universität
wien

MASTERARBEIT

Titel der Masterarbeit

"An Architectur of a Workflow Execution Engine to Enable Network Based Execution of Dynamic Workflows"

Verfasser

Gerhard Stürmer

angestrebter akademischer Grad

Diplom Ingenieur (Dipl.-Ing.)

Wien, 2010

Studienkennzahl lt. Studienblatt:	A 066 926
Studienrichtung lt. Studienbuchblatt:	Masterstudium Wirtschaftsinformatik
Betreuer:	Ao. Univ.-Prof. Univ.-Prof. Dipl.-Ing. Dr. Erich Schikuta

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die Zitate deutlich kenntlich gemacht zu haben.
Wien, den 18.Feburar 2010

Unterschrift:

_____

Gerhard Stürmer

Contents

1 Introduction **1**
- 1.1 Definition of Terms . . . 2
- 1.2 Structure of this work . . . 3

2 Related Work **5**

3 Dynamic Workflows **9**

4 Architecture **13**
- 4.1 Overview . . . 13
- 4.2 Included Handlers . . . 16
- 4.3 Handler Wrapper and Interfaces . . . 17
  - 4.3.1 The Control Flow Interface . . . 18
  - 4.3.2 The Intervention Interface . . . 19
  - 4.3.3 The Properties Interface . . . 21
- 4.4 The Workflow Execution Engine . . . 21
- 4.5 Workflow Description . . . 22

5 Implementation **23**
- 5.1 Domain Specific Language . . . 24
- 5.2 Start/Stop/Continue . . . 26
- 5.3 Example Realization . . . 26
- 5.4 Evaluation . . . 28
  - 5.4.1 Basic Control Flow Patterns . . . 29
  - 5.4.2 Advanced Branching and Synchronization Patterns . . . 30
  - 5.4.3 State Based Patterns . . . 33
  - 5.4.4 Multiple Instances . . . 36
  - 5.4.5 Cancellation and Force Completion . . . 38
  - 5.4.6 Termination and Triggers . . . 39
  - 5.4.7 Iterations . . . 41
  - 5.4.8 Comparison to Other Engines . . . 41
- 5.5 Technical Realization . . . 45
  - 5.5.1 Basics . . . 46
  - 5.5.2 Applied Metaprogramming . . . 47
  - 5.5.3 Elements from the Workflow perspective . . . 50
  - 5.5.4 Elements from the Controllers perspective . . . 54
  - 5.5.5 Search procedure . . . 55

6 Workflow Execution in a cloud environment **57**
6.1 Dist-WEE 58
6.1.1 Overview 58
6.1.2 Resources and Interface mappings 59
6.1.3 User Interface 63

7 Conclusion **67**
7.1 Contribution of this work 68
7.2 Further research 68

A Abstract **79**

B Kurzzusammenfassung **81**

C Curriculum Vitae **83**

Chapter 1

Introduction [1]

Self-healing behavior [1], SLA compliance (like checking temporal constraints at runtime) or novel repair algorithms [2] often require the modification of existing workflow engines. Workflow engines usually come with a set of different APIs including:

- The Business API that allows access to a workflow specific set of operations to realize functionalities invoked by the workflow engine.
- The workflow specific Engine API that allows to intervene with running workflows and change the behavior of the engine.

These workflow engine specific APIs are not very flexible, as the APIs are often limited to a set of operations either restricted by the functionality of the workflow engine or even shaped by the feature set of the workflow description language (e.g. WS-BPEL [3]). The aim of this thesis is to overcome these shortcomings by applying service oriented concepts to the architecture of workflow engine.

Modularizing the workflow engine has the goal to standardize ways of interaction between several aspects of a workflow engine:

(A) The workflow execution engine handling the control flow.

(B) The execution of external activities which for example may be SOAP or RESTful services.

(C) The environment of the execution consisting of execution context (variables) and endpoints as well as the state of a workflow.

Splitting up the workflow engine into three pieces covering the above mentioned aspects and making them accessible through well-defined APIs allows to create independent services for logging and repair which are no longer fixed parts of the workflow engine but loosely connected.

We envision workflow execution as the interaction of highly independent modules, each of them covering a certain execution aspect. In order to create a workflow execution engine that enables to carry out a dynamic workflow execution we specify the following design rationale:

[1] This section is a modified and extended version of the section "Introduction" in: G. Stürmer, J. Mangler, and E. Schikuta, "Building a Modular Service Oriented Workflow Engine", IEEE International Conference on Service-Oriented Computing and Applications (SOCA'09) December 14-15, 2009, Taipei, Taiwan.

- The core of such an interaction is the workflow execution engine.
- The execution engine features a minimal set of operations to cover custom control flow patterns.
- The execution engine is not responsible for handling interactions with external services, monitoring, security, and repair strategies other than exceptions which are part of the workflow description. Each of these responsibilities is delegated to arbitrary external modules through a unified interface. These external modules are further called handlers.
- The execution engine is able to be started, stopped and continued.
- While the execution engine is stopped
  - the thread of control is modifiable,
  - the workflow structure is freely changeable (add/replace/remove activities, change control structures), and
  - the workflow context is modifiable (e.g. to fit a different thread of control).
- It is possible to replace handlers on the fly.
- The handlers themselves are able to trigger all functionalities mentioned above, and therefore are able to intervene with the control flow via the execution engine.

The flexibility gained through realizing the requirements allows us to create a loosely coupled system consisting of the workflow execution engine and the handlers, that enables to use runtime-information to dynamically change the workflow in the above mentioned manners.

In this thesis we present the prototype of the fully dynamic Workflow Execution Engine (WEE) that enables us a level of interaction with running workflows that is not possible with established workflow engines now [4] [5]. We show that WEE supports workflow patterns up to a level that is not reached by many existing workflow engines, with the addition of a degree of flexibility that fosters the creation of dynamic workflows. As highly dynamic business environments (e.g. the "Cloud") and Service Oriented Architecture (SOAs) became a commodity we conclude that workflow engines have to be modularized and gain more flexibility in order to assist the creation of dynamic business processes.

1.1 Definition of Terms

Before we can continue with how to modularize a workflow engine we have to define some essential terms common in literature.

We define a *workflow* as a behavioral description that combines several external services (e.g. webservices described by a WSDL).

A *workflow description* is commonly understood to consists of a set of control structures and activities that possibly call the above mentioned external services. When such a description is executed it results in a workflow instance that holds the runtime information and associated data which is further referred to as *workflow context*.

We further define the set of control structures and activities in a running workflow instance as *workflow structure*.

Activities are defined as the sum of information associated with the execution of an external service (e.g. state, return values).

For a running workflow instance, the *thread of control* is a sequence of activities constrained by control structures.

A *workflow engines* copes with all different aspects of workflow enactment, control flow, data, exception handling, resources, logging, security and others.

A *workflow execution engine* is a part of a workflow engine that only deals with the control flow aspect including the invocation of external services.

1.2 Structure of this work

Section 2 gives an overview about related work that refers to this field of workflows and presents different approaches that have been taken by scientists.

In Section 3 we will further elaborate on the desired properties of dynamic workflows. We will investigate problems actual workflows have and formulate our motivation for dynamic workflows. Possible applications and benefits conclude this section.

We build on this analysis to identify a possible architecture in Section 4. We propose an layered system which covers the requirements of dynamic workflows and enables a network based execution. We will discuss possible fields of applications for different handler types and how they interact with the workflow execution engine. A definition of the interfaces between the different layers and the responsibilities of each layer is part of this section.

How we put the proposed architecture into practice is explained in Section 5. We introduce our prototype implementation of the workflow execution engine including a newly designed domain specific language in order to describe the workflow structure. The elements of the domain specific language are described and an example realization will illustrate how our prototype implementation enables the different mechanisms of dynamic workflows. In an evaluation part of this section, we will compare the feature set of our workflow execution engine to standardized workflow patterns and other workflow engines. We conclude the implementation part with a detailed explanation of the metaprogramming facilities we used to express the domain specific language. This explanation of the realization also includes the technical setup we used for our workflow execution engine.

Workflow execution in a cloud environment is addressed in Section 6. We will elaborate how our architecture fosters the integration of different handlers from the cloud to embed them as replaceable services. We will also introduce Dist-WEE, a RESTful interface for our prototype of a workflow execution engine. An exemplary user interface demonstrates how this RESTful interface utilizes the WEE in a service oriented architecture.

The conclusion in Section 7 summarizes the acquired results and determines the contribution of this thesis. An outlook on future research provides incentives to employ the characteristics of our newly developed workflow execution engine.

Chapter 2

Related Work

Workflow systems focus on the execution of defined processes. They orchestrate activities and control structures (like conditions or loops) which are usually defined in prior. In a business environment the workflow descriptions, containing these activities and control structures, are derived from business processes [6]. As real world business processes can change over time or must be adapted ad-hoc [7],[8], the need to map this adoption to the workflows was identified. In the field of dynamic changes of workflows, Ellis et al. [9] created a formal definition and provided a mathematical approach to analyze the problem. They focused on evolution of workflows, which deals with the problem of applying a changed process description to already running workflow instances. Basically, three options are possible:

1. Complete the workflow instance according to the original process description
2. Abort and optionally enact a new workflow instance with the changed process description
3. Transform/Migrate the workflow instance to the new process description

As options one and two may have undesirable effects (especially considering long running workflow instances in case one) the third option often is feasible but difficult to handle. Ellis therefore defined different regions (touched/untouched) within a workflow to express at which position workflow instances can be easily migrated to a changed process description. The "dynamic change bug" raised in this work often refers to the problem that the correctness criteria for workflows may be violated when changing the workflow description dynamically [10], [11]. The correctness criteria describes that a changed workflow instance must not result in an inconsistent state because of the changes. This becomes true if the changes include impossible arrangements in the control flow (like placing a join without having a prior split) but can also result from situation that lead into deadlock situations. The term "Adaptive Workflow" was introduced to describe the field of changes to workflow instances. Van der Aalst et al. [12] classified two different types of change:

Structural changes which refer to the workflow evolution as described above.

Individual changes which only effects the workflow instances that is changed. Individual changes (also referred to as ad-hoc changes) can be further divided into entry time changes and on-the-fly changes. While entry time changes are applied when a case is not yet in the system, the on-the-fly changes are applied to a workflow instance that is

Table 2.1: Types of Exceptions by [13] and their impact to workflows

Category	**Impact on workflow**
Noise	none
idiosyncratic	individual change
evolutionary	structural change

already in the execution process of the workflow system.

Structural and individual changes can occur at the same time and influence each other. An individual change may be proved to be a good change and influence the overall process description (therefore becoming a structural change). Also, structural changes can result in the transformation of running workflow instances by using the same techniques as needed for individual changes.

While this categorization focuses on the effect for the workflow instances, Kammer et al. [13] made a classification by the reason of the change. In general, exceptions are seen as one of the main sources for the need of dynamic changes. Exceptions are described as abnormality or a deviance from the defined model. Exception situations prevent a case to be continued without to fail and therefore require an intervention. This intervention can be manually or automatically by a specific algorithm.

Along to the categorization of Van der Aalst et al. mentioned above, Kammer et al. identifies three different types of exceptions as can be seen in Table 2.1. Noise exceptions can occur in workflow execution but do not affect the further execution of the workflow and can therefore be ignored or tolerated without taking actions. Idiosyncratic exceptions affect only the workflow instances where they occur but they do not need the overall process model to be changed as it is still valid for new or other running instances. The occurrence of evolutionary exceptions are proving that a defined process description is not able to reach the specified goal at all. Therefore they are a reason to change the process description and apply structural changes.

Kammer et al. further investigates the possibilities of adaptive workflows by proposing on-the-fly workflow composition. In this case, the process description is mainly not available in prior but can be built further while in execution. The process description therefore has to be further enlarged or created with the information that is coming while the workflow instance is being executed. The appending of the process description can be done from an external source or by the workflow itself (which is referred to as reflexivity).

A persistent theme in adaptive workflows are security aspects. Domingos et al. [14] modified and extended the role-based access control model for the use in adaptive workflow systems. Traditional workflow management systems implement access control aspects mainly when it comes to task assignment. The role model considers who is allowed to perform or delegate which tasks. In an adaptive workflow environment, the model has to include which changes are allowed, where they are allowed and who is allowed to perform them. Domingos et al. extends the role-based access control model along these directions: Authorizations considering the objects (like instances, activities, etc.) and authorization considering the adaption of the authorization itself. The first direction addresses the impact of the adaption (*what* is changed), whereas the second direction deals with the consequence of such a change. A

changed workflow description often needs to adapt the role model which reflects the access rights for the changed workflow description. Adaptive workflows therefore have to consider not only changes to workflows, but also to the access control model.

A great deal is being written about different approaches for implementations of adaptive workflow systems. Reichert et al. [15] presented ADEPTflex, a system that provides a large set of change operations like task insertion, task deletion or task sequence changes. The focus is mainly on ensuring correctness and consistency while these operations are executed. However, the possible change operations are limited to a fixed set.

The idea to utilize multiagent systems for adaptive workflows has been developed by Buhler et al. [16]. They propose that a workflow description language like BPEL4WS can be used to formulate a setup for a multiagent system that can react customized to unexpected environment situations. Adaptive workflow engines can be expressed as the combination of webservices and agents. The agents are the coordinating actors of webservices in this view. Agents are understood to be autonomic, situation-aware, proactive and capeable for interaction with other agents or systems. In their view, workflow execution is an act of cooperative problem soving and therefore applicable to be done by a multiagent system. A process definition represents an initial way of webservice arrangements in this system. The agents may decide to deviate from this path if they experience problems in the execution process.

Baresi et al. [17] investigated self-healing in BPEL processes and adopted therefore the ActiveBPEL [18] workflow engine. To achieve the self-healing behavior, the system must be able to detect faults immediately and limit their impacts. A recovery logic then has to provide an adequate compensating strategy. The authors developed a supervision framework for BPEL processes called Dynamo and define pre- and post-conditions with JBoss Rules [19] to express the supervision rules. Each rule contains a location (= what is supervised), monitoring parameters (customize the degree of supervision), a monitoring expression (= constraint that triggers the recovery) and a recovery strategy (= actions that should be taken). The recovery strategies can take several actions like stop execution, call an external service, substitute the activity, retry a service call or change an endpoint. However, the recovery strategy does not perform major adaptations to the workflow description. The authors evaluate their approach also on a case study.

Chapter 3

Dynamic Workflows [1]

We define dynamic workflows as the possibility to intervene into the control flow and the context of a workflow at runtime. This includes minor changes like modifying an endpoint or context variable but also the replacement of a whole workflow.

To explain the benefits of dynamic workflows, we start with the following problems:

Diluted Business Code. When designing a workflow, the designer has to consider a multitude of possible errors. E.g. when services become unavailable a valid compensation strategy has to be in place. The multitude of error handling additions has to be seen as technical or administrative overhead which dilutes the original aim of the workflow [20] and makes it more difficult to read and maintain the workflow description in respect to the underlying business process. Therefore, it has to be possible to handle certain errors by defining independent and separated strategies and correct them when they occur, e.g. by modifying the endpoint and redoing the execution.

Builtin Repair Strategies. Existing Workflow Engines define their own repair strategies or at least provide a small set of possible options to carry out repair actions [6] (see efforts taken in [4] or [5]). These strategies are built directly into the workflow engines. They can be neither changed nor is it easy to include repair strategies according to events at runtime. Dynamic workflows have to not only allow for minor changes to the context but also for injecting compensating activities or even the complete replacement of the running workflow. However, it is not the responsibility of the execution engine to provide predefined ways to repair a failed workflow. A repair strategy should be adoptable to the reason of failure and may vary from case to case even within one flow of execution.

Dynamic workflow concepts have the goal to manage issues that cannot be predicted at design time. As the requirements are not fully available at design time or the situation can change in a way not foreseen at design time. So it is necessary to allow modifications which move the already running workflow back into a clean state. The decision how a broken workflow has to be repaired or enhanced cannot be made in advance. Therefore it may be either made by a human or a specialized program. Even if some of the derived problems can be solved

[1] This section is a modified and extended version of the section "The concept of dynamic workflows" in: G. Stürmer, J. Mangler, and E. Schikuta, "A domain specific language and workflow execution engine to enable dynamic workflows," in Proceedings of the 1st International Workshop on Workflow Management in Service and Cloud Computing, IEEE Computer Society Press, 2009.

by placing the solution into the workflow description, this is often not advisable as it again dilutes the business code. Consider an example of dynamic service selection: A workflow has to be executed within a given time constraint. The creator of the workflow description wants to focus on the goal of the workflow (arrangement of activities), having the constraint in mind when selecting the initial service composition. However, it should not be the responsibility of the creator to check for the constraint after each activity and possibly replace further invoked services. A simple series of activities then becomes bloated up with service selection algorithms and structures. Instead, a separated module attached to the workflow engine has to supervise the execution and initiates a repair engine when the constraint is going to be violated. The repair engine is then able to make the service selection based on the actual information of the execution process and the environment.

As a consequence of the above mentioned issues it should be the sole responsibility of an execution engine to provide flexible mechanisms for external modules to intervene into the execution. Repair strategies themselves should be rather realized as external modules and associated with given workflows at runtime by providing proper access to runtime information of the execution engine. To solve the above mentioned problems, we propose improvements in the following areas:

Modifying attributes. Changes to the workflow context represent the least invasive modification. Two applications are imaginable: the modification of endpoints and context variables. The possibility to change endpoints allows to replace external services on-the-fly. The service selection can be overruled during the course of the execution due to results already gathered. Constraints (e.g. SLAs) can be fulfilled more easier as the fulfillment can be exerted not only at design time but also at runtime. It is possible to decide for less expensive services if better progress is made as expected at design time, or to use more expensive services if progress is less than expected. With the ability to change context variables, it is possible to correct wrong results or influence the control flow. This includes dynamic control of decisions or loops like the insertion of additional iterations until data quality is sufficient (e.g. in machine learning applications). Both modifications, endpoints and context variables, may depend on progress as well as runtime parameters not available at design-time.

Proactive injection of activities. An extra activity (or block of activities) has to be added to the workflow due to runtime reasons. This can be necessary whenever the successful execution of the workflow is at risk and can be avoided through additional tasks. An example is an additional testing phase in a project.

Workflow restructuring. This intervention has potentially the most severe impact on a workflow. The main purpose for this area is motivated in enabling unanticipated repair steps. Unanticipated repair steps are necessary whenever a workflow is going to fail and a simple addition or substitution of activities is not sufficient. This becomes true if those actions have side-effects that need to be repaired or avoided as well and is also necessary whenever a clean state cannot be reached without major modifications. Unanticipated repair steps include the substitution of parts, or the complete replacement of the actual workflow.

Thread-of-Control modification. When restructuring parts of a workflow, it is often necessary to set a specific point to start execution from. But also without the need for workflow restructuring, this tool is considerable powerful. Modifying the thread of

control allows:

- Skip an activity by setting the thread of control after the activity.
- Redo an activity by setting the thread of control at the activity.
- Correct a wrong decision by setting the thread of control to an alternative branch, or set the thread of control to the decision itself and repeat it.
- Control a loop by setting the thread of control to the beginning of the loop body.

Preemptive Problem Handling. A workflow execution engine has to react proactive on problems before they escalate. Possible applications include temporal conformance checking [21] and other issues related to Quality of Service (QoS) or Service Level Agreements (SLAs) [22]. An external monitor should be responsible for checking the compliance of certain constraints. Whenever it detects possible problems, it may decide to choose new services and even restructure the original workflow.

