Merz, B. Freisleben. Greedy and Local Search Heuristics for Unconstrained Binary Quadratic Programming. *Journal of Heuristics*, 8(2):197–213, 2002. doi:10.1023/A:1017912624016. URL http://dx.doi.org/10.1023/A%3A1017912624016. (Cited on pages 59, 60 and 71)

[MPVDAP08] N. Mulyar, M. Pesic, W. M. P. Van Der Aalst, M. Peleg. Declarative and procedural approaches for modelling clinical guidelines: addressing flexibility issues. In *Proceedings of the 2007 international conference on Business process management*, BPM'07, pp. 335–346. Springer-Verlag, Berlin, Heidelberg, 2008. URL http://dl.acm.org/citation.cfm?id=1793714.1793753. (Cited on pages 30 and 36)

[Pes08] M. Pesic. *Constraint-based Workflow Management Systems: Shifting Control to Users*. Ph.D. thesis, Eindhoven, University of Technology, 2008. URL http://repository.tue.nl/638413. (Cited on pages 19, 30, 36, 37, 38 and 58)

[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. 287–298. 2007. doi:10.1109/EDOC.2007.14. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4384001&tag=1. (Cited on pages 30 and 36)

[RD98] M. Reichert, P. Dadam. ADEPTflex-Supporting Dynamic Changes of Workflows Without Losing Control. *Journal of Intelligent Information Systems, Special Issue on Workflow Management Systems*, 10(2):93–129, 1998. URL http://dbis.eprints.uni-ulm.de/301/. (Cited on pages 20, 28 and 29)

[RF02] A. B. Raposo, H. Fuks. Defining Task Interdependencies and Coordination Mechanism for Colaborative Systems. In *Cooperative Systems Design*, pp. 88–103. 2002. URL http://130.203.133.150/showciting;jsessionid=1DA4439207A3311DFB66042CF354008C?cid=4906815. (Cited on page 30)

[RRD03] M. Reichert, S. Rinderle, P. Dadam. ADEPT Workflow Management System. In W. Aalst, M. Weske, editors, *Business Process Management*, volume 2678 of *Lecture Notes in Computer Science*, pp. 370–379. Springer Berlin Heidelberg, 2003. doi:10.1007/3-540-44895-0_25. URL http://dx.doi.org/10.1007/3-540-44895-0_25. (Cited on pages 20, 28 and 29)

[RRD04a] S. Rinderle, M. Reichert, P. Dadam. Correctness criteria for dynamic changes in workflow systems: a survey. *Data Knowl. Eng.*, 50(1):9–34, 2004. doi:10.1016/j.datak.2004.01.002. URL http://dx.doi.org/10.1016/j.datak.2004.01.002. (Cited on page 29)

[RRD04b] S. Rinderle, M. Reichert, P. Dadam. Flexible Support of Team Processes by Adaptive Workflow Systems. *Distrib. Parallel Databases*, 16(1):91–116, 2004. doi:10.1023/B:DAPD.0000026270.78463.77. URL http://dx.doi.org/10.1023/B:DAPD.0000026270.78463.77. (Cited on pages 20, 28, 29 and 58)

[RV07] K. Y. Rozier, M. Y. Vardi. LTL satisfiability checking. In *Proceedings of the 14th international SPIN conference on Model checking software*, pp. 149–167. Springer-Verlag, Berlin, Heidelberg, 2007. URL http://dl.acm.org/citation.cfm?id=1770532.1770548. (Cited on pages 10 and 58)

[RZ96] D. Riehle, H. Züllighoven. Understanding and using patterns in software development. *Theory and Practice of Object Systems*, 2(1):3–13, 1996. doi:10.1002/(SICI)1096-9942(1996)2:1\%3C3::AID-TAPO1\%3E3.0.CO;2-\%23. URL http://dx.doi.org/10.1002/(SICI)1096-9942(1996)2:1%3C3::AID-TAPO1%3E3.0.CO;2-%23. (Cited on page 22)

[SC85] A. P. Sistla, E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985. doi:10.1145/3828.3837. URL http://doi.acm.org/10.1145/3828.3837. (Cited on page 57)

[SdF03] M. Sheshagiri, M. desJardins, T. Finin. A Planner for Composing Services Described in DAML-S. In *In Proceedings of the AAMAS Workshop on Web Services and Agent-based Engineering*. 2003. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.2042. (Cited on page 15)

