Doctoral Thesis

A generic kernel for reliable process support

Author(s):
Hagen, Claus

Publication Date:
1999

Permanent Link:
https://doi.org/10.3929/ethz-a-002056095

Rights / License:
In Copyright - Non-Commercial Use Permitted
A Generic Kernel for Reliable Process Support

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of
Doctor of Technical Sciences

presented by
Claus Johannes Hagen
Diplom-Informatiker Universität Erlangen-Nürnberg
born November 25, 1966
citizen of Germany

accepted on the recommendation of
Prof. Dr. G. Alonso, examiner
Prof. Dr. T. Gross, co-examiner
Dr. C. Mohan, co-examiner

1999
Seite Leer /
Blank leaf
Acknowledgements

First of all, I have to thank Prof. Gustavo Alonso, with whom I have worked for the last four years and who always was an excellent supervisor, providing the support and guidance that made this work possible. I am also thankful to Prof. T. Gross and Dr. C. Mohan for accepting to be co-examiners. Thanks are owed also to Prof. H.-J. Schek, who brought me to ETH and was my first supervisor until Prof. Alonso formed his own research group.

I would like to thank all colleagues from the Information and Communication Systems Group and from the Database Research Group for their friendliness, discussions, and help. Special thanks go to Andrew Deacon, for introducing me into the secrets of transaction processing and workflow management. To Markus Kradolfer from the University of Zurich for many fruitful discussions. To Heiko Schultd for sharing the office, exchanging ideas, and good collaboration. To Amaia Lazcano and Christoph Schuler for good cooperation in common projects. To Antoinette Förster for keeping the whole business running. To Marco Schmidt for lots of technical assistance.

This work would not be possible without the contribution of many interested, talented, and engaged students, who contributed in form of semester or diploma works. Their work was invaluable for the OPERA prototype. So, thanks to Peter Arnold, Dimitar Boskov, Jakob Bosshard, Carlos Canamero, Keyuan Cao, Cornel Führer, Roberto Italia, Stefan Middendorf, Florian Nussberger, Michael Schoch, Christoph Schuler, Dominik Seitz, Oliver Separovic, Marcel Vinzens, and Beat Zimmermann.

Thousand thanks to Simone for her love and support, for keeping me motivated, and for enduring four years of train-riding, high telephone bills, and, too often especially towards the end, bad mood.

Finally, I dedicate this thesis to my parents. Their neverending love, support and encouragement have been irreplaceable gifts.
Seite Leer / Blank leaf
# Contents

Abstract 1

Kurzfassung 3

1. Introduction 5
   1.1. Motivation 5
   1.2. Problem description 6
   1.3. Document structure 9

2. Process Support in Distributed Systems 11
   2.1. Introduction 11
   2.2. A motivating example 12
   2.3. The idea of a Process Support Kernel 14
   2.4. The concept of OPERA 16
   2.5. The functionality of a Process Support Kernel 17

3. Basic Concepts and Related Work 23
   3.1. Process-based middleware 23
   3.2. Transactional middleware 27
   3.3. Distributed computing platforms 29
   3.4. Coordination languages 31

1. Process Model 33

4. Basic Process Model 35
   4.1. Requirements 35
   4.2. OCR: the OPERA Canonical Representation 38
   4.3. Example 44
   4.4. Mapping external languages to OCR 47
   4.5. Discussion 49

5. Atomic spheres 51
   5.1. Introduction and motivation 51
   5.2. Overview: The notion of atomicity 54
   5.3. Definition: spheres of atomicity 58
   5.4. Discussion 62
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.5. Load balancing policies</td>
<td>145</td>
</tr>
<tr>
<td>11.6. Discussion</td>
<td>146</td>
</tr>
<tr>
<td>12. Conclusions</td>
<td>149</td>
</tr>
<tr>
<td>12.1. Summary</td>
<td>149</td>
</tr>
<tr>
<td>12.2. Outlook</td>
<td>151</td>
</tr>
<tr>
<td>Bibliography</td>
<td>153</td>
</tr>
</tbody>
</table>
Abstract

The full exploitation of today's networked computing platforms is complicated because of the lack of suitable programming support. This is especially the case if distributed applications have to be built out of existing, stand-alone programs which were not defined to interoperate with others. What is needed to support these kinds of applications is the necessary "glue" to integrate them into a coherent whole. The motivation behind this dissertation is the belief that, for a wide range of applications, the necessary support can be provided in the form of process support systems (PSS), metaprogramming environments allowing to describe a distributed computation as a collection of program invocations and enforcing its correct execution. The background of the dissertation has been the development of a process support system kernel providing a generic set of services suitable for many process-based applications. This kernel can be (and has to be) extended with domain-specific enhancements to adapt it to specific application environments.