As stated by Rinderle et al. [10] and others, interventions can occur either at the level of workflow description (which is the focus of research activities in the field of workflow evolution) or at the instance level [23, 15]. The interventions covered by WEE solely affect the *current instance* of the workflow. Therefore the workflow description is still valid and the base for subsequent instances of this workflow. The modifications cannot be applied to other instances, because they depend on the runtime attributes of the workflow instance. So when dealing with the three fundamental issues of workflow changes as summarized by Rinderle et al. [10] which are completeness, correctness, and change realization, only completeness is relevant for our approach. As we propose a workflow execution engine, completeness has to be validated by the controlling external modules, whereas the execution engine checks syntactic correctness only.

Chapter 4

Architecture [1]

In this section we illustrate the architecture of our prototype implementation. We describe the different layers and the interactions between them. Also, we define the requirements of handlers and introduce different categories. We also show how a network of different handlers can be integrated into a workflow execution engine to foster the modularization of workflow execution.

4.1 Overview

Fig. 4.1 shows the architecture of the prototype implementation of the modular Workflow Execution Engine (WEE). The WEE is provided with a workflow description that has to be enacted. When the workflow is executed, the workflow environment contains additional information for the context (e.g. variables), the current state of execution (e.g. running, finished, stopped) and the defined endpoints.

WEEs sole purpose is to take care of the control flow, it instantiates a given workflow description, but it *DOES NOT*:

- Execute web services defined in the workflow description.
- Log the thread of control.
- Check in any way the semantic correctness of the enacted workflow description.
- Manage any resources used during the workflow enactment.

What it *DOES* is including a handler wrapper that in turn delegates the tasks to services, which we further refer to as handlers. The WEE interprets the given workflow description and takes care of the control aspects of the workflow execution. It only stores information about the context variables, the current state of the workflow execution (ready, running, finished, stopped) and the possible endpoints. The workflow description consists of a set of control structures which are executed by the WEE. Loops, decisions and other control structures

[1] This section is a modified and extended version of the section "A Novel Architecture For Modular Workflow Engines" in: G. Stürmer, J. Mangler, and E. Schikuta, "Building a Modular Service Oriented Workflow Engine", IEEE International Conference on Service-Oriented Computing and Applications (SOCA'09) December 14-15, 2009, Taipei, Taiwan.

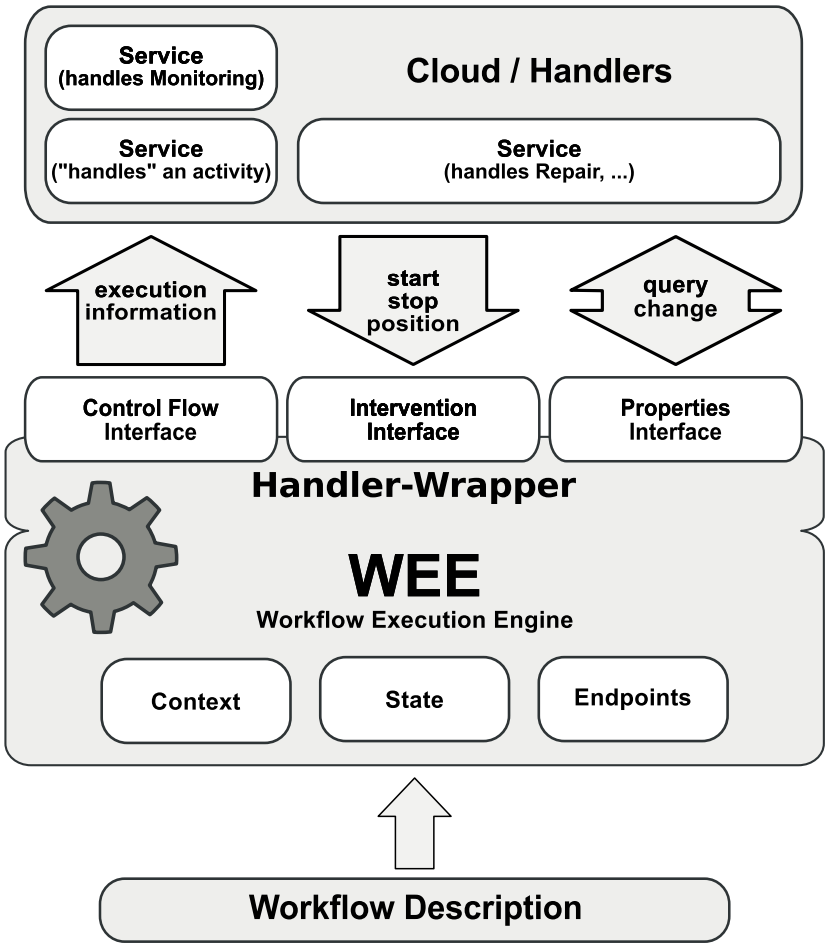

Figure 4.1: The WEE Architecture

regarding the control flow are executed by the WEE. Every other aspect, like service calls, administrative tasks etc., is forwarded to the handler wrapper which in turn includes handlers to perform tasks. Therefore, a handler is a kind of a servant to the execution engine which runs the parts of workflow that are not in the core competence of the execution engine.

But the relationship between handler and execution engine can be twofold:

Supervision & Intervention. A handler can act as a supervisor of the execution engine, getting informed about the process of execution, or intervene in cases when repair is needed (e.g. when SLAs are violated). In this case the handler has to adhere to a protocol defined by the handler wrapper. This protocol is inherent to the handler wrapper and not part of the workflow description.

Execution. The workflow description holds information how to call external applications / services. This information includes location, operation name and parameters. The handler wrapper invokes the service and hands the results back to control flow. How to invoke the service (whether it is a SOAP service, a Java class or a Phone call) is inherently implemented by the handler wrapper, the invoked service itself does not need to know anything. In constrast to *Supervision & Intervention* the protocol how to interact with the service, what to make of the return value, is part of the workflow description.

We want to explicitly point out that the above made distinction is NOT a way to differentiate between handler which (for example) monitor or repair the workflow, and services that are invoked as activities as part of the workflow description. They are both called handlers because they handle specific task in the context of the workflow, and they are both connected through the wrapper handler with the workflow.

There is no restriction for a maximum amount of different handlers that can be included into the execution process. To interact with the handlers, the WEE solely relies on the handler wrapper. It is the responsibility of the handler wrapper to embed / use the different handlers. In the process of execution, it is possible to replace a handler or queue multiple handlers for a single purpose.

With WEE focusing solely on the control flow and the integration of different handlers through the handler wrapper, a network of modules is instructed to process the execution. Each module has a clearly specified role and can be replaced by different versions, thus fostering a loose coupling and a network-based execution.

The functionality of our architecture is also depicted in Fig. 4.2 by introducing a simple flight/hotel booking example. The WEE consists of 2 parts, the workflow and the handler wrapper. Handlers are services provided for example by a cloud.

Initially the workflow consists of two activities: book a flight and book a hotel. When the first activity is carried out the WEE hands over the information about the first activity to the handler wrapper which in turn calls the booking routine on the Aeroflot homepage AND informs the monitor about the progress. The result of the call to the Aeroflot homepage is then handed back through the handler wrapper to the workflow.

For the next activity, the WEE again gives information to the handler wrapper, which first tries to book a room in the Ibis hotel. A monitor recognizes that the service call fails because the hotel is already booked out. A repair handler replaces the service call and alternatively

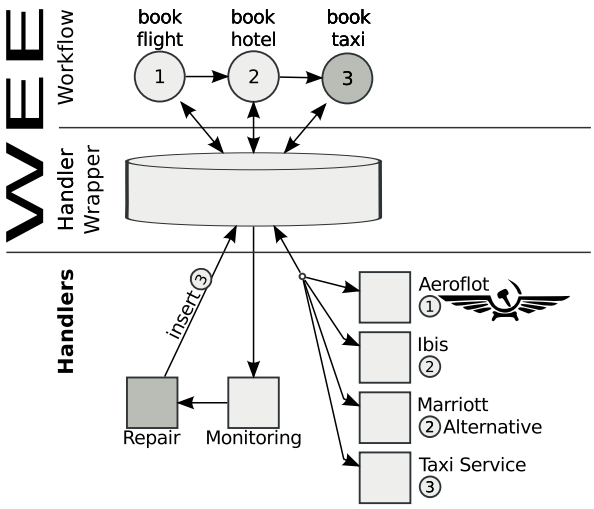

Figure 4.2: A Simple Flight / Hotel Booking Example

the room is booked in the Marriott hotel.

The monitor checks the situation and finds out that while for the Ibis hotel it is possible to walk from the airport to the hotel, for the Marriott one needs a taxi. Therefore it informs the repair handler that inserts a third activity into the workflow, which is subsequently carried out.

In the next sections, we will describe each element of the architecture in more detail.

4.2 Included Handlers

In general, responsibilities of the handlers are:

1. Supervise the process of execution (monitoring) as it is informed about each executed activity and the actual context of the workflow.
2. React on exceptions raised in the process execution (repair).
3. Deliver the results that are required by the process description (service calls).

Exercising control over the process of execution comes with enlarged responsibilities. The handlers are obligated to intervene when a workflow execution is going to fail. This may be obvious in case of exceptions or runtime errors, but is also true for other reasons like temporal conformance or quality of service.

When processing a workflow, different types of handlers may be included:

Call Handlers are services that represent that actual functionality requested by the workflow. E.g. when the workflow wants two numbers to be summed up, it delegates this task to handler wrapper, which in turn calls the service that is intended to do so. Services are selected by passing information from the workflow to the handler wrapper. Information about call handlers (which provide functionality in the form of activities to the workflow) is part of the workflow description.

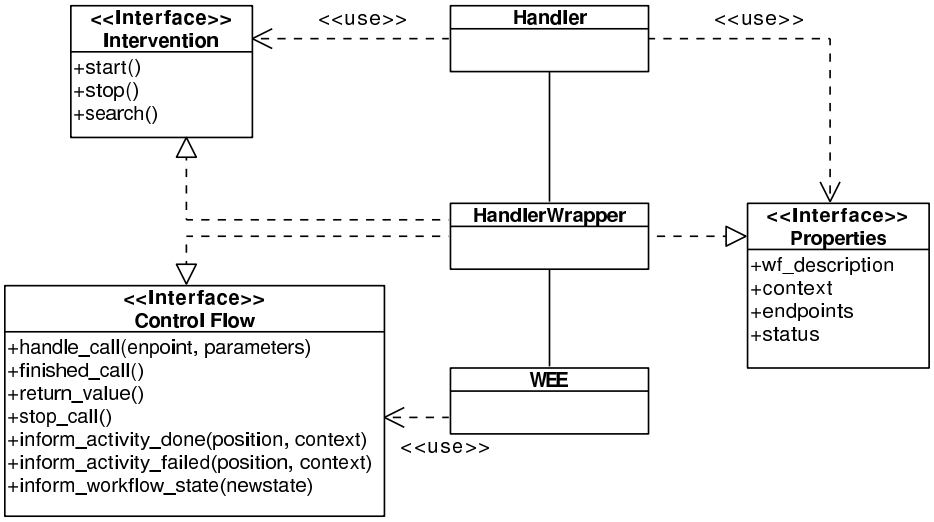

Figure 4.3: WEE Interfaces (class diagram)

Monitoring Handlers are informed about the process of execution. Logging functionality is one of the main applications of these handlers. They also supervise the execution to comply with Quality of Service (QoS) properties or Service Level Agreements (SLAs). If problems arise or are anticipated, the handler may delegate the problem-solving to a repair handler and provide the necessary information [24].

Repair Handlers are invoked to put a failing workflow back into a clean state. For this purpose, they use the intervention interface of the WEE and apply specific repair strategies according to the problem.

Security Handlers supervise the workflow execution in terms of security aspects. They grant or deny access to resources or services.

4.3 Handler Wrapper and Interfaces

As depicted in Fig. 4.1, the interaction between handlers and the WEE is enabled through the handler wrapper. Each instance of the workflow execution engine is able to have exactly one handler wrapper which has to take care about aspects which are not covered by the WEE. Which handler wrapper the WEE has to use is specified by the workflow description. Different workflows may have the need for different handler wrappers which enable functionality needed in this workflow. Some workflows include asynchronous service calls, others synchronous. Some workflows are calling webservices by a WSDL description, others may include RESTful services. A Handler wrapper can try to cover the functionality in general but the possibility to provide specialized handler wrapper for specific needs fosters the modularity of the overall workflow execution. Still, the focus in developing a handler wrapper has to stay in the ability to include handlers and delegate tasks to them.

Therefore, we propose that a handler wrapper has three interfaces:

- The **Control Flow Interface** uses execution information to trigger external services (activities, monitors).

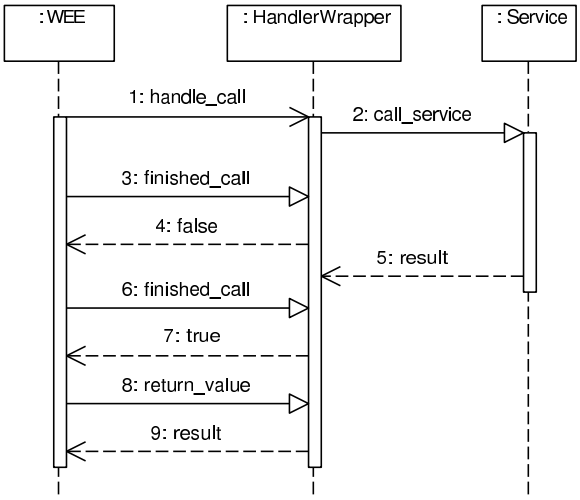

Figure 4.4: A generic example of a service call

- The **Intervention Interface** allows handlers to start, stop and set the thread of control.
- The **Properties Interface** to query information about the workflow instance.

The Interfaces are implemented by the handler wrapper. The handlers and the WEE interact solely through the handler wrapper which in turn delegates the operation to the correct receiver or executes it directly if necessary. In the curse of this section, we take a closer look on the functions the interfaces provide. Fig. 4.3 shows the available operations.

4.3.1 The Control Flow Interface

The Control Flow Interface addresses two different use cases:

Monitoring the execution process. The WEE informs the handler wrapper if an activity was executed normally or failed by invoking the methods *inform_activity_done* or *inform_activity_failed*. If the status of the workflow changes (e.g. when it is stopped) this leads to the invocation of the method *inform_workflow_status*. It is the responsibility of the handler wrapper that implements this methods and forward this information to attached monitoring facilities.

Invoking Activities As already mentioned before, the WEE does not execute service calls itself. The execution information is forwarded to the handler wrapper which in turn performs the service call. As depicted in Fig. 4.4 the WEE invokes *handle_call* and provides the endpoint of the service as well as optional parameters. The handler wrapper calls the requested service and waits for a response. Meanwhile the WEE calls *finished_call* of the handler wrapper until it returns *true*. Then, the WEE asks the handler wrapper for the result by calling *return_value*. Therefore it is possible for the handler wrapper to realize the invocation of synchronous as well as asynchronous activities:

- In the case of a synchronous activity (normal method call, request/response) the

handler wrapper opens a new thread for the call to the activity. When the method returns the result is saved and *finished_call* is allowed to return true.

– In the case of an asynchronous activity, the activity returns immediately (request/response). The activity then carries out the work and later contacts the handler wrapper to deliver the result (solicit response). In this case the handler wrapper has to implement means for the activity to return the value, after which *finished_call* is allowed to return *true* and the WEE can access the result through *return_value*.

For the WEE both ways are the same, synchronous/asynchronous *services* are merely a technical detail, asynchronous *behavior* for the WEE is a property of the workflow description. Its the responsibility of the handler wrapper to call and collect data and the responsibility of the WEE to process the workflow description,

As a consequence of the *handle_call* - *finished_call* - *return_value* schema the WEE has the possibility to inform the handler wrapper to stop a call before it finishes by calling the *stop_call*-method. The decision if the call can be stopped or has to finish is made by the handler wrapper. A *stop* is never enforced by the WEE as it cannot predict the costs of a service call abortion.

4.3.2 The Intervention Interface

It allows to *start* and *stop* the workflow execution as well as change the thread of control through *search*. The execution can be started when:

- The workflow instance is initiated (having the state *ready*).
- The workflow instance has been stopped (having the state *stopped*) and the thread of control is set to a valid start position. By default, the workflow execution will restart the execution at the same point as it has started the last time it was started.
- The workflow instance has finished execution (having the state *finished*). The workflow instance will the start over again at the position is has been started last time if the thread of control is not set to another position.

The context of a workflow instance will not be changed or reset if a workflow instance is started. This means that a workflow which is started after being stopped continues with the same context as it had when it was stopped.

When the execution is stopped, the handler wrapper (which may have active service calls) is ordered to stop the call. It is in the responsibility of the handler wrapper to decide if the call can be aborted without sanctions or if it is less expensive to follow another strategy (i.e. call a compensating service). However, the result of the handler wrapper will not be integrated into the workflow if the workflow instance is stopped. But the handler wrapper has the possibility to provide a piece of information (further referred to as *passthrough*) which indicates the state of the service call. When the handler wrapper has finished, the WEE stops the execution and provides the controller of the workflow instance with the actual context, the position identifiers where the execution has been stopped and the *passthrough* value.

The thread of control can be set to any activity within the workflow execution. Therefore, each activity has to be specified by a position identifier. As a workflow description can have

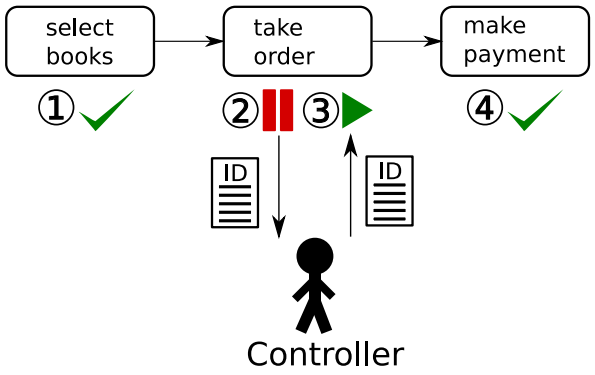

Figure 4.5: Example usage of the passthrough value

parallel branches, the possibility to set multiple start positions have to be provided. For each parallel branch, a starting position can be set. In addition, the thread of control can be set *at* the activity (meaning this activity has to be executed) or *after* the activity. This is necessary to express that an activity does not have to be executed but it may be necessary to wait at this point (e.g. for other parallel branches to be finished).

The aim of the *passthrough*-concept can be explained on an book ordering example outlined in Fig. 4.5. The workflow has 3 activities: *Select books*, *Confirm order* and *Make payment*. A user starts the workflow and defines books that should be added to the order. Then the user orders to execute the *Confirm order*-activity. While the service call to the book provider is processed, the user decides to stop the execution for some reason. According to the stop procedure, the handler wrapper (which is performing the service call) is ordered to stop the call by the WEE. The handler wrapper can ask the service provider for an ID of the shopping cart because the workflow may be continued in the future. The handler wrapper is now able to abort the service call and returns the ID as the *passthrough*-value to the WEE which forwards the information to the controller of the workflow. If the user decides to continue the workflow, the thread of control is set to the aborted activity *Confirm order*. The workflow instance is then started again and the WEE orders the handler wrapper to perform the service call to the provider. When setting the thread of control, the controller of the workflow can provide the *passthrough*-value again to the handler wrapper through the WEE. The handler wrapper is now able to continue the execution by providing the ID of the shopping cart to the service provider.

The same result would have been achieved by making the shopping cart ID explicit in the workflow description as an context variable. The service provider must then be asked for an ID *before* the service call resulting from the *Confirm order* activity can be executed. Otherwise, the handler wrapper is unable to distinguish if a service call has to be continued or if it is the first service call. The problem with this approach is that the ID is not necessary in the original workflow description and is only needed to deal with the situation that arises from the possibility to stop and continue the workflow execution. The concept of the *passthrough*-value solves these conflicts. It enables to let the workflow description stay untouched and moves the problem to the actual service call. The handler wrapper in turn is able to interpret the value as the handler wrapper itself was the creator of it.