[SMR+08a] H. Schonenberg, R. Mans, N. Russell, N. Mulyar, W. Aalst. Process Flexibility: A Survey of Contemporary Approaches. In J. Dietz, A. Albani, J. Barjis, editors, *Advances in Enterprise Engineering I*, volume 10 of *Lecture Notes in Business Information Processing*, pp. 16–30. Springer Berlin Heidelberg, 2008. doi:10.1007/978-3-540-68644-6_2. URL http://dx.doi.org/10.1007/978-3-540-68644-6_2. (Cited on pages 19, 21, 25, 28, 29 and 30)

[SMR+08b] M. H. Schonenberg, R. S. Mans, N. C. Russell, N. A. Mulyar, W. M. P. van der Aalst. Towards a Taxonomy of Process Flexibility, 2008. URL http://130.203.133.150/viewdoc/summary?doi=10.1.1.142.8476. (Cited on pages 19, 21 and 25)

[SMRM07] M. H. Schonenberg, R. S. Mans, N. C. Russell, N. A. Mulyar. Towards a Taxonomy of Process Flexibility (Extended Version). 2007. URL http://libra.msra.cn/Publication/4804581/towards-a-taxonomy-of-process-flexibility-extended-version. (Cited on pages 19, 21 and 25)

[SPO] URL http://spot.lip6.fr. (Cited on pages 58 and 71)

[ST03] R. Sebastiani, S. Tonetta. "More Deterministic" vs. "Smaller" Büchi Automata for Efficient LTL Model Checking. In D. Geist, E. Tronci, editors, *Correct Hardware Design and Verification Methods*, volume 2860 of *Lecture Notes in Computer Science*, pp. 126–140. Springer Berlin Heidelberg, 2003. doi:10.1007/978-3-540-39724-3_12. URL http://dx.doi.org/10.1007/978-3-540-39724-3_12. (Cited on pages 10, 15 and 58)

[Thi02] X. Thirioux. Simple and Efficient Translation from LTL Formulas to Büchi Automata. *Electronic Notes in Theoretical Computer Science*, 66(2):145 – 159, 2002. doi:10.1016/S1571-0661(04)80409-2. URL http://www.sciencedirect.com/science/article/pii/S1571066104804092. FMICS'02, 7th International ERCIM Workshop in Formal Methods for Industrial Critical Systems (ICALP 2002 Satellite Workshop). (Cited on pages 10, 15 and 58)

[UELW10] T. Unger, H. Eberle, F. Leymann, S. Wagner. An Event-model for Constraint-based Person-centric Flows. In *Proceedings of the 2010 International Conference on Progress in Informatics and Computing (PIC-2010)*, pp. 927–932. IEEE, 2010. doi:10.1109/PIC.2010.5687886. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/

NCSTRL/NCSTRL_view.pl?id=INPROC-2010-101&engl=0. (Cited on pages 30 and 58)

[Wag10] S. Wagner. *A Concept of Human-oriented Workflows.* Diploma thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, 2010. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2987&engl=1. (Cited on pages 30, 38 and 40)

[WLB03] J. Wainer, F. Lima Bezerra. Constraint-Based Flexible Workflows. In J. Favela, D. Decouchant, editors, *Groupware: Design, Implementation, and Use*, volume 2806 of *Lecture Notes in Computer Science*, pp. 151–158. Springer Berlin Heidelberg, 2003. doi:10.1007/978-3-540-39850-9_13. URL http://dx.doi.org/10.1007/978-3-540-39850-9_13. (Cited on page 30)

[Wor] URL http://www.workflowpatterns.com. (Cited on pages 21, 22, 25, 28 and 29)

[WRRM08] B. Weber, M. Reichert, S. Rinderle-Ma. Change patterns and change support features - Enhancing flexibility in process-aware information systems. *Data Knowl. Eng.*, 66(3):438–466, 2008. doi:10.1016/j.datak.2008.05.001. URL http://dx.doi.org/10.1016/j.datak.2008.05.001. (Cited on pages 29 and 30)

[YAW] URL http://www.yawlfoundation.org. (Cited on pages 20 and 22)

[ZD06] H. Zhao, P. Doshi. A hierarchical framework for composing nested web processes. In *In ICSOC*. 2006. (Cited on page 15)

[ZD07] H. Zhao, P. Doshi. Haley: A Hierarchical Framework for Logical Composition of Web Services. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pp. 312 –319. 2007. doi:10.1109/ICWS.2007.95. (Cited on pages 10, 15 and 16)

[Zha09] H. Zhao. *Scalable composition of Web services under uncertainty.* Ph.D. thesis, University of Georgia, 2009. URL http://athenaeum.libs.uga.edu/handle/10724/11685. (Cited on pages 10, 15 and 16)

All links were last followed on June 4, 2013.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Ort, Datum, Unterschift