Of the large number of open problems existing in such a system, the thesis concentrates on a number of key topics. The focus is, besides the development of a basic process modeling language and a basic architecture (which has been implemented in form of the OPERA system), on quality aspects of distributed computing. The range of possible quality improvements in a PSS context can be categorized into increasing the quality of the enactment service (by providing fault tolerance), increasing the quality of the process model (by adding new constructs which simplify process modeling), and increasing the quality of the architecture (by providing robustness and scalability). The need to provide an enhanced "quality of service" leads to a number of problems originating in the high degree of distribution, heterogeneity, and node autonomy to be found in the execution environment. Solving them requires the application of known concepts from related areas of research and their adaption to the special needs of process support systems.

Fault tolerance, i.e., the ability of a system to continue operating in spite of failures, has two general aspects: Writing fault-tolerant applications and the fault tolerance of the system itself. To solve the problem of application fault tolerance, the thesis aims at combining database concepts (transactions) and programming language mechanisms (exception handling) into a flexible solution for failure handling. Also addressed is the integration with inter-task communication mechanisms, which lead to interesting recoverability problems. Providing system fault tolerance requires to mask failures such as server crashes or communication errors. A fault-tolerant system has to introduce redundancy in order to guarantee proper operation even in the case of component failures. To this end, the thesis introduces a cluster architecture, on top of which backup mechanisms can be implemented. From a technical point of view, there are several possibilities to implement such backup services. The main contribution of the thesis in this regard is a performance study evaluating the advantages and drawbacks of the various variants. The distributed architecture is also the key to system scalability, which is addressed in the thesis by the description of a process migration facility allowing to move processes transparently between servers.

Together, the solutions presented in this dissertation enable the construction of adaptable process support systems which allow the reliable execution of distributed applications. This has also been demonstrated by the OPERA prototype, which has shown its usability in several projects.


Kurzfassung

Prozessmigrationsalgorithmus vor, der auf Basis der Replikationsmechanismen arbeitet und die Voraussetzung für Lastbalancierungsmechanismen ist.

Insgesamt erlauben die Ergebnisse dieser Dissertation die Konstruktion adaptierbarer Prozessunterstützungssysteme, die die verlässliche Ausführung verteilter Applikationen ermöglichen. Dies hat sich anhand des OPERA-Prototypen gezeigt, der bereits in verschiedenen Projekten eingesetzt wird.
1. Introduction

1.1. Motivation

Networks of workstations have become the predominant computing platform of our days. Due to ever-increasing technological advances and ever-falling hardware costs, computer networks have become commodity items, and, as a result, aggregations of computers connected by a LAN are now the most pervasive computing infrastructure. The performance of standard PCs has reached a degree that allows to build supercomputers out of commodity, off-the-shelf (COTS) components [BSS+95]. At the same time, the proliferation of the Internet and its protocols has delivered both a de-facto communication standard (TCP/IP) and a world-wide communication environment. These two developments allow the deployment of applications whose components can be arbitrarily spatially distributed.

From a research point of view, this means we now have a chance to implement truly distributed systems, making use of the large body of (mostly theoretical) research that has been conducted in this field over the past twenty years. In practice, however, building distributed applications on top of computer networks is still a complex task due to the lack of adequate distributed programming tools. Existing tools for distributed system construction (middleware) are only partial solutions. Systems such as CORBA [Obj92] or DCE [Ope95], which aim at providing programming platforms that simplify distributed application development, are limited in scope. On the one hand, they lack many of the services which are needed by distributed applications, like transaction support and automatic scheduling services. In CORBA, many such services have been defined, but most of them have never been implemented. On the other hand, they require dedicated invocation interfaces, which are costly to provide, especially for so-called legacy applications, i.e., applications for centralized, non-cooperative architectures. These kinds of applications are, however, the vast majority of software existing today. Because of the missing software support, it is not possible to integrate those autonomous programs into a coherent whole, which leads to a situation in which the immense potential of the network is used only for file and printer sharing. Many distributed environments would benefit from support software providing the necessary glue to integrate heterogeneous, already existing programs into coherent units. Such a unit (a set of disparate programs together with control and data flow dependencies among them) can be seen as a (distributed) process [AHST97]. The suitable "software glue" could take the form of a process support system (PSS), and the main goal of this thesis is to discuss how such a PSS can be constructed.