4.3.3 The Properties Interface

External services can access attributes of the WEE and the workflow execution through the properties interface. They can read and change the workflow execution context (i. e. variables) as well as defined endpoints. Also, they can read the actual status of execution (which can be *ready/running/stopped/finished*). In addition, the workflow description can be changed, e.g. in case of repair, while the execution is not in state *running*. The given values are not check for syntactically or semantically correctness. Therefore, the user of the properties interface is responsible to set correct values.

4.4 The Workflow Execution Engine

The WEE is provided with an initial workflow description. When the execution is started by the controller of the workflow, a new instance is created. The WEE deals with the control flow aspect of such a running workflow instance and stores information that is associated with the workflow instance. These informations include:

Context. Initial information and information that is generated during the workflow execution is stored in the context. The context consists of a set of variables used by the workflow and can be input parameters for services or stored results from services.

Endpoints. A workflow orchastrates a set of services. The location of a service is specified in an endpoint. Endpoints usually provided as URIs but are not restricted to this format and can be arbitrary strings.

State. The state of the workflow instance can be *ready*, *running*, *stopped* or *finished*. Initially, the the workflow instance is *read* if the workflow instance can be started for the first time. While the workflow instance is executed, the state is set to *running*. If the execution is completed normally, the state is set to *finished*. While the state is *running*, the controller of the workflow instance can stop the execution by sending the stop-signal. After the execution is halted, the state is set to *stopped*.

The controller of the workflow instance interacts with the WEE through the handler wrapper. The WEE acts on behalf of the controller and interprets the workflow description.

Each time the WEE executes an element of the workflow description, the following options are possible:

- A *complex activity* incorporates a service call and is not executed directly by the WEE. The WEE gathers the necessary information for the service call (service location, parameters, etc.) and forwards it to the handler wrapper. The implementation of the service call (i.e. technical requirements) is not provided by the WEE, therefore the actual service call is delegated. After the service call has finished, the WEE integrates the result of the service call into the workflow. The logic how to integrate the result is provided by the workflow description (e.g. how to parse a XML-structured result value).
- A *simple activity* is directly executed within the WEE. This mechanism is provided due to performance and simplicity reasons. Activities like simple aritmethic calculations or value assignments do not need to result in a (possibly costly) service call.

- A *decision* is made according to a given condition. The subsequent branch is only executed if the condition is validated to be true.
- A *loop* repetitive executes the subsequent branch as long as a given condition is validated to be true.
- *Parallel threads* are forked. They run concurrent and merge at specified points during the course of execution. A merge can be initiated when either all or a specified amount of parallel branches finished execution.
- *Context and endpoint manipulations* are performed by the WEE as the management of these information elements is done directly within the WEE.

Monitoring aspects are not incorporated in the WEE. Therefore, the WEE only informs the handler wrapper about significant events during the execution of the workflow description. The handler wrapper is informed about the process of execution after each activity is completed. A position identifier, the actual context and possible exceptions are provided. The handler wrapper is also informed a priori the execution of a complex activity. In this case, endpoint and parameters are provided in addition to the position identifier. A change in the workflow execution state is also communicated to the handler wrapper.

Each time the WEE forwards information to the handler wrapper, the handler wrapper in turn can forward the information to specific monitoring handlers. If more informations are necessary, the monitoring handlers can gather them through the different interfaces already described in section 4.3.

4.5 Workflow Description

The workflow description contains a set of control structures (like conditions, loops, parallel branches) and activities. The workflow description expresses the formal and temporal dependencies between these activities. The activities in such a workflow description represent the information which is necessary to call an external service (like endpoint and parameters) and how to integrate the result of such a service call into the workflow. The workflow description represents the business logic which is transfered to a language that can be interpreted by a workflow engine. Different workflow languages, like BPEL [3], YAWL [25] or XPDL [26], have been developed due to specific needs. The proposed architecture itself does not prefer any language. For our prototype implementation we developed a simple workflow description language. The goal is to provide a small set of control structures which additionally makes it easy to introduce the dynamic workflow mechanisms provided by the WEE. A more detailed description of the implementation of the workflow description language is part of the next section.

Chapter 5

Implementation [1]

In the course of this section we will gradually describe the implementation of our components as depicted in Section 4.

A workflow description (e.g. WS-BPEL [3]) is translated to a domain specific language (DSL). Our DSL consists of a set of methods and operators, that allows to write simple, high-level programs, that can be directly executed by a host language. The functionalities of the DSL are not only restricted to those of workflow description languages. Where the workflow description languages are narrowed to design time elements only, the WEE-DSL also copes with aspects of runtime (e.g. endpoint modifications). So the WEE-DSL is an intermediate language, that other workflow description languages like WS-BPEL are translated into, plus a set of methods to interact with the description a runtime. The WEE acts like a virtual machine and executes the code in the intermediate language.

The WEE-DSL uses the programming language Ruby [27] as a host language because of the following reasons:

(1) Ruby has powerful metaprogramming facilities

(2) There are already successful examples for designing DSLs in Ruby

(3) It enables us to dynamically modify code at runtime through *instance_eval*, *class_eval* and Eigenclasses

Using an existing programming language brings further advantages. It allows to build executable intermediate code. Therefore a developer of a workflow is not only restricted to the elements designed in the WEE-DSL. These elements are only first means to provide a vocabulary and support for building up a workflow. Wherever these elements are insufficient, the designer may use proprietary elements of the Ruby programming language. However, it should stay the goal to provide a comprehensive set of DSL-elements as each proprietary use detracts the control of the execution engine. Further advantages of the executable intermediate code are the possibility to easily write generators for workflow description languages like WS-BPEL to translate them to intermediate code. One can also write the code directly

[1] This section is a modified and extended version of the section "WEE Prototype Implementation" in: G. Stürmer, J. Mangler, and E. Schikuta, "A domain specific language and workflow execution engine to enable dynamic workflows," in Proceedings of the 1st International Workshop on Workflow Management in Service and Cloud Computing, IEEE Computer Society Press, 2009.

(or modify generated code) to focus on the functionality instead of the syntax of a particular workflow description language.

The WEE and DSL are designed to be minimal, meaning that the WEE should focus on its core competence, the control flow only. Other functions are outsourced to external handlers that are provided from outside and can be easily replaced.

5.1 Domain Specific Language

To design a workflow in our prototype we implemented a set of DSL-elements that will be explained in conjunction with the example in Listing 5.1.

A flight as well as a hotel have to be booked for three people. If the payed sum for all three people exceeds 10000 credits an external entity is informed. The set of available DSL-elements is:

activity is an atomic operation and executes a specific task. We distinguish between two types of activities: *manipulate-activities* (see line 40) are simple operations that are executed within the WEE. The intention is to provide an easy way to perform calculations or context changes. *call-activities* (see line 21) are used to carry out more complex tasks which are encapsulated in any kind of service. The execution of a *call-activity* is delegated to the handler wrapper. Therefore it is irrelevant if the service is a webservice or any other kind of service. The handler wrapper is provided with the location of the service (i.e. an endpoint) and optional parameters.

parallel defines two or more parallel branches which are executed concurrently (see line 18). Each branch is executed in a separated thread but they still share the context of the workflow. We distinguish between two variants of parallel execution. First, the *wait*-variant is on hold until each branch has finished before the thread of control is passed to the subsequent branch. Second, the *nowait*-variant waits for a given amount of branches to be merged. Other branches who are not finished are informed to stop execution and are no longer executed by the WEE. The *nowait*-variant implements a kind of race between the branches to finish. It can be used to start different approaches to finish a job, but continues as soon as any result is available.

choose defines a decision in the control flow (see line 44). Multiple or none of the available *alternative*'s (guarded by a condition as shown in line 45) can be chosen. Also, an else-path is provided by the *otherwise*-keyword

cycle enables top-controlled loops (see line 17).

critical implements the critical section pattern (see line 20). A codeblock encapsulated in the critical-keyword is defined to be protected by a semaphore. In a multi-threaded environment, a critical section ensures exclusive execution. If a thread enters execution of a critical section, all other threads that want to enter this section have to wait until the first thread has exited the critical section. Each critical section is defined by a symbolic name. Forming multiple codeblocks labeled with the same symbolic name enables critical sections to span over different parts of a workflow.

In addition to the control structures we also defined keywords for special workflow purposes. These keywords are:

handler defines which handler wrapper should be included (see line 2).

endpoint defines the location of a service, i.e. an URI of a webservice (see line 4). Endpoints are provided to the handler wrapper if an external service has to be invoked.

context defines context variables of a workflow (see line 10). Each of the context variables is supervised by the execution engine.

LISTING 5.1: Example Workflow

```
1 class Workflow < Wee
2  handler MyHandler
3
4  endpoint :epAirBook => "uri:air/booking"
5  endpoint :epHotelBook => "uri:hotel/booking"
6  endpoint :epAirPay => "uri:air/payment"
7  endpoint :epHotelPay => "uri:hotel/payment"
8  endpoint :epInform => "uri:company/inform"
9
10 context :persons => 3
11 context :creditcard => "Visa_12345"
12 context :airline => nil, :hotel => nil
13 context :from => "Vienna", :to => "Prag"
14 context :sum => 0
15
16 control flow do
17  cycle(@persons > 0) do
18   parallel(:wait) do
19    parallel_branch do
20     critical(:airbooking) do
21      activity :bookFlight, :call, epAirBook, @from, @to do |id|
22       @airline = id
23      end
24      activity :payFlight, :call, epAirPay, @airline, @creditcard do |amount|
25       @sum += amount
26      end
27     end
28    end
29    parallel_branch do
30     critical(:hotelbooking) do
31      activity :bookHotel, :call, epHotelBook, @to do |id|
32       @hotel = id
33      end
34      activity :payHotel, :call, epHotelPay, @hotel, @creditcard do |amount|
35       @sum += amount
36      end
37     end
38    end
39   end
40   activity :countdown, :manipulate do
41    @persons -= 1
42   end
43  end
44  choose do
45   alternative(@sum > 10000) do
46    activity :inform, :call, epInform, @sum
47   end
48  end
49 end
50 end
```

5.2 Start/Stop/Continue

One of the main abilities of the WEE is to start the execution at a given point in the workflow description. The user of the engine is able to deliver the code that has to be executed and defines at which position within the code the execution should start. Because of parallel execution it must be possible to set multiple starting points, one starting point for each branch. In nested parallel branches, the number of starting positions increases accordingly.

To implement this function it is necessary to expand the DSL by a position identifier for each executable step. After starting the execution, the given starting positions, which can be in any branch of parallel executions and conditions, have to be found. The best way to find the positions is by running through all code paths that are possible. The thread of control runs in a "search"-mode, virtually executing the actions and taking a predefined way through the branches. If the thread matches a starting position it has to be marked as real execution thread. Basically the activities before this point are ignored. From the starting position on, the activities will be executed normally.

A similar procedure is used when the execution has to be stopped. If the handler wrapper is still executing an external service call, it is informed by a stop signal. The handler wrapper then has to decide if the call can be stopped without sanctions or if it has to be completed normally. The handler wrapper also has the possibility to return a piece of information ("passthrough") to the execution engine. In case of a continuation of the workflow this "passthrough" can be provided to the handler wrapper again. However, after the pending handler wrapper has returned, following activities are only virtually executed and therefore omitted.

5.3 Example Realization [2]

As elaborated in Section 4.3 the WEE allows for enhanced manipulation functionality to address the specific needs of Dynamic Workflows through the handler wrapper. In this section we describe how the interfaces exposed through the handler wrapper enable the realization of the example introduced in Section 4 (see Fig. 4.2).

Listing 5.2 shows a simple implementation of the control flow interface. During instantiation, a monitoring handler is included to enable logging (see line 3). The *handle_call*-operation (line 5 to line 12) is performed by the WEE to instruct the control flow interface to execute a service call. The control flow interface executes the service call in a new thread (see line 8 to line 11), thus this interface implements the synchronous variant. While the service call is not finished, the *finished_call*-operation returns *false* (line 13). When the service call returns (*finished_call* will then return *true*), the WEE asks for the return value by calling the *return_value*-operation in line 14. The *stop_call*-operation only informs the monitor (see line 16) but does not abort the service call in this implementation. The monitoring operations are only indicated in this implementation and result in a simple forward to the monitoring handler as it can be seen in line 19.

[2] This section is a modified and extended version of the section "Example Realization" in: G. Stürmer, J. Mangler, and E. Schikuta, "Building a Modular Service Oriented Workflow Engine", IEEE International Conference on Service-Oriented Computing and Applications (SOCA'09) December 14-15, 2009, Taipei, Taiwan.

LISTING 5.2: A Simple Handler Wrapper

```
1 class ControlFlow < Wee::ControlFlowWrapperBase
2   def initialize
3     @monitor = Monitor.new("http://monitor.org")
4   end
5   def handle_call(endpoint,*parameters)
6     @finished = false
7     @monitor.send("call",endpoint,parameters)
8     Thread.new do
9       @ret_val = Service.call(endpoint,parameters)
10      @finished = true
11    end
12  end
13  def finished_call; @finished; end
14  def return_value; @ret_val; end
15  def stop_call
16    @monitor.send("stopped")
17  end
18  def inform_activity_done(activity,context)
19    @monitor.send("done",activity,context)
20  end
21  # ... other methods
22 end
```

From the improvements suggested in section 3, the following cases are covered by our example:

Workflow Description Restructuring and Thread-of-Control Modification. The initial workflow description is replaced by a modified version. The new version can be a minor modification of the existing description but may also be complelty new. Manipulating the thread of control allows to specifiy at which activity the execution has to start from.

Listing 5.3 shows an example how to replace a running workflow. It stops the execution and replaces the original workflow description by a sequence of three activities.

In line 1 a new instance of the workflow is created. In this phase the state of the workflow instance is *ready*. A simple implementation of the intervention interface is shown in line 3 to line 17. It forwards the commands to the WEE and runs it in a new thread. In the example is assumed that a monitoring handler detects that the workflow is going to fail and orders a repair handler to take actions. The repair handler stops the execution of the workflow (see line 29, the state is set to *stopped*) through the intervention interface and replaces the original workflow description by a repaired version (line 30 to 34). As the flight has already been booked, the thread of control is set to the booking of the hotel (line 35) before the execution is continued again in line 36 (the state is set to *running* again).

LISTING 5.3: Example of Workflow Restructuring

```
1 wf = TripWorkflow.new # initial workflow
2
3 class IWrap < Wee::InterventionWrapperBase
4   def initialize(wf)
5     @wf = wf
6   end
7   def start
8     @tr = Thread.new { wf.start }
```

```
 9   end
10   def stop
11     @wf.stop
12     @tr.join
13   end
14   def search(position)
15     @wf.search position
16   end
17 end
18
19 int = IWrap.new
20 int.start
21 int.wait_for_external_input
22
23 ##########################################
24 # meanwhile in a repair handler somewhere
25 # else ...
26
27 int = Intervention.new("http://www.wee.org")
28 pro = Properties.new("http://www.wee.org")
29 int.stop
30 pro.wf_description do
31   activity :book_flight, :call, aeroflot
32   activity :book_hotel, :call, marriot, @marriot_customer_id
33   activity :book_taxi, :call, yellowcab
34 end
35 pro.search Wee::SearchPos.new(:book_hotel)
36 int.start
```

Modifying attributes of an existing workflow. Listing 5.4 shows how to change an endpoint and a context variable.

After a new intervention interface is instantiated (line 1), the new context variable *marriot_customer_id* is inserted by line 2. The additional endpoints needed for the marriot booking service and the taxi service are added in line 3 and line 4.

LISTING 5.4: Endpoint and Context Manipulation

```
1 pro = Properties.new("http://www.wee.org")
2 pro.context = {:marriot_customer_id => "666"}
3 pro.endpoint :marriot => "www.marriot.com"
4 pro.endpoint :yellowcab => "www.yellowcab.com"
```

5.4 Evaluation [3]

In this section we try to evaluate the coverage of possible workflow control structures with our basic set of elements. We choose the workflow patterns repository by Van der Aalst et al. [28] as starting point to validate our prototype. The control flow patterns focus on the dependencies between multiple activities and branches of a workflow.

[3] This section is a modified and extended version of the section "Evaluation" in: G. Stürmer, J. Mangler, and E. Schikuta, "A domain specific language and workflow execution engine to enable dynamic workflows," in Proceedings of the 1st International Workshop on Workflow Management in Service and Cloud Computing, IEEE Computer Society Press, 2009.

Each pattern can be supported by our execution engine in one of four different ways which are ordered by degree of support:

directly supported: The pattern is supported by an explicit language element in the WEE-DSL or a simple combination of them. ♡♡

modified workflow: The pattern can be expressed by rearranging existing elements. This is possible if the behaviour of the pattern can be imitated through the composition of other patterns that are implemented by the WEE. ♡

handler/external: The pattern can be implemented in cooperation with an adopted handler wrapper or other external modules. These modules are still orchastrated by the WEE through the worfklow description but contribute the actual logic how to implement the pattern in coordination with the WEE. *

orchestrated instances: The pattern can only be expressed by controlling multiple instances of the execution engine. In addition, these instances have to be orchestrated and managed by the controller of the instances (e.g. the controller is responsible for data exchange between the instances). As the controller has the control about each instance, the orchestration can be itemized down to the execution of single activities. Of course, the pattern is then not implemented by the WEE, the logic is mainly within the controller. ×

5.4.1 Basic Control Flow Patterns

Each of the basic control flow patterns is directly supported by an explicit WEE-DSL element.

Sequence. The sequence pattern indicates that an activity is started after another activity finishes. An example implementation can be seen in Listing 5.5. ♡♡

LISTING 5.5: Sequence pattern

```
1 activity :a1, :call, endpoint1
2 activity :a2, :call, endpoint2
```

Parallel split & Synchronization. The parallel split generates multiple branches which have to be executed concurrently. The synchronization merges this branches together as soon as all parallel branches have finished. Listing 5.6 shows the implementation in the WEE-DSL. ♡♡ ♡♡

LISTING 5.6: Parallel split & Synchronization

```
1 parallel :wait do
2   parallel_branch do
3     activity :branch1_a1, :call, endpoint1
4   end
5   parallel_branch do
6     activity :branch2_a1, :call, endpoint2
7   end
8 end
```

Exclusive Choice & Simple Merge. Exclusive choice selects one of multiple possible branches to be executed. Which branch to continue on is decided by a condition. The simple merge joins multiple branches into a single subsequent branch. The subsequent ♡♡ ♡♡

branch is executed as soon as one of the multiple branches has finished. The implementation of these two patterns can be seen in Listing 5.7. Multiple branches can be defined by the keyword *alternative*. Each alternative is guarded with a condition that indicates in which situation the branch should be executed as indicated in line 3. If none of the defined alternatives is executed, the optional *otherwise*-section is executed as indicated in line 6.

LISTING 5.7: Exclusive choice & Simple Merge

```
1 context :x => 5
2 choose do
3   alternative(@x > 1) do
4     activity :alternative1_a1, :call, endpoint1
5   end
6   otherwise do
7     activity :alternative2_a1, :call, endpoint2
8   end
9 end
```

5.4.2 Advanced Branching and Synchronization Patterns

♡♡ × ♡♡ **Multi-Choice, Multi-Merge & Structured Synchronizing Merge**. The multi-choice pattern extends the exclusive choice pattern by allowing multiple branches to be executed. The merge of these branches therefore has to merge one or more possible branches. The WEE-DSL does not distinguish between these patterns and the exclusive choice & the simple merge pattern. An example of a multi-choice with two executed branches can be seen in listing 5.8. The alternatives in line 3 and line 6 are both executed in this example. In this case the branches are not executed concurrent but one after another. Listing 5.9 shows an example how to implement the patterns to execute the branches concurrent which results in a structured synchronizing merge. Each *parallel_branch*-block is executed in a separated thread. To indicate which threads are running parallel, a *parallel*-block has to enclose the *choice*-block. The multi-merge pattern distinguishes from the structured synchronizing merge by keeping the multiple threads of control beyond the merging point. If the multi choice pattern results in the execution of two or more branches, each thread of control is passed to the subsequent branch in the way that the subsequent branch is executed two or more times. The multi merge pattern is not supported by the WEE.