Because of the lack of dedicated process support software, developers have tried to use Workflow Management Systems (WFMS) [GHS95, CHRW97, DKÖS98] for this task [MVW96]. WFMS were originally designed to automate business processes. To this end, they provide a modeling language allowing to describe control and data flow dependencies between business activities that cooperate in order to achieve a certain goal, like loan approval or insurance claim processing. Each activity can either be the execution of a program or a human task, and the modeling environment is accompanied by a runtime system which controls the execution of business processes based on the formal specifications, i.e., notifies users and invokes programs in the right order. This includes the
1. Introduction

ability to start programs on disparate nodes, due to the fact that workflow systems were designed for the LAN based environments found in modern offices. Because of this capability to control the execution of distributed applications based on a “super-program”, WFMS seem to be eligible as general process support tools. It turns out, however, that existing WFMS do not provide the flexibility needed to support many kinds of distributed applications [MVW96]. The reason for this is that WFMS are too specialized. Because of their focus on business processes, they lack important functionality needed in many applications areas, like flexible modeling languages or data management functionality. At the same time, they are not extensible because of missing interfaces which would allow to customize their functionality.

It has been pointed out that a full-fledged support system for distributed applications has to provide much more than what is present in workflow-management systems. Grimshaw et al. [GFLH98] envision a metasystem as a system which “gives users the illusion that the files, databases, computers, and external devices they can reach over a network constitute one giant transparent computational environment”. On top of such a metasystem, meta-applications will be running, which are “multicomponent applications whose components may have been previously executed as stand-alone computations”. Combining the ideas of metasystems and process support leads to OPERA, a project aiming at the development of a process support kernel (PSK) which can be seen as the nucleus of a metasystem [AHST97]. The rationale behind the idea of such a kernel is to provide the basic services needed to facilitate the construction of distributed applications in a heterogeneous environment. Hence, the kernel will serve as unique middleware that integrates a collection of disparate machines, programs, and data. The services such a kernel has to provide are manifold. They range from basic communication, data transfer and data conversion support over adequate programming tools to advanced aspects such as quality of service guarantees, monitoring facilities, distributed scheduling services, and mechanisms for bookkeeping, auditing, and analysis. In fact, a process support kernel can be seen as an operating system at high granularity, managing programs instead of threads, files instead of disk blocks, and computers instead of processors.

There are many areas which can benefit from such systems. Possible applications range from scientific simulation systems [GFLH98, AH97] to complex inter-enterprise applications supporting virtual enterprises [APH+99]. Even workflow management systems could benefit from the existence of a process support kernel. Some of the features that are provided by the kernel, like the ability to control distributed transactions, are very useful in workflow contexts which have to access distributed information systems.

1.2. Problem description

This thesis can be seen as the first step towards a generic process support kernel. It should be obvious that tackling all the problems that have to be solved to come up with a complete solution is beyond the scope of a single dissertation. Because of this, the thesis addresses only a number of key topics. The focus of the thesis is, besides the development of a basic process modeling language and a basic architecture (which has also been implemented), on quality aspects of distributed computing. The range of possible quality improvements in a PSS context can be categorized into increasing the quality of the enactment service (by providing fault tolerance), increasing the quality of the process model (by adding new constructs which simplify process modeling), and increasing the quality of the architecture (by guaranteeing scalability). The need to provide an enhanced “quality of service” leads to a number of problems which have their origin in the high degree of distribution, heterogeneity, and node autonomy which is found in the execution environ-
1.2. Problem description

Improving Quality of Service
Atomicity          Exception Handling  Availability  Scalability

New services
Event Handling      Interprocess Communication

Adapting the kernel
Scientific Modeling  Electronic Commerce

Basic Architecture and Design  Implementation of OPERA

Concept of PSS and PS kernel

Figure 1.1.: Overview: Main topics of the thesis and the respective chapters

ment. Solving them requires the application of known concepts from related areas of research and their adaption to the special needs of process support systems. The various topics of interest will be outlined shortly in this section and are discussed in detail in later chapters.

Fig. 1.1 tries to visualize the contributions of the thesis and the relations among them. The numbers indicate the chapters discussing the various topics.

- The idea of process-based distributed computing and the concept of a process support system as “software glue” for constructing distributed applications serves as a foundation for all other work of the thesis. Based on this initial concept, a core modeling language and a generic system architecture have been developed and implemented. The main challenge in designing the basic kernel has been to provide the necessary flexibility which allows to extend the kernel in various directions. This required the design to be modular, to provide appropriate interfaces which facilitate adding new functionality and allow to adapt the system for a specific application. The basic extensible design made it possible to study the various quality improvements mentioned above.