LISTING 5.8: Multi-Choice & Structured Synchronizing Merge - Not concurrent

```
1 context :x => 5
2 choose do
3   alternative(@x > 1) do
4     activity :alternative1_a1, :call, endpoint1
5   end
6   alternative(@x < 10) do
7     activity :alternative2_a1, :call, endpoint2
8   end
9   otherwise do
10    activity :alternative3_a1, :call, endpoint3
11  end
12 end
```

LISTING 5.9: Multi-Choice & Structured Synchronizing Merge - Concurrent

```
1 context :x => 5
2 parallel do
3   choose do
4     alternative(@x > 1) do
5       parallel_branch do
6         activity :alternative1_a1, :call, endpoint1
7       end
8     end
9     alternative(@x < 10) do
10      parallel_branch do
11        activity :alternative2_a1, :call, endpoint2
12      end
13    end
14    otherwise do
15      parallel_branch do
16        activity :alternative3_a1, :call, endpoint3
17      end
18    end
19  end
20 end
```

Structured Discriminator. The structured discriminator pattern is similar to the structured synchronizing merge pattern. While the structured synchronizing merge pattern is used to join branches forked by a choice, the structured discriminator merges branches forked by a parallel split. Also, the structured discriminator does not wait until each branch finished but continues the subsequent branch as soon as *one* branch finishes. Other parallel branches may be still executed and are terminated as they approach at the structured discriminator. The structured discriminator is not supported directly in the WEE-DSL. However, the structured discriminator has two variants. × × ♡♡

The *Blocking Discriminator pattern* does not allow to fork parallel branches while another instance is still executing one of the branches. As the WEE focuses on the execution within one workflow instance, it does not support interactions or management between multiple instances. Therefore, this pattern is not supported.

The *Canceling Discriminator pattern* passes the thread of control to the subsequent branch as soon as one of the parallel branches has finished. All remaining parallel branches that are still executed are canceled. The WEE-DSL supports this pattern by providing a parameter to the parallel-block as can be seen in listing 5.10. The *wait*-parameter can be used to set how much branches have to be finished before the execution in the subsequent branch is continued. The *no-longer-necessary*-signal is sent to still running branches, urging them to quit execution.

LISTING 5.10: Canceling Discriminator

```
1 parallel :wait => 1 do
2   parallel_branch do
3     activity :branch1_a1, :call, endpoint1
4   end
5   parallel_branch do
6     activity :branch2_a2, :call, endpoint2
7   end
8 end
```

Structured Partial Join. The structured partial join pattern is similar to the structured discriminator but continues the subsequent branch as soon as a *specified amount* of branches finishes execution. As the WEE-DSL uses the same constructs for the Discriminator, the structural partial join and its variant, the *Blocking Partial Join* are not supported. The *Canceling Partial Join* works as a generalized version of the canceling discriminator, therefore the listing 5.10 shows both patterns. × × ♡♡

× **Generalised AND-Join**. The generalised AND-join pattern merges branches as soon as all incoming branches have been completed. In contrast to the synchronization pattern, the generalised AND-join supports the join across multiple threads of controls and passes the thread of control to the subsequent branch each time all incoming branches are completed. This is also true even when the parallel branches are not created by the same parallel split event. The WEE does not support the generalised AND-join pattern as it is only able to merge branches that result from the same parallel split event.

× **Local Synchronizing Merge & General Synchronizing Merge**. These two patterns
× can be used instead of the synchronization if the process description is not structured. Both patterns are able to pass the thread of control to the subsequent branch if all branches are executed or if it is sure that all outstanding branches are not going to arrive at the merge point. Other than in the local synchronizing merge situation, the information if branches are going to arrive at the merge point is not available locally in the general synchronizing merge situation. This information may be gathered by the evaluation of possible prospective states of the executed branch. Both patterns cannot be translated into the WEE-DSL as the process definition is expressed in a block oriented manner. Therefore, the process description is available in a structured way which conflicts with the basic reason for the two patterns.

♡♡ **Thread Split & Thread Merge**. The thread split pattern generates a specific amount of
♡♡ threads. Each of the threads executes the subsequent branch. The thread merge pattern is able to merge multiple thread that are generated by the thread split pattern. The pattern description states that the number of threads that have to be generated must be specified at design-time. Although the concepts of threads within a workflow instance was not intended in the design of the WEE-DSL apart from parallel branches, the dynamic generation of threads can be achieved by cascading the *parallel_branch*-element into a loop (*cycle*-element). As each *parallel_branch* forks a thread, this arrangement implements the thread-split and the thread-merge pattern through a minor modification of the workflow. An example implementation can be seen in listing 5.11.

LISTING 5.11: Thread-Split & Thread-Merge

```
1 parallel :wait do
2   context :x => 3
3   cycle("@x >= 0") do
4     activity :countdown, :manipulate do
5       @x -= 1
6     end
7     parallel_branch do
8       activity :a, :call, endpoint1
9     end
10  end
11 end
```

5.4.3 State Based Patterns

Deferred Choice. The deferred choice pattern chooses one of multiple possible branches to be executed. The decision, which branch has to be executed, is delayed as long as possible. In contrast to the choice-patterns, the decision is not made explicit by a condition. After one branch starts the execution, each other branch is aborted, therefore the decision is more based on a race of the branches which is executed next. There is no explicit element in the WEE-DSL to express the deferred choice pattern. However, with the implementation of the canceling discriminator, it is possible to achieve a similar result. A possible implementation can be seen in listing 5.12. Each branch is represented by a single activity which is part of a *parallel*-block (which in this case is a canceling discriminator). Line 4 and line 9 are showing the two representative activities. As soon as one of them completes, a context variable expressing which branch to execute is set (see line 5 and line 10). As the canceling discriminator aborts all other branches, the context variable now indicates which branch has to be executed further. The use of the exclusive choice pattern is now possible (see line 14). ♡

LISTING 5.12: Deferred Choice

```
1 context :choice => nil
2 parallel :wait => 1 do
3   parallel_branch do
4     activity :representA, :call, endpoint1, 1 do
5       @choice = 1
6     end
7   end
8   parallel_branch do
9     activity :representB, :call, endpoint2, 2 do
10      @choice = 2
11    end
12  end
13 end
14 choose do
15   alternative(@choice == 1) do
16     # branch 1 comes here
17   end
18   alternative(@choice == 2) do
19     # branch 2 comes here
20   end
21 end
```

Milestone. The execution of an activity is enabled as long as a specific point in the execution of the workflow is reached. In general, the milestone pattern needs at least two parallel branches. One branch marks if the milestone is reached and enables the execution of specific activities in the other branch(es). These activities cannot be executed before the milestone is reached or after the milestone is passed. The WEE does not provide a direct implementation of the milestone pattern. In general, two approaches can be used two tackle the problem: ♡

1. The milestone is activated by an explicit activity and deactivated by a explicit activity. Before an activity which is enabled by the milestone is executed, a *choose*-element checks for the activation. Listing 5.13 shows an example implementation. After activity *:activate_milestone* is executed (line 4), the milestone is reached, which is indicated by the context variable *milestone*. In the parallel branch, the

activity *:enabled* (line 17) is executed if the context variable *milestone* is true when the *choose*-element is reached (line 15). The milestone is valid as long as the activity *:keep_milestone* is running (line 7). The activity *:deactivate_milestone* (line 10) resets the context variable *milestone* and the *:enabled*-activity is not executed anymore. Instead, the *:not_enabled*-activity is called (line 20).

2. The controller of the workflow "injects" the enabled activity by restructuring the workflow. The workflow execution is suspended and the workflow description is adjusted by the enabled activity. When the milestone is expired, the workflow description is set back to the original. While doing this, the controller has to take care about all active activities and track the thread of control for each branch. In this scenario, the controller supervises when and how long a milestone becomes active. This alternative is more difficult to implement but applicable if the milestone logic should not be part of the workflow description. This may be true if specific activities should only be executed when certain technical issues are fulfilled (e.g. services are only temporary available or their call should be avoided normally)

LISTING 5.13: Milestone

```
1 context :milestone => false
2 parallel do
3   parallel_branch do
4     activity :activate_milestone, :manipulate do
5       @milestone = true
6     end
7     activity :keep_milestone, :manipulate do
8       sleep 5
9     end
10    activity :deactivate_milestone, :manipulate do
11      @milestone = false
12    end
13  end
14  parallel_branch do
15    choose do
16      alternative(@milestone) do
17        activity :enabled, :call, endpoint1
18      end
19      otherwise do
20        activity :not_enabled, :call, endpoint2
21      end
22    end
23  end
24 end
```

♡♡ **Critical Section**. Two or more areas of different (parallel) branches are defined to be prohibited to be executed at the same time for a given workflow instance. When an activity of a critical section is executed, that critical section has to be completed before the critical section can be entered again. This pattern is also commonly described as mutex or semaphore. The WEE-DSL has a separate element to define critical sections. Listen 5.14 shows an example of a critical section which spans over two parallel branches. As a critical section can span across different branches, an identifier is necessary to indicate the connection.

LISTING 5.14: Critical Section

```
1 parallel do
```

```
2   parallel_branch do
3     critical(:id) do
4       activity :parallel1, :manipulate do
5         sleep 2
6       end
7     end
8   end
9   parallel_branch do
10    critical(:id) do
11      activity :parallel2, :manipulate do
12        sleep 2
13      end
14    end
15  end
16 end
```

Interleaved Routing. A set of branches has to be executed. The branches can be executed in any order but must not be executed concurrently. None of the branches must be active as long as another branch is executed. The thread of control is passed to the subsequent branch when all branches (that are part of the interleaved routing) are finished. The WEE-DSL does not have an explicit element for the interleaved routing pattern. This pattern can be easily implemented by the use of the critical section pattern. Listing 5.15 shows a sample implementation. Two (initially parallel) branches are opened in line 2 and line 8. As the branches share a critical section, they cannot run simultaneously but one after another. ♡♡

LISTING 5.15: Interleaved Routing

```
1 parallel do
2   parallel_branch do
3     critical(:id) do
4       activity :branch1_task1, :call, endpoint1
5       activity :branch1_task2, :call, endpoint2
6     end
7   end
8   parallel_branch do
9     critical(:id) do
10      activity :branch2_task1, :call, endpoint3
11    end
12  end
13 end
```

Interleaved Parallel Routing. The interleaved parallel routing pattern enhances the interleaved routing pattern by restricting the execution to the activity-level. Parallel branches may be active but only one activity over all branches in the interleaved parallel routing is executed at one time. Again, this can be achieved by the use of the critical section pattern. Not the critical section has to allow a "switch" in the execution of the parallel branches. Therefore, the *critical*-elements have to span around each activity to give activities from other branches the chance to be executed. This can be seen in listing 5.16. ♡♡

LISTING 5.16: Interleaved Parallel Routing

```
1 parallel do
2   parallel_branch do
3     critical(:id) do
```

```
4         activity :branch1_task1, :call, endpoint1
5       end
6       critical(:id) do
7         activity :branch1_task2, :call, endpoint2
8       end
9     end
10    parallel_branch do
11      critical(:id) do
12        activity :branch2_task1, :call, endpoint3
13      end
14    end
15  end
```

5.4.4 Multiple Instances

The multiple instances patterns are created to deal with the generation of multiple instances of an activity or task within a given workflow instance. The different patterns focus on aspects like the amount of created instances, synchronization and merging. The generated instances run independent and concurrent to each other but mostly need to be synchronized with the workflow instance when finished. The WEE basically has no concept of multiple instances for an activity. However, the multiple instance patterns overlaps with the thread split/thread merge pattern and can be expressed by them. If this is not the case, the behavior often can be simulated by a suitable handler wrapper implementation. As the handler wrapper is ordered by the WEE with the execution of an activity, the handler wrapper can spawn the needed amount of instances and is able to control them as long as needed.

* **Multiple Instances without Synchronization**. Multiple instances of an activity are created but run independent from the workflow instance. The subsequent branch does not have to wait for the execution of the multiple instances. The amount of instances is not defined, consequentially the timespan in which new instances can be created is not defined. This multiple instance pattern cannot be expressed by the WEE-DSL. An adjusted handler wrapper can simulate the behavior by spawning the needed amount of instances in separated threads without giving the WEE notice. This solution does not address the problem if the workflow instance finishes before all instances spawned by the handler wrapper have finished.

♡♡ **Multiple Instances with a Priori Design-Time Knowledge**. A number of instances is created. The amount is specified at design time. The thread of control is passed to the subsequent path as soon as all instances have been finished. The WEE supports this pattern in the same way as the thread-split and thread-merge pattern. Listing 5.11 therefore can also be seen as an implementation of this pattern. The context variable x (defined in line 2 determines the amount of created instances of the activity defined in line 8.

♡♡ **Multiple Instances with a Priori Run-Time Knowledge**. A number of instances is created. The amount of instances is determined by the workflow logic before the first creation of an instance. The WEE supports this pattern in the same way as the thread-split and thread merge pattern. In contrast to the solution above, the number of created instances is determined by an activity. Listing 5.17 shows an example implementation of the pattern. In line 2, the context variable is initialized to a default value. The activity

determine (line 3 to line 5) sets the number of instances that have to be created. The loop (line 6) then generates the multiple instances of activity *a* (line 11).

LISTING 5.17: Multiple Instances with a Priori Run-Time Knowledge

```
1 parallel :wait do
2   context :x => 0
3   activity :determine, :call, endpoint1 do |count|
4     @x = count
5   end
6   cycle("@x >= 0") do
7     activity :countdown, :manipulate do
8       @x -= 1
9     end
10    parallel_branch do
11      activity :a, :call, endpoint1
12    end
13  end
14 end
```

Multiple Instances without a Priori Run-Time Knowledge. A number of instances have to be created. The number of instances is not determined until the last instance has finished execution. The logic if more instances have to be created cannot be determined a priori but is derived from the execution process, resources or external services. As the logic of creating further instances has to be placed inside the workflow description, the implementation differs from the multiple instances patterns above. The execution of the subsequent branch has to be blocked until the last instance has finished the execution. Listing 5.18 shows an example implementation with the WEE-DSL. In line 2 a context variable is defined which indicates if a new instance of the activity has to be created. This is at least true for the first time. The loop in line 3 creates a new instance of activity *a* (line 5) as long as the context variable *create_instance* is true. The context variable *create_instance* is set by an external service (line 7) which defines if another instance of activity *a* has to be created. As the call of the external service is not part of the parallel execution, it is independent from the execution of the activity instance. Therefore it can determine the creation time for each activity instance. Multiple instances can be created parallel or one after another by blocking the service call until an instance has finished execution. ♡♡

LISTING 5.18: Multiple Instances without a Priori Run-Time Knowledge

```
1 parallel :wait do
2   context :create_instance => true
3   cycle("@create_instance") do
4     parallel_branch do
5       activity :a, :call, endpoint1
6     end
7     activity :decide, :call, endpoint2 do |result|
8       @create_instance = result
9     end
10  end
11 end
```

Static Partial Join for Multiple Instances. The static partial join for multiple instances forwards the thread of control to the subsequent branch as soon as a given amount of instances finished execution. Other, still running instances, have to be completed to re-enable the pattern but the result of these instances is withdrawn. The WEE × ♡♡ ×

does not support this pattern as it can only join all finished instances or abort instances which are no longer necessary. There are two variants of this pattern: The *Canceling Partial Join for Multiple Instances* pattern is analogue to the canceling partial join pattern described above. Other than the merge of branches, the canceling partial join for multiple instances pattern joins a given number of instances. The number of instances can be determined at design time or before the first instance is created. Listing 5.19 shows how to implement this pattern with the WEE-DSL. Multiple *parallel-branch*es (and therefore multiple instances of activity *a*) are created by the loop (line 3 to 10). As each *parallel_branch* belong to the *parallel*-block defined in line 1, the number of *parallel_branches* (and therefore the number of instances) that have to be completed can be defined here. The *parallel*-construct forwards the thread of control to the subsequent path as soon as the given number of *parallel_branches* has completed. The *nolongernecessary*-signal is sent to unfinished *parallel_branches*.

LISTING 5.19: Canceling Partial Join for Multiple Instances

```
1 parallel :wait => 1 do
2   context :x => 3
3   cycle("@x >= 0") do
4     activity :countdown, :manipulate do
5       @x -= 1
6     end
7     parallel_branch do
8       activity :a, :call, endpoint1
9     end
10  end
11 end
```

The *Dynamic Partial Join for Multiple Instances* pattern provides the most flexibility when dealing with the merge of multiple instances. New instances can be created as long as the last instance has not finished. The number of instances depends on the execution progress or external information. After an instance has been executed, the thread of control can be passed to the subsequent branch. All other running instances are then withdrawn. The WEE does not support this pattern as the number of instances which have to be completed must be known at the first instance creation.

5.4.5 Cancellation and Force Completion

♡♡ **Cancel Task & Cancel Case**. The workflow instance is aborted and removed from ♡♡ execution. The cancel task pattern and the cancel case pattern are distinguished by the option *when* the cancellation is possible: The cancel task pattern specifies that the cancellation is possible at a specific activity. This activity and further also the workflow instance is aborted. The cancel case pattern allows to abort a workflow instance not only at the point of the execution of an activity but for the whole execution of the instance. The WEE does not distinguish between these patterns. It allows to stop the execution at any point during execution. The controller of the workflow instance has the possibility to send the stop-signal at any time. If a *call*-activity is executed, the execution is delegated to the handler wrapper. Therefore the WEE cannot guarantee that the activity is aborted. The WEE can only inform the handler wrapper that it should stop the execution. The handler wrapper is then responsible to decide whether the service call can be aborted or not. This is important as the immediate abortion

may lead to an inconsistent state or cannot be done without sanction which has to be avoided by the handler wrapper.

Cancel Region. Other than the cancel task and cancel case pattern, the cancel region pattern may not lead into the cancellation of the workflow instance. If an active branch (or parts of it) is outside a cancel region, this branch is unaffected by the cancellation. The cancel region pattern defines a set of activities (which may be located also in different branches). Activities which are in the process of execution within this area are aborted when the cancel region pattern is activated. Activities which are in the process of execution outside of this area stay unaffected. To implement this pattern with the WEE comes with some difficulties. The WEE cannot abort only parts of the execution. If the WEE receives the stop-signal from the controller, the workflow instance is stopped. Therefore, also activities which should stay in the process of execution are affected, which is not the intention of the pattern. Even if the handler wrapper does not allow the abortion, the result of the service call will not be integrated into the workflow by the WEE. The handler wrapper will not even be asked to provide the result value. But the WEE will ask for a *passthrough* value which gives the handler wrapper the possibility to store the result and reuse it when the workflow is continued. The sequence diagram in Fig. 5.1 now shows how the cancel region pattern can be implemented by the WEE. After the handler wrapper receives the *stop_call*-signal, the service call is continued unaffected and the result of the service call is stored. The passthrough-value (identifying the result for later use) is returned to the controller of the workflow instance. When the controller continues the execution, the thread of control is set to the activities outside of the cancel region. The handler wrapper is provided with the passthrough-value and can look up the stored result. Therefore the service call does not have to be repeated (which may be costly). *

Cancel Multiple Instance Activity. Multiple instances of an activity are executed within one workflow instance. The cancel multiple instance activity pattern aborts the workflow instance and subsequentially all active instances of an activity. Already completed instances of an activity are unaffected. The WEE covers this pattern with the same procedure as the cancel case pattern. The WEE allows to stop a workflow instance at any point during execution, therefore it is also possible to abort running instances of an activity. ♡♡

Complete Multiple Instance Activity. Multiple instances of an activity are created. During the execution of the instances, the pattern indicates that the subsequent branch has to be executed. Therefore, remaining instances are no longer executed and withdrawn. Already finished instances are synchronized and their results are integrated into the workflow. The thread of control is passed to the subsequent branch. The WEE does not support this pattern. Although the WEE supports the manipulation of the thread of control, it is necessary to stop the execution before the manipulation can be done, which is not in the intention of the pattern. ×

5.4.6 Termination and Triggers

Implicit Termination. A branch terminates if no further control structures have to be executed. If all branches have terminate, the workflow instance is also terminated and ♡♡

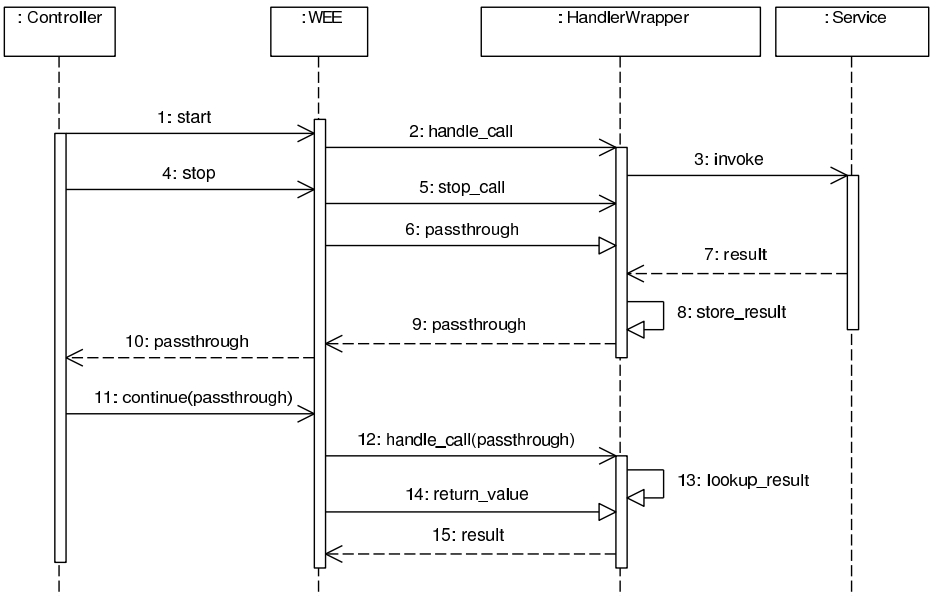

Figure 5.1: Cancel Region Pattern

the workflow execution is marked as completed successfully. Due to the block structure of the WEE-DSL, the implicit termination is inherent.

♡♡ **Explicit Termination**. The workflow instance is terminated as soon as a specified point in the execution is reached. All remaining branches are canceled. The WEE allows the handler wrapper or the controller of the workflow to stop the execution at any point during the execution by sending the *stop*-signal. The same mechanism is provided to the workflow description. Listing 5.20 shows an example implementation. The activity in line 1 sends the *stop*-signal to the WEE. If the handler wrapper has still running service calls, the handler wrapper is ordered to stop them if possible. Other activities will not be started. In contrast to the implicit termination, the end state of the workflow instance is set to stopped.

LISTING 5.20: Explicit Termination

```
1 activity :stop, :manipulate do
2   stop
3 end
4 activity :whatever, :manipulate do
5   # will not be executed
6 end
```

* **Persistent & Transient Trigger**. An activity is enabled by a trigger. The persistent
* trigger pattern is implemented if the trigger event is stored until the activity is scheduled for execution. The event enables the activity during any time in the workflow execution. The transient trigger enables an activity only when it is scheduled for execution. If the transient trigger event occurs before the activity is scheduled for execution, the event is withdrawn. An activity which has not been enabled by a trigger blocks the execution until a trigger event occurs or the workflow execution is canceled. The WEE does not support triggers directly. As the execution of a service call is delegated to the handler wrapper, the trigger logic can be placed there. The handler wrapper can block the

execution until the trigger event occurs. The downside of this approach is that the triggering logic is not anymore in the workflow description.

5.4.7 Iterations

Arbitrary Cycles. The arbitrary cycles pattern allows unstructured loops. The thread of control can be set back to a defined point in the workflow description. The defined point can be located within another arbitrary cycle. Fig. 5.2 shows an example of a of an arbitrary cycle. As the WEE-DSL has a block structure, the arbitrary cycles pattern can not be expressed directly. Instead, the WEE allows modification of the thread of control which can be used to simulate the behavior. Fig. 5.3 shows how a modified handler wrapper has to act to implement the pattern. Activity 1 and Activity 2 are performed normally as service calls. The next *call_service*-message is then interpreted by the handler wrapper. The decision if the thread of control has to be altered is based on the condition provided by the workflow description. If the condition is validated as true, the thread of control is set to the given position identifier. The necessary information can therefore be specified in the workflow description and is not hard-wired in the handler wrapper. Although this implements the arbitrary cycles pattern, a complete implementation of the handler wrapper may also take care of possible side-effects which can arise when dealing with parallel active branches. *

Structured Loop. An activity or a set of activities and control structures are executed repetitive. The loop can be top-controlled or bottom-controlled. In contrast to the arbitrary cycles pattern, a structured loop can be expressed in a block manner. The WEE-DSL directly supports the structured loop pattern with the *cycle*-element. This element implements the top-controlled loop. Bottom-controlled loops are not supported but can be expressed through a top-controlled loop with minor overhead. Listing 5.21 shows a sample of a structured loop. ♡♡

LISTING 5.21: Structured Loop

```
1 context :x => 3
2 cycle("@x > 0") do
3   activity :countdown, :manipulate do
4     @x -= 1
5   end
6 end
```

Recursion. The execution of an activity results in the execution of a new instance of the overall workflow description that is currently executed. Each recursion has to have at least one exit condition. As the WEE does not has the concept of multiple instances, the recursion cannot be supported directly. Instead, a recursion can be seen as a normal service call which in turn invokes a new workflow instance. The implementation can be therefore done by the handler wrapper. *

5.4.8 Comparison to Other Engines

When we compare WEE with other workflow engines, the boundaries of our execution engine become apparent. Our execution engine focuses on the control flow aspect of workflows.

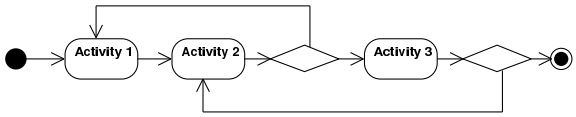

Figure 5.2: Arbitrary Cycles Pattern Example

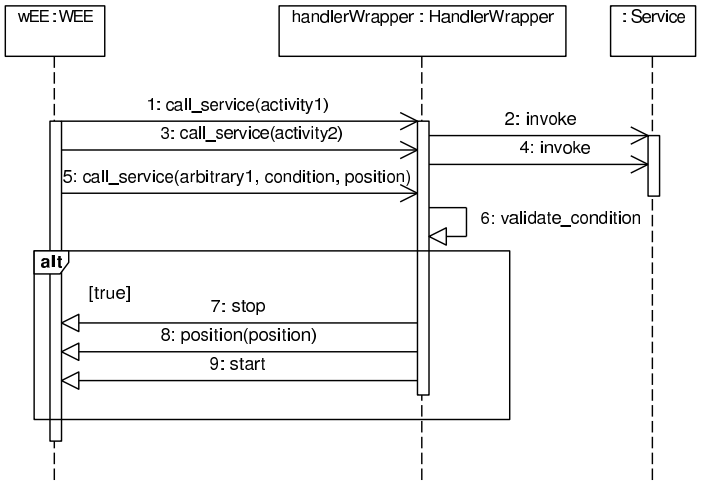

Figure 5.3: Arbitrary Cycles Pattern Implementation

More precisely, we even focus on a single thread of control within our execution engine (except parallel branches).

Existing workflow engines on the other hand have to cover a much larger set of functions and do not only focus on the control flow aspect. They also provide support for data and resource handling, security, logging, repair strategies and much more.

Table 5.1 shows a summary of the pattern coverage as described so far.

Table 5.1: Workflow Patterns Coverage

Pattern class	Pattern name	directly supported	modified workflow	handler/external	orchestrated instances
Basic Control Flow	Sequence	♡♡			
	Parallel Split	♡♡			
	Synchronization	♡♡			
	Exclusive Choice	♡♡			
	Simple Merge	♡♡			
Advances Branching and Synchronization	Multi-Choice	♡♡			
	Structured Synchronizing Merge	♡♡			
	Multi-Merge				×
	Structured Discriminator				×
	Blocking Discriminator				×
	Cancelling Discriminator	♡♡			
	Structured Partial Join				×
	Blocking Partial Join				×
	Cancelling Partial Join	♡♡			
	Generalised AND-Join				×
	Local Synchronizing Merge				×
	General Synchronizing Merge				×
	Thread Merge	♡♡			
	Thread Split	♡♡			
Multiple Instances	Multiple Instances without Synchronization			*	
	Multiple Instances with a Priori Design-Time Knowledge	♡♡			
	Multiple Instances with a Priori Run-Time Knowledge	♡♡			
	Multiple Instances without a Priori Run-Time Knowledge	♡♡			

Pattern class	Pattern name	directly supported	modified workflow	handler/external	orchestrated instances
	Static Partial Join for Multiple Instances				×
	Cancelling Partial Join for Multiple Instances	♡♡			
	Dynamic Partial Join for Multiple Instances				×
State Based	Deferred Choice		♡		
	Interleaved Parallel Routing	♡♡			
	Milestone		♡		
	Critical Section	♡♡			
	Interleaved Routing	♡♡			
Cancellation and Force Completion	Cancel Task	♡♡			
	Cancel Case	♡♡			
	Cancel Region			∗	
	Cancel Multiple Instance Activity	♡♡			
	Complete Multiple Instance Activity				×
Iteration	Arbitrary Cycles			∗	
	Structured Loop	♡♡			
	Recursion			∗	
Termination	Implicit Termination	♡♡			
	Explicit Termination	♡♡			
Trigger	Transient Trigger			∗	
	Persistent Trigger			∗	

Wohed et al. [29] created a pattern based evaluation of different open source workflow systems. The analysis of our pattern support above allows to compare the coverage of our execution engine with this evaluation. In contrast to Wohed et al., we used a more detailed categorization with levels of support instead of just coverage. To allow a comparison, we assume that *directly supported* (♡♡) is equal to "+", *modified workflow* (♡) and *handler/external* (∗) are equal to "+/-" and *orchestrated instances* (×) refers to "-" as this is the least

Table 5.2: Support of Control Flow Patterns

Product	*+*	*+/-*	**-**
WEE	22	10	11
StaffWare 10	14	0	29
WebSphere MQ 3.4	11	0	32
Oracle BPEL PM 10.12	18	6	19
JBoss jBPM 3.1.4.2	13	2	28
OpenWFE 1.7.3	20	4	19
Enhydra Shark 2.0	11	0	32

supported category.

The results of the comparison (see Tab. 5.2) show that WEE, although minimal, covers a range of patterns that enables it to outperform several commercial solutions. We conclude therefore that converting workflow description languages to the WEE-DSL can be achieved with low effort.

5.5 Technical Realization

In this section, we investigate further how we implemented the WEE. We will describe how we used the possibilities of the Ruby programming language and point out areas that may be of special interest. First, we will elaborate the basic design we applied. Next we will depict Ruby metaprogramming techniques we used to create the DSL. The following sections then specify the respective DSL-elements categorized by the usage perspective. In the last section, we have an eye on the search mechanism which influences some of these elements.

Befor we continue with the description of our implementation, we summarize some terms and elements of the Ruby programming language:

- A class is defined with: *class* Identifier [<Superclass] body *end*.
- An instance method is defined with: *def* identifier[*(*parameters**)*] body *end*.
- A class method is defined with: *def self::*identifier[*(*arguments**)*] body *end*.
- An instance variable is accessed with: @identifier.
- A class variable is accessed with: @@identifier.
- A local variable is accessed with: identifier.
- A code block is a lambda (i.e. lambda calculus). Blocks have access to the context they are invoked in and may have arguments. They can be passed to a method (or another code block) as an argument. When passed to a method, they are available implicit but can be made explicit in the argument list with &identifier. Code blocks are defined with: invocation *do*[|arguments*|] body *end*
- A symbol is an internalized & unique string and defined with: *:*identifier

- An array is defined with: *[* element1, element2, ... *]*
- A hash is defined with: { key1 => value1, key2 => value2, ... } The Brackets can be omitted if the hash can be identified as a hash without being separated from the rest of the term.
- The term "Metaprogramming" refers to the ability to manipulate the structure and behavior of a program by an external program or the program itself.
- An "Eigenclass" is the class definition derived from (and still linked with) an instantiated object and independent from the overall class description of this object.

5.5.1 Basics

A workflow is defined as a ruby class. We already provided examples how such a class can look like. With the help of listing 5.22 we identify the following characteristics:

LISTING 5.22: A Basic Example of a Workflow

```
1 class ExampleWorkflow < Wee
2   handler BasicHandler
3   context :x => 5
4   endpoint :flight => 'http://someuri.com'
5
6   control flow do
7     # ... workflow description belongs here
8   end
9 end
```

- The workflow class (*ExampleWorkflow*) inherits from the WEE-class (*Wee*) which provides DSL-elements as inherited class or instance methods. Each workflow instance therefore embeds a WEE-instance. Technically, the WEE does not orchestrate the workflow from outside but is an integrated part of it. The programming logic how to handle an *activity*, *choice*, etc. is inherited from the parent class *Wee*.
- Handler
  - *handler* classname [, argument]*

  With the *handler*-keyword, a handler wrapper class can be specified. *handler* is a class method of the WEE-class. The first argument has to be a class and must inherit from *Wee::HandlerWrapperBase*. Other optional arguments will be provided as constructor arguments each time the WEE instantiates the handler wrapper class. The class *Wee::HandlerWrapperBase* does only provide default settings at this stage of development but could also provide commonly needed functionality in the future.
- Context and Endpoints
  - *context* :identifier => value [, :identifier => value]*
  - *endpoint* :identifier => value [, :identifier => value]*

  Initial *context* variables and *endpoints* can be defined with the corresponding keywords. Both keywords, *context* and *endpoint*, are class methods of the WEE-class and expect

a hash object as argument. Each key-value pair of the hash object is transformed into a context variable or an endpoint.

  - A context variable which is defined through the keyword *context* is transformed to an instance variable and can then be accessed like a normal ruby instance variable with the @-sign in the workflow description. The key of the hash entry defines the variable name, the value of the hash entry defines the value of the instance variable. The difference to the usage of only instance variables is that the context variables will be supervised by the WEE. The controller or a external handler could not access the context variables through the control of the WEE if they were handled as instance variables only. Therefore, the context keyword was introduced to signalize that this variable has to be supervised and accessible via the WEE from outside of the class.
  - An endpoint which is defined through the keyword *endpoint* is translated into an instance method. The key of the hash entry defines the name of the method. The method returns the value of the hash entry.

  A detailed description how these elements are transfered can be found in the following section.

- Workflow description
  - *control flow* &codeblock

  The workflow description itself is written into a code block. The code block is passed to the class method *control* as argument. The control method accepts two arguments: The first is a dummy argument and results from the usage of the *flow* keyword. *flow* is also a class method and does nothing. The second argument is the code block with the workflow description. The code block is stored in a instance method called *__wee_execute* with the use of *define_method*. The method *__wee_execute* sets the state to *running* and executes the code blocks. After the code block has finished, the state is set to *finished* if the code block was executed without being stopped. The method returns an array containing the final state, the context and the positions where it was stopped if applicable.

5.5.2 Applied Metaprogramming

As elaborated in the previous section, we used some metaprogramming facilities of the Ruby programming language to express our domain specific language. Creating a domain specific language mainly means to provide keywords which provide specific behavior. For each example given here, we must clearly distinguish at which time (class definition time, object/instance time) changes are made. It is also important to note *what* is changed. Changes to the class definition only affect new created objects of such a class. Changes to an object only affect that object, even if it is a structural change. These changes only alter the Eigenclass of the object, and not the overall class description.

The first case describes how the endpoint definitions are translated to an method: Listing 5.23 shows how the class method *endpoint* is defined. For each entry of the input hash, an instance method with the given name is created with the *define_method*-method (see line 3 to 5). *define_method* takes the method name as argument, the method body is provided as

code block. Each endpoint should be changeable during the course of execution, therefore an assignment method with the same name (+ '=') is defined in line 6 to 9. The assignment method redefines the method. As this happens at instance time, *define_method* cannot be used. Instead we can redefine the method using *def*. As a consequence (*def* is executed at instance time!), the variable scope changes and the argument new_value is not available when the method should be redefined. We therefore have to store the value into a explicit instance variable which is created in line 7 using *instance_variable_set*.

LISTING 5.23: Implementation of the class method *endpoint*

```
1 def self::endpoint(param)
2   param.each do |name,value|
3     define_method name do
4       return value
5     end
6     define_method "#{name}=" do |new_value|
7       instance_variable_set("@#{name}", new_value)
8       instance_eval("def #{name}\n return @#{name}\n end")
9     end
10  end
11 end
```

The second case describes how the context variables are translated to an instance variable: The problem here is that the class description specifies the value of the variable at class definition time. As the creation of an instance variable at class definition time is impossible, the creation must be "carried over" to instance creation time. In a first step, we store the variable definition in a class variable. When the object is then created, we can allocate the instance variables with *instance_variable_set*. Listing 5.24 shows the actual implementation. In line 3, we merge the given hash into the class variable where we store each initial definition of a context variable. The merge is only necessary to allow us to process multiple occurrences of *context*. We then define a method called *init_context* which will allocate the instance variables (see line 4 to 8). It is important that this cannot be done immediately, as we are still in the time of class definition. We have to execute this method at object instantiation time, therefore we have to place the method call into the constructor of the class. We do this dynamically by redefining the initialize method of the class in line 13 to 15. For our purpose, this is a valid solution as we do not have programming logic placed in the constructor. If the original constructor must be executed, we would rename the constructor first and call it in our new constructor definition.

LISTING 5.24: Implementation of the class method *context*

```
1 def self::context(param)
2   @@__context ||= {}
3   @@__context.merge! param
4   define_method :init_context do
5     @@__context.each do |name, value|
6       instance_variable_set(("@" + name.to_s).to_sym,value)
7     end
8   end
9   prepare_initialize
10 end
11
12 def self::prepare_initialize
13   define_method :initialize do
14     init_context if methods.include?('init_context')
```

```
15   end
16 end
```

The third case covers how the workflow description is stored and replaceable after object creation: As we see in the workflow example, the workflow description is provided as a code block. Whenever the controller of the workflow wants to start, this code block has to be executed. A first approach is to store the given code-block as an instance variable. Listing 5.25 shows an example. The original method is defined in line 1 to line 3. To replace the method behavior, the method *replace* is called. The given code block is stored into the instance variable *@blk* in line 5. The original method is redefined with the use of *instance_eval* (line 6 to 10), therefore only this object is affected. Of course we could skip the redefinition of the method after the first replacement as we only have to call the block. This approach is valid but has a downside: It is not possible to bring a flexible amount of arguments into the code block. In the given example, no argument can be passed to the code block. Also, saving the block into an instance variable can be seen as a suboptimal solution as the workflow description is more similar to a method itself then a peace of information that has to be stored. The alternative approach therefore tries to embed the workflow description more into the object behavior. Listing 5.26 achieves this with the use of the Eigenclass. The original method is defined in line 1 to line 3. In line 5, we get the Eigenclass and let it evaluate the *define_method* in line 6 (at this point is no difference in the use of *instance_eval* or *class_eval*). In contrast to the first approach, we completely replace the method body each time with the new workflow description. We do not need to store the code block as a variable. Instead we directly integrate it into the objects behavior. As we change the Eigenclass, the replacement also effects this object only.

LISTING 5.25: Replace a method body with the instance variable approach

```
1 def method
2  p "original"
3 end
4 def replace(&blk)
5  @blk = blk
6  instance_eval do
7   def method
8    @blk.call
9   end
10  end
11 end
```

LISTING 5.26: Replace a method body with the Eigenclass approach

```
1 def method
2  p "original"
3 end
4 def replace(&blk)
5   (class << self; self; end).instance_eval do
6     define_method :method, &blk
7   end
8 end
```

5.5.3 Elements from the Workflow perspective

A workflow description can contain several elements of the DSL. This section describes how this elements are implemented and points out aspects that have to be considered.

context and endpoint.

- *context* :identifier => value [, :identifier => value]*
- *endpoint* :identifier => value [, :identifier => value]*

For an initial definition *context* and *endpoint* are defined as class methods as pointed out above. As the class methods are independent of the object, this is not sufficient when we want to declare a new context variable or endpoint during the execution of a workflow description. Therefore, both elements have to be defined as instance methods as well.

activity.

- *activity* :position, *:call*, endpoint [, parameters]* [&codeblock]
- *activity* :position, *:manipulate* &codeblock

The activity-element is a core component and represents an atomic operation of the workflow. *activity* is an instance method of the WEE-class. The first argument is the position identifier of the activity. The position identifier has to be unique within the workflow description and is used for the search procedure. The second argument identifies the type of the activity. Valid values are *:call* to indicate the activity is a service call, or *:manipulate* to indicate that the activity has to be executed within the WEE. The arguments endpoint and parameters are only processed if the type of the activity is *:call*. They will be provided to the handler wrapper when the service call has to be executed. An outline of the procedure how an activity is performed can be seen in Fig. 5.4. The first decisions check if the activity has to be executed at all. This is not true if:

- The controller has stopped the workflow execution. As the workflow description is executed by the Ruby interpreter, an abortion of the execution is not directly possible. This is problematic as the WEE must have the control about the stop procedure (like aborting service calls in a secure way). Therefore, a stop does not mean that the execution itself is stopped. Instead the workflow description is further executed by the interpreter, but activities do not execute their workflow logic.
- The active branch is not longer necessary. If parallel branches are executed, the WEE enables to merge a predefined amount of finished branches. Other branches should then not be executed further as they do not contribute to the workflow progress and waste resources. In this case, the same procedure is applied as if the branch has been stopped.
- The active branch is still in search-mode and the activity is not a starting position for the execution. A more detailed description of the implementation of the search procedure is given in section 5.5.5.

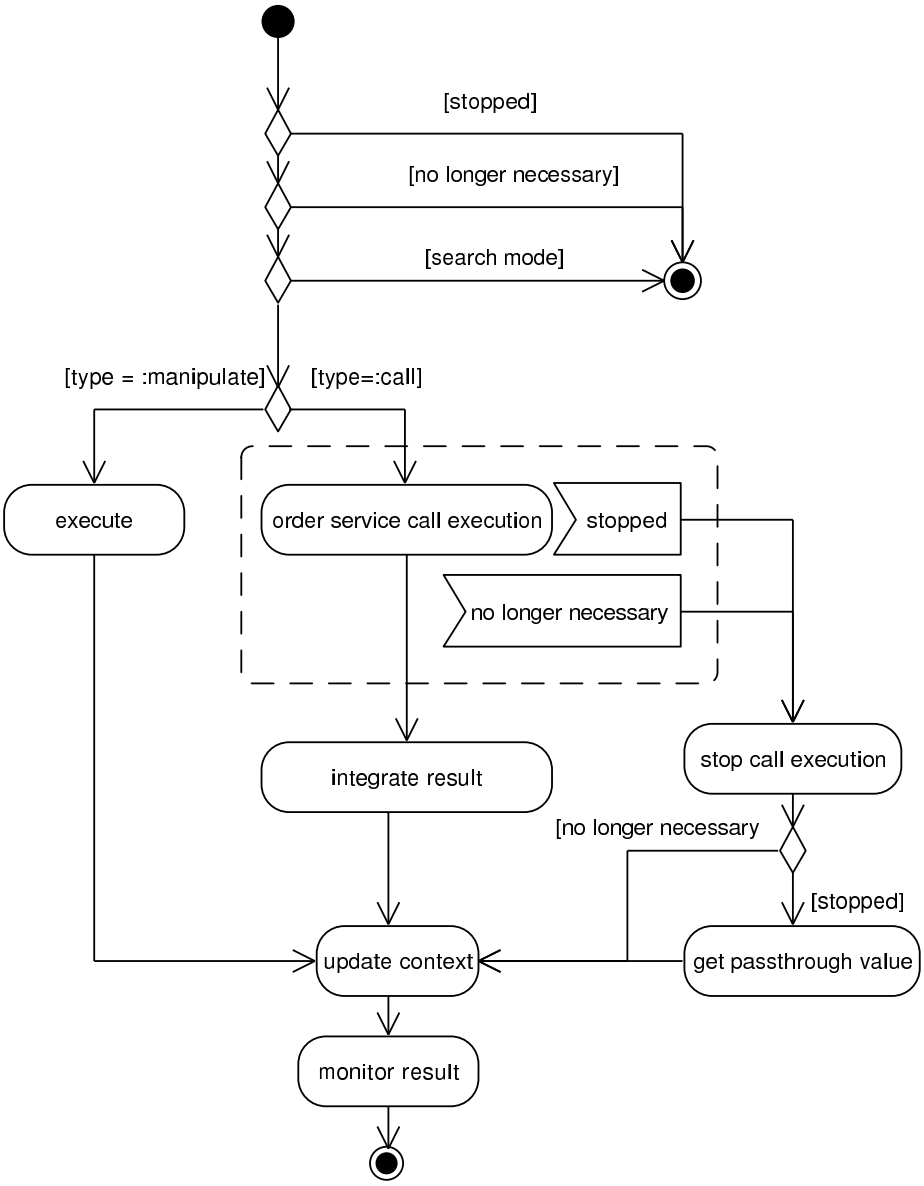

Figure 5.4: Execution of an activity

After this initial check, the type of the activity determines the further procedure. Activities of type *:manipulate* have to provide a code block that is directly yielded. *:call* activities are basically directed to the handler wrapper for execution. The handler wrapper is provided with the endpoint and the parameters. The procedure how the handler wrapper is invoked in the service call has already been described in section 4.3.1, basically the WEE polls the handler wrapper until the service call is completed or the service call has to be aborted.

The service call can be aborted for two reasons: (1) The execution of the workflow has to be stopped. (2) The branch is no longer necessary as enough parallel branches have finished. Both cases cause the WEE to inform the handler wrapper to stop the service call by calling the *stop_call*-method. If the execution is stopped, the handler wrapper is asked for a *passthrough value* which can be used to continue the service call later. This is not necessary when the call is aborted for the second reason. In this case, the workflow execution itself continues normally, the service call will not be continued in any case.

However, if the service call was not aborted and completed successfully, the result has to be integrated into the workflow. The creator of the workflow description can therefore provide a code block to the activity. This code block is yielded and the result of the service call is provided to the code block as parameter. If the parameter is an array, the WEE enables to expand the parameters to an argument list if necessary. The behavior can be set the handler wrapper. If *handlerwrapper.expand_params* returns *true*, an array will be provided as an argument list to the code block, otherwise it will be provided as a single parameter. The programmer of an handler wrapper does not necessary take care of this, the basic handler wrapper (class *WEE::HandlerWrapperBase*) provides a default value (which is *false*).

After the workflow logic has been processed, the workflow context is updated for supervision purposes and the result of the activity execution is monitored by invoking the appropriate methods of the handler wrapper (*inform_activity_done* or *inform_activity_failed*), depending if the execution was successfully or failed.

parallel.

- *parallel* [*:wait* | *:nowait* [=> finished_branches_count]] *do*
  [*parallel_branch* &codeblock]*
  end

The parallel-element is used to express concurrent running branches. Each of the separated branches is defined with *parallel_branch*. Both elements are defined as instance methods. *parallel* encapsulates branches that have to run in concurrent and coordinates the fork of the threads. The instance method *parallel* does not fork the threads itself. In a first step, it yields the provided code block in a mutex. The code block contains the definition of the parallel branches. Therefore, the instance method *parallel_branch* is called for each parallel branch within the parallel block. The *parallel_branch* now only has to fork a new thread with the provided code block, which now contains the code only for this branch (see Fig. 5.5). The new thread is added to a list. After all threads have been generated, the *parallel*-method uses this list to know which threads are direct child elements of it that have to be supervised. It is important to state that the mutex does not inhibit nested parallels, therefore it is not a "simulated concurrency". The

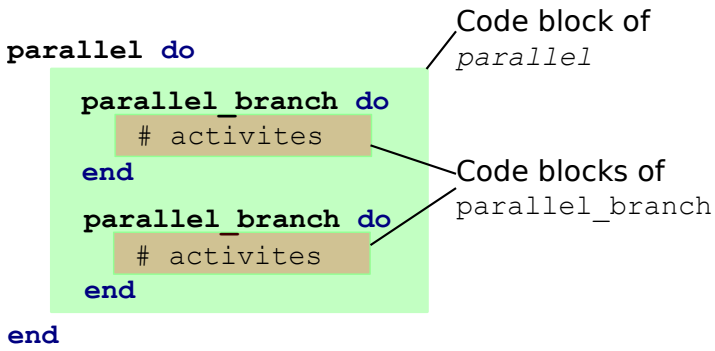

Figure 5.5: Code block arrangement of parallel branches

Mutex only protects the generation of the threads as they have to be controllable in the way that a merge will merge the correct threads together. Each *parallel* therefore only has its own list of running threads which are its direct children. As soon as all thread of the same "level" are generated they can run concurrently. Nested parallel commands can be executed as soon as the *generation* of the higher-ranked threads has been completed. It does not depend on the *completion* of a parallel or higher-ranked thread.

By default, the instance method *parallel* finishes when all parallel branches have finished execution (i.e. all threads have another status than *alive*). It is possible to determine the count of finished by providing an appropriate argument. For the readability of the DSL, this argument is a hash with one entry. The key of this entry must be *:wait*, the value must be the number of parallel branches that have to be waited for. If the key is *:nowait*, the default value is 1. To wait for the specified amount of thread, the *parallel*-method calls *Thread.pass* until the specified amount of threads have another state than *alive*. After this is reached, all other threads do not need to be executed further, they have to receive the *no_longer_necessary*-signal. This is done by setting a flag in the thread environment (*thread[:nolongernecessary] = true*). As we already pointed out above, this flag is checked each time the activity-method is entered (and is also checked in decisions, loops etc.).

choice.

- *choice do*
  [*alternative(*condition*)* &codeblock]+
  [*otherwise* &codeblock]
  end

Choice is an instance method that allows to choose if specific branches are executed. Choice is only executed if the state is not stopped and the actual thread is not marked as "no longer necessary". The instance method *choice* groups multiple alternatives. The decision logic is implemented in the instance method *alternative*. A choice contains one or more alternatives and an optional block if none of the alternatives are chosen. The procedure is as follows: *choice* yields the code block in a separated thread environment. The instance method *alternative* is called with a boolean as argument. When the DSL is interpreted, the boolean is evaluated out of the condition. If the boolean is true, the code block of *alternative* is yielded and the thread environment is flagged with

Thread.current[:alternative_executed] = true. If the code block of *choice* contains an *otherwise*, this method is called when the code block is yielded. The instance method *otherwise* checks if the thread environment is flagged with *:alternative_executed*. If not, it yields the given code block. The thread is only forked to create a separated environment for each choice block. Nested choices cannot interfere each other as each *alternative* and *otherwise* refers to its thread environment that belongs to the choice block.

cycle.

- *cycle(*"condition"*)* &codeblock

The instance method *cycle* repetitive yields a given code block as long as a given condition is true. Other than for *alternative* the result of the condition can change over time. Therefore it is not sufficient to write the condition directly in the DSL. The result of this would be that the condition is evaluated by the ruby interpreter and the **result** is given to the instance method, NOT the condition. To have the possibility to evaluate the condition itself, the condition has to be provided as a string. This enables to evaluate the string with the use of *eval*. If this results true, the code block is yielded.

critical.

- *critical*[*(:*identifier*)*] &codeblock

The instance method *critical* creates a mutex and yields the given block within that mutex. An optional argument enables to select a specific mutex. Each mutex is stored in a hash to enable critical section that span different code blocks.

5.5.4 Elements from the Controllers perspective

context. As already mentioned above, context is defined as class AND instance method. From the controllers perspective, only the instance method is applicable. In addition to the already described behavior, the method stores each defined context variable in a hash. *context* returns this hash each time it is called. The hash is updated each time an activity may have altered one of the context variables.

replace.

- *replace* &codeblock

In the initial workflow definition, the workflow description is specified through the class method *control* as described above. When the workflow class is already instantiated, the workflow description can be replaced with the instance method *replace*. The original instance method *__wee_execute* is replaced as described in section 5.5.2.

start. This instance method orders the WEE to execute the workflow description. As already described, the workflow description is stored as an instance method called *__wee_execute*. To start the workflow execution, it is only necessary to invoke this method.

stop. A stop orders the workflow execution to be aborted. This instance method does not interrupt the execution directly, but sets a stop flag which is checked in each DSL element. If the flag is set, the DSL elements return without execute their workflow

logic. The *activity* checks for this flag not only directly after the invocation but also during a running service call. The exact behavior has already been described in the previous section. However, the flag is also checked every time a code block should be yielded, therefore it is recognized in almost every DSL-element.

search.

- *search* [*new Wee::Searchposition(:*position*, :at*|*:after*[,passthrough]*)*]+

  This instance methods sets the starting positions where the execution has to start from when *start* is called the next time. Basically, the starting positions are stored in an instance variable. The search logic itself is placed in the DSL-elements of the workflow description. The next section will investigate the implementation in more detail.

5.5.5 Search procedure

A main feature of the WEE is to start the execution of a workflow description at a specified point. As already mentioned in section 5.2, this is not directly possible, therefore the behavior is emulated. The controller sets the starting positions by calling the instance method *search*. A search-flag (*@__wee_search*) is set to indicate that the workflow execution has to be performed in search mode first. For each invocation of *activity*, a check is performed if this position is a start position. If this is true, the search-flag is set to *false* to indicate that the following activities are not omitted due to search reasons. During the execution in the search mode, all code paths have to be executed until the starting position is found. Therefore, the search mode affect not only the *activity*-method:

- Choice. While in search mode, each alternative must be executed as a starting position can be located in any of the branches. Also, the *otherwise*-branch must be executed. The search procedure is one of the reasons why we do not rely on the *if*-condition the Ruby programming language provides. The WEE would not be able to control the control flow if decisions would be made by Ruby.
- Parallel. Start positions can be located in parallel branches. If this is the case, a start position for each parallel executed branch must be provided. The state of the search-flag must be processed separated for each parallel branch. Every time a new thread is forked, the actual state of the search-flag is set for this thread (*Thread.current[:branch_search] = @__wee_search*). Each activity within this thread has to check this thread-local flag only and ignores the overall search flag. If an activity within this thread is a starting position, it deactivates the overall search flag and the search flag of this thread (*Thread.current[:branch_search] = @__wee_search = false*). This ensures that a parallel branch (in a concurrent thread) which has not discovered its starting position is not influenced and the branches are still separated. The correct setting of the overall search flag is necessary to correctly perform the subsequent branch after the parallel block.
- Cycle. The code block of a cycle-element must be performed at least once during the search procedure. Therefore, the condition is ignored if the search mode is active. If the starting position is within this code block, the search flag is set to *false*. After the code block is successfully executed, the condition of the cycle is evaluated only if the workflow is no longer in search mode.

Chapter 6

Workflow Execution in a cloud environment [1]

Cloud environments foster the "Software as a Service" [30] paradigm, in providing easy access to replaceable pieces of infrastructure, like storage, CPU power or services implementing arbitrary functionality. When using these pieces of infrastructure in a business environment, they become part of workflows, which are handled by workflow engines [18, 31]. Workflow engines are specialized in orchestrating business processes consisting of multiple activities, these activities are commonly realized as web services. We believe that a cloud is the perfect execution environment for this kind of activities.

In this section we explain how we envision workflow execution in a cloud environment by moving the workflow engine itself to the cloud. Enabling dynamic workflows addresses topics that come up in the cloud:

- In contrast to traditional workflow engines, there is no control over the availability of services, so they have to be replaceable at runtime.
- Integrating services from external vendors is different from orchestrating a workflow in that important information about an activity is hidden inside the external service. In case of a problem it is desirable to involve the external service in the repair process, because it has access to problem specific information.

Modularizing and moving the workflow engine to the cloud has the goal to standardize ways of interaction between the workflow engine and cloud services. Modularizing the workflow engine helps to clarify security aspects. Security in cloud based systems basically means trusting service providers [32]. Having separated contexts with well defined ways to access/modify information will provide starting points for security mechanisms.

In a cloud based environment where many providers of services exist, it is a necessity that a workflow engine has to handle different tasks in a dynamic manner as compared to traditional business or scientific scenarios that workflow engines covered so far.

With the ability to have external handlers for monitoring, repair and other tasks, several

[1] This section is a modified and extended version of different sections in: G. Stürmer, J. Mangler, and E. Schikuta, "Network Based Execution of Dynamic Workflows in Grid and Cloud Based Environments" 3rd Austrian Grid Symposium, September 28-29, 2009, Linz, Austria.

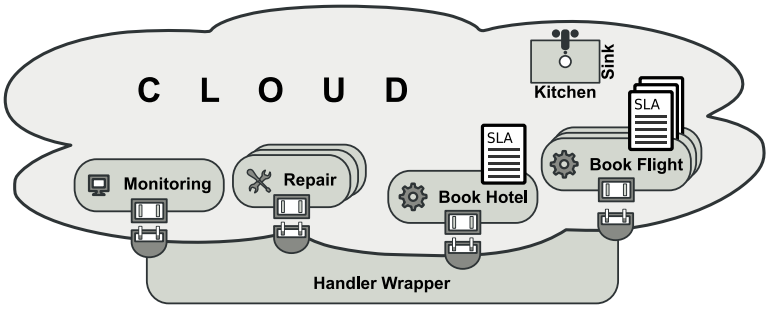

Figure 6.1: Workflow Execution in a Cloud Environment

interesting topics come up. **Our vision** is to not only supply services to customers, but also repair handlers and monitors. These handlers would have the potential to react much quicker and much more precise on upcoming issues as they are not coupled with the workflow exposed by the customer but also know about the otherwise hidden context of the supplier. Also of interest can be custom repair algorithms, applied in conjunction with runtime information rather than specified at design time. Figure 6.1 shows how multiple services by multiple vendors could be used and tied together as handlers, included in a workflow through the handler wrapper.

A customer provides the workflow description that has to be executed by an execution engine. The execution engine itself may choose services and handlers according to the needs of the customer, specified by SLAs.

Also of interest for cloud based environments will be the security aspect. Although a handler provided by a supplier can use information from the supplier without exposing information to the outside world, interfering with the workflow requires a certain level of trust from the consumers side, which can be addressed by SLAs.

6.1 Dist-WEE

In the development of the WEE, the claim for a service oriented architecture was self-evident. Workflow execution is done by the collaboration of different services, some of them as business services defined in the workflow description, others as repair or monitoring services to carry out tasks of the workflow execution engine. In this section we introduce Dist-WEE (Distributed WEE), a RESTful interface to interact with WEE-workflows.

6.1.1 Overview

The interfaces defined in Section 4.3 can be exposed to the services through different technologies. In a first attempt, we created a RESTful interface [33] that encapsulated the WEE and the handler wrapper. The composition is depicted in Fig. 6.2. Services like monitoring and repair can access the WEE through the Dist-WEE. Possible user interfaces or other consumers of workflows can spawn workflow instances and track the progress through the same mechanisms. For demonstration purposes, we created a prototype of such a user interface (described in more detail in section 6.1.3) to simulate the access of a repair service. The

user interface usually will have an embedded monitoring facility to supervise the execution process. The Dist-WEE receives the requests from the different services and forwards them to the WEE interfaces. Multiple workflow instances (and therefore multiple WEE-instances) can be accessed through the Dist-WEE interface. When the WEE executes a workflow, it can access external business services (e.g. hotel or flight booking services) through the handler wrapper. These service calls can be exposed also through a RESTful interface, but may also be accessible through SOAP or other technologies.

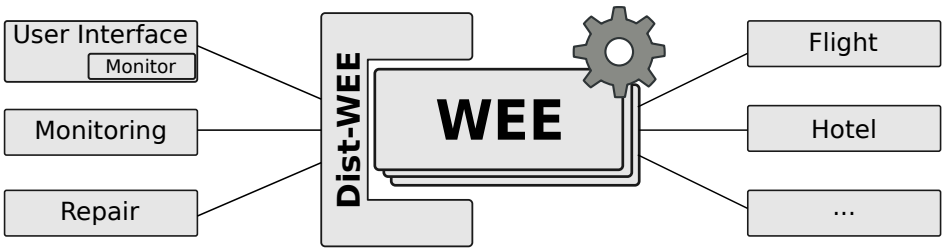

Figure 6.2: Outline of Dist-WEE

6.1.2 Resources and Interface mappings

In this section, we will describe the REST-interface in detail. We will elaborate for each available resource which methods and parameters are available and to which interface they appeal.

Resource "/": The root resource of the Dist-WEE. Calls to this resource concern the management of the different WEE instances.

- POST Creates a new and empty WEE-instance. A new WEE-instance has no predefined context variables, endpoints or workflow description. A user of the Dist-WEE therefore creates an empty template of a workflow and has to define the workflow through the properties interface exposed as described below.
  - request: *instance-name* - An optional name for the WEE-instance.
  - response: An unique identifier for the generated WEE-instance.
- GET Fetches the list of existing WEE-instance.
  - request: no parameters.
  - response: *list-of-workflow-instances*: A XML document containing the unique identifiers and names of all WEE-instances that are managed through the Dist-WEE. Listing 6.1 shows an example of such a XML document. The root-node (*div*) contains a number of anchor-nodes. Each anchor represents a WEE-instance. The href-attribute contains the identifier of the WEE-instance, the value of the node contains the optional name. In this case, two WEE-instances are available. A instance (id = 1) named "examplewf1" and a second instance (id = 2) named "examplewf2". Using this definition allows to make this result not only machine-readable but also enables to display it in a browser (the style-attribute is a consequence of this dual usage).

LISTING 6.1: An example output of type *list-of-workflow-instances*

```
1 <div>
2   <a style="display:block" href="1">examplewf1</a>
3   <a style="display:block" href="2">examplewf2</a>
4 </div>
```

- DELETE Removes a WEE-instance completely. The WEE instance can only be deleted if the state is *finished* or *stopped*. Until the instance is removed, it can be changed or started again. If the instance is removed, it cannot be accessed further.
  - request: *instance-id* - The unique identifier of the WEE that should be removed.
  - response: none.

Resource "/{*instance-id*}/": A dynamic and abstract resource representing a WEE-instance. In this case, the actual identifier of the instance replaces {*instance-id*}. The resource itself cannot be accessed but contains the sub-resources state and properties.

Resource "/{*instance-id*}/state/": Represents the state of the instance. Methods applied to this resource query or change the state. This resource accesses the intervention interface described in section 4.3.2.

- GET Queries the actual state of the workflow execution.
  - request: no parameters.
  - response: The state of the workflow execution. The response value must be *ready*, *running*, *stopped*, or *finished*
- PUT Changes the state of the WEE-instance.
  - request: *control-message* - Orders to start or stop the execution of the instance. Possible parameter values are therefore *start* or *stop*.
  - response: *state* - the new state of the workflow execution. Possible response values are same as for GET.
- PUT Sets the starting positions.
  - request: *search-pos* - Sets the starting position for the WEE-instance. The next time this instance is started, it starts the execution from the given position. The parameter consists of three elements: the position identifier, an identifier the execution has to start if *at* or *after* this position and a *passthrough* value. Until now, Dist-WEE only supports to set a single starting position. Starting within a parallel branch is not implemented.
  - response: none.

Resource "/{*instance-id*}/properties/": An abstract resource that encapsulates the different properties of the workflow instance (context, endpoints and workflow description). This resource accesses the properties interface described in section 4.3.3. Only sub-resources of this resource can be accessed.

Resource "/{*instance-id*}/properties/context/": Gains access to the context of the workflow instance.

- GET Queries the workflow instance for context variable names.
  - request: no parameters.
  - response: *list-of-context-vars* - A XML document containing the context variable names of the workflow instance. Listing 6.2 shows an example of such a XML document similar to the instances-list mentioned above. The root-node (*div*) contains a number of anchor-nodes. Each anchor represents a context variable. The href-attribute contains the name of the context variable, the value of the node also contains the variable name. This makes it better readable when displayed in a browser.

  LISTING 6.2: An example output of type *list-of-context-vars*

```
1 <div>
2   <a style="display:block" href="x">x</a>
3 </div>
```

- POST Adds a new context variable
  - request: *context-pair* - Name and initial value of the new context variable.
  - response: *context-id* - The name of the new context variable.

Resource "/{*instance-id*}/properties/context/{*context-id*}/": A dynamic resource that gains access to the value of a specific context variable, identified by the value of {context-id}.

- GET Queries the value of the context variable.
  - request: no parameters.
  - response: *context-value* - The value of the context variable.
- PUT Sets the value of the context variable to a given value.
  - request: *context-value* - The new value of the context variable.
  - response: none.
- DELETE Removes the context variable from the workflow context.
  - request: no parameters.
  - response: none.

Resource "/{*instance-id*}/properties/endpoints/": Gains access to the endpoints of the workflow instance.

- GET Queries the workflow instance for available endpoints.
  - request: no parameters.
  - response: *list-of-endpoint-ids* - A XML document containing the endpoint ids of the workflow instance. Listing 6.3 shows an example of such a XML document similar to the context-list mentioned above. The root-node (*div*) contains a number of anchor-nodes. Each anchor represents an endpoint. The href-attribute contains the ID of the endpoint, the value of the node also

contains the variable ID. This makes it better readable when displayed in a browser.

LISTING 6.3: An example output of type *list-of-endpoint-ids*

```
1 <div>
2   <a style="display:block" href="flight">flight</a>
3 </div>
```

- POST Adds a new endpoint.
  - request: *endpoint-pair* - ID and initial value of the new endpoint.
  - response: *endpoint-id* - The ID of the new endpoint.

Resource "/{*instance-id*}/properties/endpoints/{*endpoint-id*}/": A dynamic resource that gains access to the URI of a specific endpoint, identified by the value of {endpoint-id}.

- GET Queries the value of the endpoint.
  - request: no parameters.
  - response: *endpoint-value* - The URI of the endpoint.
- PUT Sets the value of a endpoint to a given URI.
  - request: *endpoint-value* - The new URI of the endpoint.
  - response: none.
- DELETE Removes the endpoint from the workflow instance.
  - request: no parameters.
  - response: none.

Resource "/{*instance-id*}/properties/description/": Accesses to the workflow description of this instance.

- GET Queries the workflow description.
  - request: no parameters.
  - response: *description* - The workflow description of the instance.
- PUT Sets the workflow description of the instance.
  - request: *description* - The new workflow description.
  - response: none

Resource "/{*instance-id*}/properties/handlers/": Queries and accesses the handlers that take part on the workflow execution. Basicall, these handlers are intended to provide workflow engine logic on behalf of the handler wrapper, like monitoring or repair logic. As our prototype handler wrapper is not designed to manage these handlers, this resource is implemented to replace or query the handler wrapper itself with another version.

Resource "/{*instance-id*}/properties/handlers/{*handler-type*}/": The Dist-WEE is designed to accept arbitrary handlers categorized by a system that has to be understood only by the handler wrapper. Therefore, the *handler-type* can be chosen freely. In our prototype implementation, the handler-type has no impact. However, each handler wrapper is identified by its class and can be provided with an optional argument which is provided for each instantiation.

- GET Queries the handler wrapper and its argument
  - request: no parameters.
  - response: *handler-pair* - The class of the handler wrapper and the argument (as a string).
- PUT Sets the handler wrapper of the instance.
  - request: *handler-pair* - The class of the handler wrapper and an optional argument (as string). The class of the handler wrapper must be available for the WEE-instance.
  - response: none.

6.1.3 User Interface

To test the Dist-WEE interface, we developed a prototype user interface. The UI serves as a tool to carry out tasks that monitoring and repair services may carry out automatically. The UI has an attached monitor to receive status changes and monitoring information. In this section, we describe the setup of this UI and its components.

The UI has two screens: an overview screen of all existing instances and a detail screen for a specific instance. The UI itself is embedded in a RESTful architecture. The screens are provided as normal HTML-pages trough the REST interface. They fetch the needed informations through a set of AJAX requests [34] as depicted in Fig. 6.3. The UI uses a adopted version of the handler wrapper which forwards the monitoring information to the REST interface of the UI. Therefore the AJAX requests are sent to the Dist-WEE interface for informations about instances or to the UI RESTful interface to fetch monitoring information.

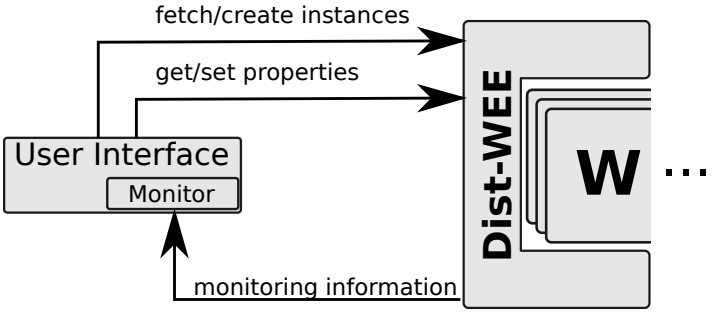

Figure 6.3: Communication between Dist-WEE and UI

Overview A screenshot of overview page can be seen in Fig. 6.4. The upper region lists created instances and enables the generation of a new instance. To fetch the list of instances, an AJAX request with the method GET is send to the resource "/" of the

Dist-WEE. To create a new instance, a POST request is send to the "/" resource. For each new instance, the handler wrapper is set to the adapted version of the UI. This handler wrapper expects a URI as argument when it is instantiated. The URI refers to the REST interface of the UI, the monitoring messages will be sent to the resource "/{*instance-id*}/monitor/" with a POST method. The rest of the page contains the monitoring messages of all instances. The UI fetches this information with an AJAX-request on its own REST interface with a GET method to the resource "/monitor/". This resource contains the combined monitoring information of all instances.

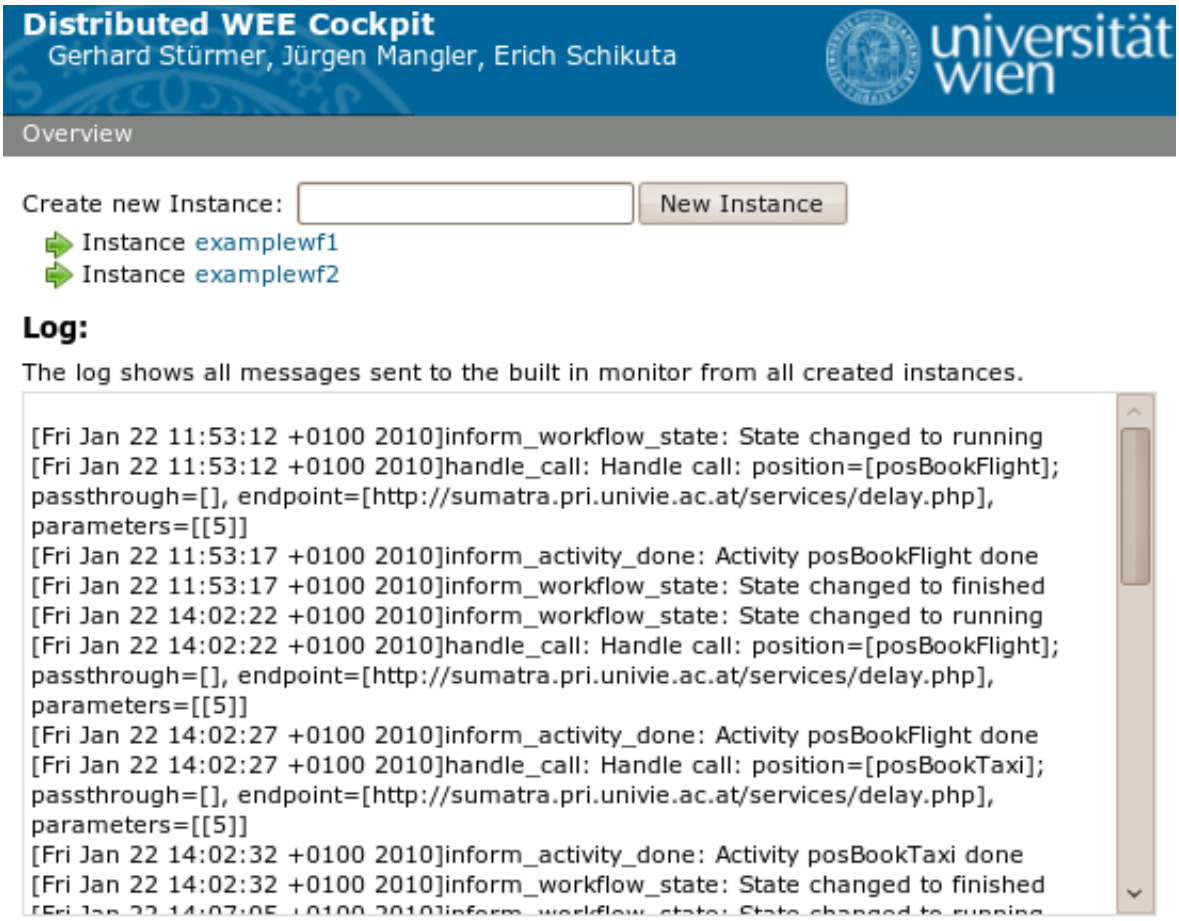

Figure 6.4: Screenshot of the overview page

Details The details page shows informations about a specific workflow instance as depicted in Fig. 6.5. When the details page is displayed, it fetches the information about the instance with a set of AJAX-requests to the properties and state resource of the Dist-WEE. Changes to the properties are applied when the button on the bottom is pressed. The details page then sends the according POST or PUT requests to the properties interface. In our example, the workflow has an attached monitor which refers to our user interface as described above. When the workflow execution is started, the details page sets the input fields to read only and sends a PUT with the "start" control-message to the Dist-WEE. Then the UI frequently polls the properties interface and updates the displayed informations. The actual position of the thread of control can be extract from the monitoring information, the REST interface of the UI provides

this information with a GET on the resource "/{*instance-id*}/monitor/actpos". This position is displayed on the details page as shown in Fig. 6.6. After the execution is finished or the user stops the execution, the properties are refreshed and the input fields are enabled again.

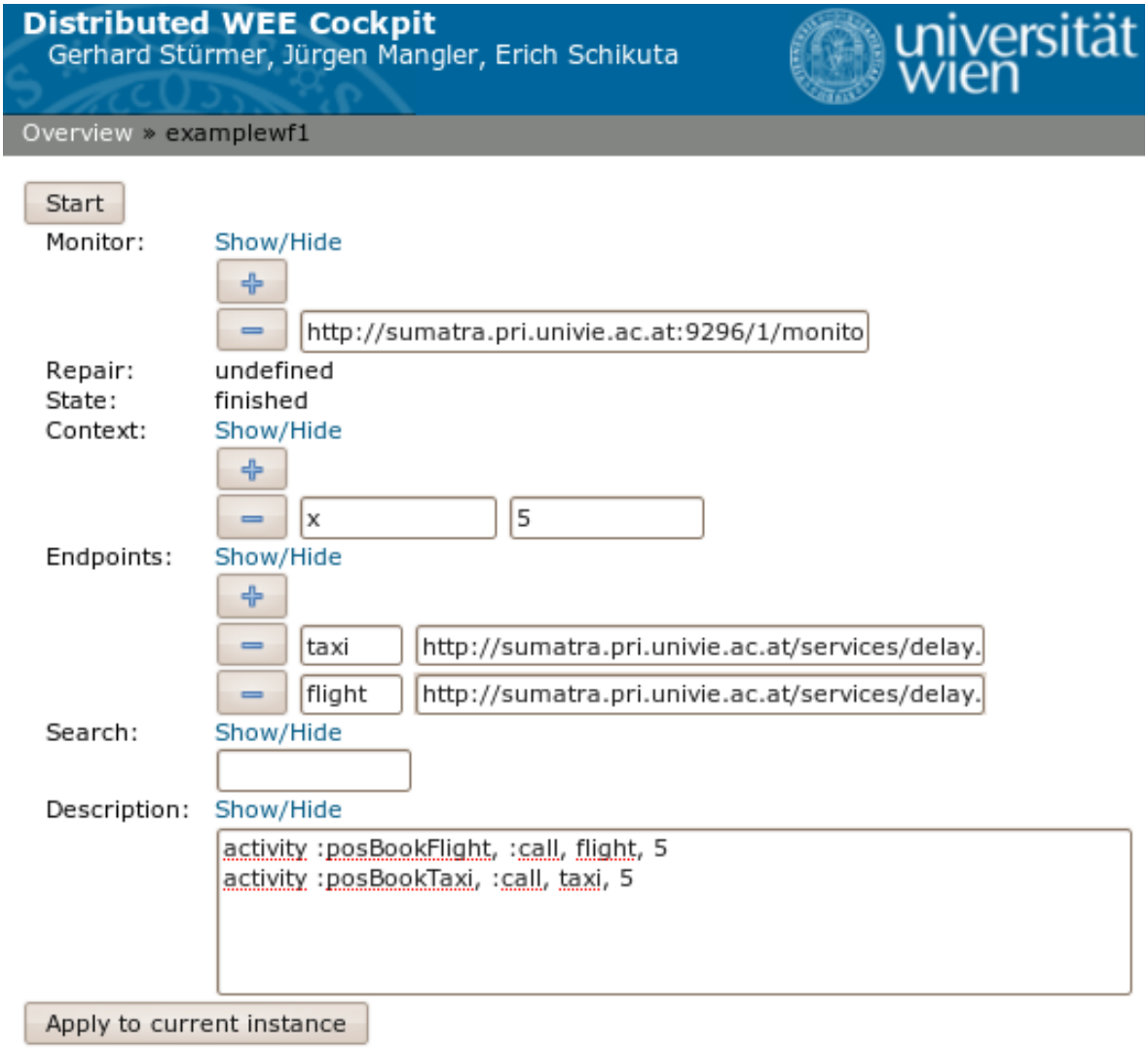

Figure 6.5: Screenshot of the details page

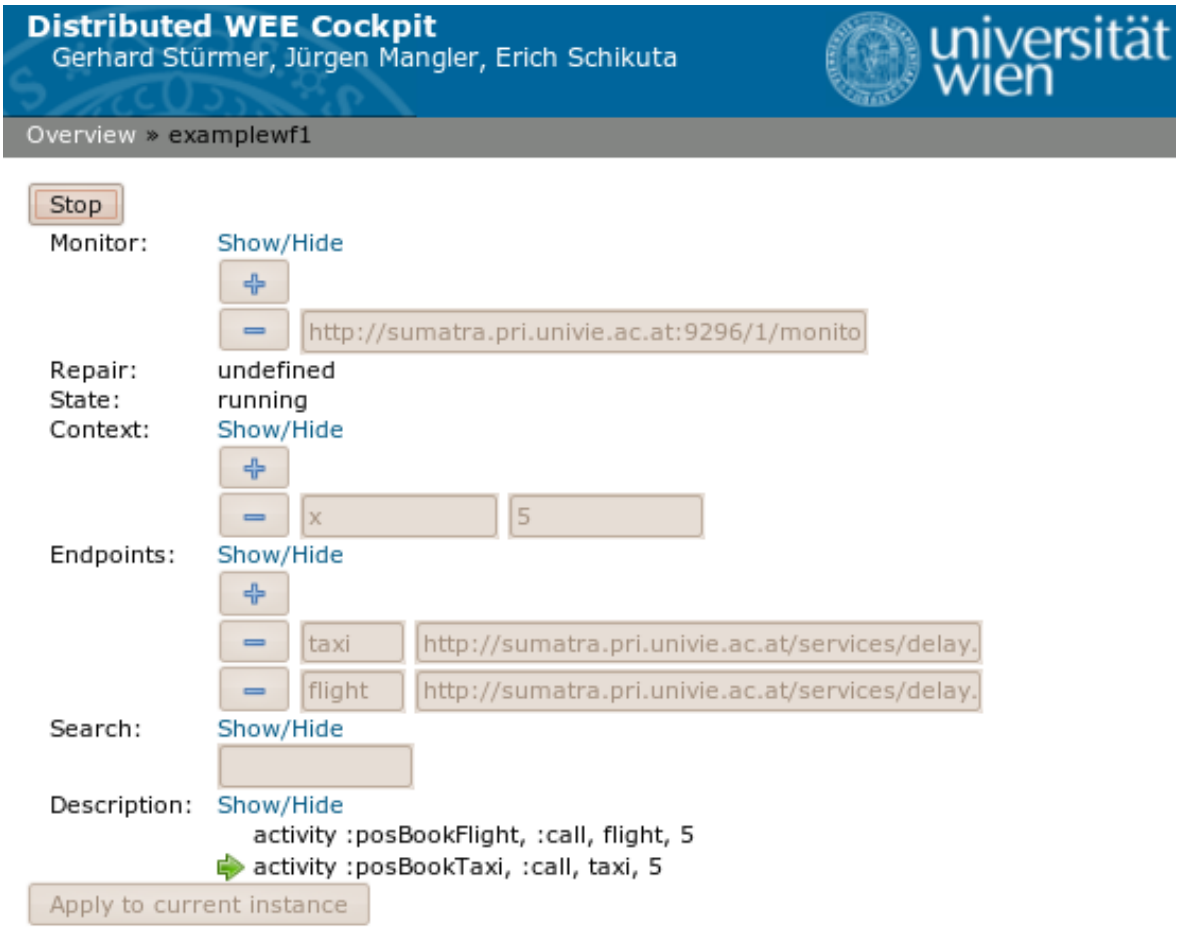

Figure 6.6: The pending activity is indicated on the details page.

Chapter 7

Conclusion

In this thesis, we presented the idea of dynamic workflows. Dynamic workflows allow for supervision and intervention of workflow execution from external services. This enables for the outsourcing of modules that were until now integrated into the workflow engine. We created a layered architecture that covers the idea of dynamic workflows and utilizes external services (a.k.a. handlers) to not only cover business logic requested in the workflow description, but also carry out logic which was hard-wired in traditional workflow engines. The architecture enables monitoring and repair and is designed to be open to other aspects like security or resource allocation. A handler wrapper integrates the different external handlers into the workflow execution. The handler wrapper coordinates the communication between the handlers and the workflow execution engine.

The execution of a workflow is carried out by a network of external services that interacts with the workflow execution engine. This network based execution is not a condition for dynamic workflows, but a direct consequence. The concept of dynamic workflows enables the outsourcing of functionality from the workflow engine to external modules. This allows to easily embed services from external providers into the workflow execution and to integrate the system into a service oriented architecture.

A prototype implementation of a workflow execution engine demonstrates the feasibility of our approach. We designed a domain specific language with a set of elements to describe workflows and evaluated the DSL by comparing with common control flow patterns for workflows. This allowed an evaluation of the completeness of the conceptual DSL. We also illustrated in detail how we implemented the prototype in the Ruby programming language with the help of its metaprogramming facilities.

In the last section, we outlined how to deploy our architecture in a cloud environment. The possibility to integrate external services like repair algorithms or specialized monitors may encourage providers to expand their portfolio in these sections. To demonstrate this integration, we introduced Dist-WEE, a RESTful interface for our prototype and demonstrated its usability with a custom user interface.

7.1 Contribution of this work

In recent years, adaptive workflows have been investigated as pointed out in section 2. They deal with issues of workflow evolution and ad-hoc changes to workflows. But present systems restrict the possible changes whereas dynamic workflows persue the concept that almost everything is changeable. These changes can be ordered by an external service and do not need to be originated from a monolithic workflow engine or the workflow description. The changes can affect the environment of the workflow (like context variables or endpoint) as well as the workflow description itself. Where present systems allow only specific changes to the workflow description (e.g. on the basis of transformation patterns), dynamic workflows allow to modify the workflow description without restrictions.

Consequentially, this enables to externalize components which were until now firmly built into workflow engines. Monitoring tools can now supervise the execution from outside as they have access to the workflow context. Repair algorithms do not need to be an integral part as they do not need to obey extensive constraints about which changes are allowed.

We therefore expect that existing repair-strategies can be migrated or realized for Dist-WEE. Dist-WEE is developed to provide a playground for new algorithms or for the adoption of existing algorithms.

7.2 Further research

The prototype implementation does not exceed 400 lines of code and is written to be compact and easy to use. We want to foster the exploration of external services that should be embedded into the workflow execution. The architecture is designed to allow radical interventions into the workflow execution. Therefore the areas of future research are manifold. The following future research questions arise:

- How can security aspects be integrated in such an environment? Which rights and liabilities do external handlers come with? What happens when the competences (e.g. repair competences) of a set of external handlers overlap?
- Which different types of handlers can be identified and outsourced from the workflow engine?
- What repair algorithms or strategies exist that can be enabled through dynamic workflows?

We believe that dynamic workflows are worth to be investigated in more detail. We hope that our prototype can inspire researchers to try out new algorithms and facilitates research in this area.

Bibliography

[1] D. Ardagna, M. Comuzzi, E. Mussi, B. Pernici, and P. Plebani, "PAWS: a framework for executing adaptive Web-Service processes," *IEEE Software*, vol. 24, no. 6, p. 39–46, 2007.

[2] E. Schikuta, H. Wanek, and I. U. Haq, "Grid workflow optimization regarding dynamically changing resources and conditions," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 15, 2008.

[3] "OASIS web services business process execution language (WSBPEL) TC." http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel, 2007. [Last accessed: 31.01.2010].

[4] S. Modafferi, E. Mussi, and B. Pernici, "SH-BPEL: a self-healing plug-in for Ws-BPEL engines," in *Proceedings of the 1st workshop on Middleware for Service Oriented Computing (MW4SOC 2006)*, p. 48–53, ACM New York, NY, USA, 2006.

[5] M. Adams, A. H. M. ter Hofstede, D. Edmond, and W. M. P. van der Aalst, "Facilitating flexibility and dynamic exception handling in workflows through worklets," in *Proceedings of the CAiSE*, vol. 5, p. 45–50, 2005.

[6] D. Georgakopoulos, M. Hornick, and A. Sheth, "An overview of workflow management: From process modeling to workflow automation infrastructure," *Distributed and parallel Databases*, vol. 3, no. 2, p. 119–153, 1995.

[7] P. Soffer, "On the notion of flexibility in business processes," in *Proceedings of the CAiSE*, vol. 5, p. 35–42, 2005.

[8] K. Ploesser, J. Recker, and M. Rosemann, "Towards a classification and lifecycle of business process change," *Proceedings of the 9th Workshop on Business Process Modeling, Development and Support. CEUR Workshop Proceedings*, vol. 335, p. 11, 2008.

[9] C. Ellis, K. Keddara, and G. Rozenberg, "Dynamic change within workflow systems," in *Proceedings of conference on Organizational computing systems*, p. 10–21, 1995.

[10] S. Rinderle, M. Reichert, and P. Dadam, "Correctness criteria for dynamic changes in workflow systems—-a survey," *Data & Knowledge Engineering*, vol. 50, no. 1, p. 9–34, 2004.

[11] M. Kamath and K. Ramamritham, "Correctness issues in workflow management," *Distributed Systems Engineering*, vol. 3, p. 213–221, 1996.

[12] W. M. van der Aalst, T. Basten, H. M. W. Verbeek, P. A. Verkoulen, and M. Voorhoeve, "Adaptive workflow: On the interplay between flexibility and support," *Enterprise Information Systems*, p. 63–70, 2000.

[13] P. J. Kammer, G. A. Bolcer, R. N. Taylor, A. S. Hitomi, and M. Bergman, "Techniques for supporting dynamic and adaptive workflow," *Computer Supported Cooperative Work (CSCW)*, vol. 9, no. 3, p. 269–292, 2000.

[14] D. Domingos, A. Rito-Silva, and P. Veiga, "Authorization and access control in adaptive workflows," *Lecture Notes in Computer Science*, vol. 2808, p. 23–38, 2003.

[15] M. Reichert and P. Dadam, "ADEPT flex—supporting dynamic changes of workflows without losing control," *Journal of Intelligent Information Systems*, vol. 10, no. 2, p. 93–129, 1998.

[16] P. A. Buhler and J. M. Vidal, "Towards adaptive workflow enactment using multiagent systems," *Information Technology and Management*, vol. 6, no. 1, p. 61–87, 2005.

[17] L. Baresi, S. Guinea, and L. Pasquale, "Self-healing BPEL processes with dynamo and the JBoss rule engine," in *International workshop on Engineering of software services for pervasive environments: in conjunction with the 6th ESEC/FSE joint meeting*, p. 20, 2007.

[18] L. L. C. ActiveBPEL, "ActiveBPEL-BPEL execution engine." http://www.activebpel.org/, 2009. [Last accessed: 31.01.2010].

[19] "JBoss - business rules management system." http://www.jboss.com/products/platforms/brms/, 2010. [Last Accessed: 31.01.2010].

[20] C. Hagen and G. Alonso, "Exception handling in workflow management systems," *IEEE Transactions on software engineering*, vol. 26, no. 10, p. 943–958, 2000.

[21] J. Eder and A. Tahamtan, "Temporal consistency of view based interorganizational workflows," *Proc. of UNISCON*, vol. 8, 2008.

[22] M. Bichier and K. Lin, "Service-oriented computing," *Computer*, vol. 39, no. 3, p. 99–101, 2006.

[23] S. W. Sadiq, O. Marjanovic, and M. E. Orlowska, "Managing change and time in dynamic workflow processes," *International Journal of Cooperative Information Systems*, vol. 9, no. 1-2, p. 93–116, 2000.

[24] J. Eder, J. Mangler, E. Mussi, and P. Pernici, "Using stateful activities to facilitate monitoring and repair in workflow choreographies," in *International Workshop on Self Healing Web Services*, (Los Angeles, CA, USA), 2009.

[25] W. M. P. V. der Aalst and A. H. M. T. Hofstede, "YAWL: yet another workflow language," *Information Systems*, vol. 30, no. 4, p. 245–275, 2005.

[26] W. M. Coalition, "XPDL support and resources." http://www.wfmc.org/xpdl.html, 2009. [Last Accessed: 31.01.2010].

[27] "Ruby." http://www.ruby-lang.org/en/about/, 2009. [Last Accessed: 31.01.2010].

[28] W. M. P. V. D. V. der Aalst, A. H. M. T. Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns," *Distributed and Parallel Databases*, vol. 14, no. 1, p. 5–51, 2003.

[29] P. Wohed, N. Russell, A. H. M. ter Hofstede, B. Andersson, and W. M. P. van der Aalst, "Patterns-based evaluation of open source BPM systems: The cases of jBPM, OpenWFE, and enhydra shark," tech. rep., BPM Center Report BPM-07-12, BPMcenter.org, 2007.

[30] M. Turner, D. Budgen, and P. Brereton, "Turning software into a service," *Computer*, vol. 36, no. 10, p. 38–44, 2003.

[31] Oracle, "Oracle process manager." http://www.oracle.com/technology/products/ias/bpel/index.html, 2009. [Last accessed: 31.01.2010].

[32] B. Schneier, "Cloud computing." http://www.schneier.com/blog/archives/2009/06/cloud_computing.html, 2009. [Last Accessed: 31.01.2010].

[33] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Transactions on Internet Technology (TOIT)*, vol. 2, no. 2, p. 115–150, 2002.

[34] J. J. Garrett and D. Ajax, "Ajax: A new approach to web applications," *Nine*, 2007.

List of Figures

4.1 The WEE Architecture 14
4.2 A Simple Flight / Hotel Booking Example 16
4.3 WEE Interfaces (class diagram) 17
4.4 A generic example of a service call 18
4.5 Example usage of the passthrough value 20

5.1 Cancel Region Pattern 40
5.2 Arbitrary Cycles Pattern Example 42
5.3 Arbitrary Cycles Pattern Implementation 42
5.4 Execution of an activity 51
5.5 Code block arrangement of parallel branches 53

6.1 Workflow Execution in a Cloud Environment 58
6.2 Outline of Dist-WEE 59
6.3 Communication between Dist-WEE and UI 63
6.4 Screenshot of the overview page 64
6.5 Screenshot of the details page 65
6.6 The pending activity is indicated on the details page. 66

List of Tables

2.1 Types of Exceptions by [13] and their impact to workflows 6

5.1 Workflow Patterns Coverage 43
5.2 Support of Control Flow Patterns 45

Listings

5.1 Example Workflow . . . 25
5.2 A Simple Handler Wrapper . . . 27
5.3 Example of Workflow Restructuring . . . 27
5.4 Endpoint and Context Manipulation . . . 28
5.5 Sequence pattern . . . 29
5.6 Parallel split & Synchronization . . . 29
5.7 Exclusive choice & Simple Merge . . . 30
5.8 Multi-Choice & Structured Synchronizing Merge - Not concurrent 30
5.9 Multi-Choice & Structured Synchronizing Merge - Concurrent . 31
5.10 Canceling Discriminator . . . 31
5.11 Thread-Split & Thread-Merge . . . 32
5.12 Deferred Choice . . . 33
5.13 Milestone . . . 34
5.14 Critical Section . . . 34
5.15 Interleaved Routing . . . 35
5.16 Interleaved Parallel Routing . . . 35
5.17 Multiple Instances with a Priori Run-Time Knowledge . . . 37
5.18 Multiple Instances without a Priori Run-Time Knowledge . . . 37
5.19 Canceling Partial Join for Multiple Instances . . . 38
5.20 Explicit Termination . . . 40
5.21 Structured Loop . . . 41
5.22 A Basic Example of a Workflow . . . 46
5.23 Implementation of the class method *endpoint* . . . 48
5.24 Implementation of the class method *context* . . . 48
5.25 Replace a method body with the instance variable approach . . . 49
5.26 Replace a method body with the Eigenclass approach . . . 49
6.1 An example output of type *list-of-workflow-instances* . . . 60
6.2 An example output of type *list-of-context-vars* . . . 61
6.3 An example output of type *list-of-endpoint-ids* . . . 62

Appendix A

Abstract

Workflow engines deal with the execution of a business process specified by a process description. A process description contains a set of activities that realize a business objective or policy goal. Current workflow engines rely on the workflow designer to include error handling logic for all problems that occur during execution. Whenever this is not sufficient, the workflow engine applies predefined recovery or modification strategies. These strategies are part of the workflow engine itself or realized as a plugin. Adaptive workflow engines allow to apply specific modifications to the workflow environment or the process descriptions during execution with a set of APIs.

This thesis introduces a concept called "Dynamic Workflows" to overcome the static and inflexible APIs provided by these workflow engines. We propose an architecture that executes a workflow through a collaboration with external services. The core of our architecture is a workflow execution engine that concentrates on the control flow aspects. Other facets like monitoring or repair are handled by autonomous services. The services can intervene into the workflow execution through interfaces. Other than in current workflow engines, we pursue to provide a high degree of latitude. External services are able to modify the environment and the process description ad libitum.

The modularization fosters not only the integration in distributed environments (like clouds, service oriented architectures) but also enables to customize or replace components, that are until now part of the workflow engine.

To demonstrate the feasibility of our architecture, we also demonstrate a prototype implementation that is accessible through a RESTful API.

Appendix B

Kurzzusammenfassung

Workflow Engines dienen der Ausführung von Geschäftsprozesse welche durch eine Prozessbeschreibung definiert werden. Eine Prozessbeschreibung besteht aus mehreren Aktivitäten und dient der Erreichung eines bestimmten Zieles oder Zustandes. Aktuelle Workflow Engines vertrauen darauf, dass sich der Ersteller der Prozessbeschreibung um alle nötigen Fehlerbehandlungen kümmert welche während der Ausführung gebraucht werden könnten. Ist dies nicht ausreichend, können vordefinierte Reperaturstrategien angewendet werden. Diese Strategien sind Bestandteil der Workflow Engine selbst oder werden durch ein Plugin zur Verfügung gestellt. Adaptive Workflow Engines erlauben während der Ausführung bestimmte Modifikation an der Workflowumgebung oder an der Prozessbeschreibung, wobei diese Modifikationen durch ein vordefiniertes Set an Möglichkeiten limitiert werden.

In dieser Masterarbeit wird das Konzept der "Dynamic Workflows" vorgestellt, welche die starren APIs aktueller Workflow Engines überwinden sollen. Wir schlagen dabei eine Architektur vor, die Workflows durch die Kollaboration von mehreren externen Services abarbeitet. Im Zentrum steht dabei die Workflow Execution Engine welche sich auf die Kontrollflußaspekte konzentriert. Andere Aufgaben wie z.b. Monitoring oder Reperatur werden von autonomen Services durchgeführt die in die Ausführung des Workflows mithilfe von Schnittstellen eingreifen. Anders als in traditionellen Workflow Engines streben wir dabei einen hohen Freiheitsgrad an was die Möglichkeiten betrifft die diese Schnittstellen zur Verfügung stellen sollen. Externe Services können sowohl die Ausführungsumgebung als auch die Prozessbeschreibung nach Belieben verändern.

Die von uns vorschlagene Modularisierung fördert nicht nur die Integration in verteilten Systemen (wie z.b. Clouds), sondern ermöglichen es auch die diveresen Komponenten an die eigenen Bedürfnisse anzupassen oder zu ersetzen.

Die Umsetzbarkeit unserer Architektur wird anhand einer Prototyp-Implementierung, welche über ein REST API verfügt, demonstriert.

Appendix C

Curriculum Vitae

Gerhard Stürmer

Geboren am 17.01.1982 in Horn

Ausbildung

2003-2007	Bachelorstudium Wirtschaftsinformatik, Universität Wien
2004-2009	Bachelorstudium Politikwissenschaft, Universität Wien
2007-2010	Masterstudium Wirtschaftsinformatik, Universität Wien
2008/2009	ERASMUS-Semester, Dublin City University, Dublin, Irland

Publikationen

G. Stürmer, J. Mangler, and E. Schikuta, "A domain specific language and workflow execution engine to enable dynamic workflows," in Proceedings of the 1st International Workshop on Workflow Management in Service and Cloud Computing, IEEE Computer Society Press, 2009.

G. Stürmer, J. Mangler, and E. Schikuta, "Network Based Execution of Dynamic Workflows in Grid and Cloud Based Environments" 3rd Austrian Grid Symposium, September 28-29, 2009, Linz, Austria.

G. Stürmer, J. Mangler, and E. Schikuta, "Building a Modular Service Oriented Workflow Engine", IEEE International Conference on Service-Oriented Computing and Applications (SOCA'09) December 14-15, 2009, Taipei, Taiwan.