- Improving the quality of service of a PSS requires primarily to enhance its fault tolerance, i.e., the ability of the system to provide services even if failures occur. Because of the potential complexity of processes and the large number of participating systems, it is difficult to solve. The problem has two general aspects: (1) Writing fault-tolerant applications and (2) the fault tolerance of the system itself. In regard to applications, the modeling language has to be extended with constructs for error detection and error handling, and the runtime system has to be modified in order to implement the new semantics. Concepts for application fault tolerance have been developed in many fields. Two techniques are especially relevant for process support environments: transactions and exception handling. Fault-tolerance of the PSS, i.e., its ability to survive partial or complete failures of the system components, can be fostered through persistence. Storing the states of all processes on stable storage avoids the loss of information if servers fail. This provides a basic degree of fault tolerance, which can be enhanced if continuous availability is to be guaranteed. If, however, stronger guarantees like continuous availability are required, advanced techniques such as replication and backup mechanisms are needed.

Regarding fault tolerance, the thesis investigates the following topics:

Failure atomicity: The concept of atomicity as it is used in databases provides a well-known abstraction for failure handling. It is based on backward recovery: In the case of a failure,
an application or parts of it are "rolled back" to a previous consistent state. From this state, the computation can continue by re-trying the previously failed instructions or by following alternative execution paths. Providing atomicity in a PS environment is complicated by the fact that the components of a process are autonomous and do not necessarily support the rollback of activities. Hence, the challenge is to adapt the notion of atomicity to the new environment. This will lead to the support of transactional distributed computing, which is inevitable if processes access multiple data repositories and have to retain global consistency. Database researchers have developed a plethora of advanced transaction models [Elm92] which tried to solve similar problems. They remain, however, mostly theoretical concepts which have to be adapted and extended to be useful for process support systems. An atomicity concept for distributed processes is developed in chapter 5.

Exception handling: Paradoxically, adding fault tolerance mechanisms to a system can reduce its fault tolerance because of an increased system complexity [AL81]. Because of this, programming language designers have very early started to develop language elements that allow to separate the failure handling aspects from the "normal" flow of control in a program [Par72, Goo75]. Such elements are of great benefit in a process modeling language. Properly used, they can provide for forward recovery and thus complement the backward recovery provided by the atomicity concept. A problem is, however, how to combine both concepts and adapt them to the context of distributed processes. These issues are discussed in detail in chapters 6 and 7.

Availability: Fault tolerance of the system (in contrast to fault tolerance of the application) requires to mask failures such as server crashes or communication errors. If processes are time-critical, outages of the process support system and the resulting blocking of processes are not acceptable. A fault-tolerant system has thus to introduce redundancy in order to guarantee proper operation even in the case of component failures. By replicating process state information, it is possible to guarantee liveness of processes even if single servers crash. From a technical point of view, there are several possibilities to implement such a replication mechanism. Chapter 10 evaluates these variants and provides a performance study.

• Improving the quality of the modeling language means enhancing its ability to reflect user requirements in an adequate way. One restriction in most existing process modeling languages is, for example, the missing support for inter-process communication. Most PSS adhere to a black box principle, i.e., processes, blocks, and activities do only return information when they have finished execution. No exchange of intermediate data is possible. This is a major drawback for many applications which require the cooperation of running tasks in order to reflect application semantics properly. In chapter 8 a mechanism for the controlled exchange of information between running tasks is proposed. It is based on an event mechanism that implements a publish-and-subscribe policy to control the channels for data exchange. The main challenge in designing such a mechanism is to guarantee its proper integration with atomicity facilities. Atomicity can require the partial rollback of tasks if failures occur. This implies that information which was already sent to other tasks is now invalid. A number of algorithms for the handling of this problem are discussed in chapter 8.

• Improving the architectural quality of the system requires the guarantee of scalability. With increasing size of distributed systems, scalability of the PSK becomes a matter, and appropriate mechanisms have to be devised which avoid that the administrative services of the
PSK become a bottleneck. A solution is to implement the PSK itself as a distributed application, which can in principle guarantee unlimited scalability. In chapter 9, we develop a cluster architecture for a PSK and discuss the problem of load balancing in such a system.

Many of the quality aspects discussed in this thesis were not investigated in detail by previous work. This is especially true for the strategy of combining well-approved language features and advanced transaction mechanisms in order to develop a flexible and powerful failure handling mechanism for process support systems. Similarly, previous approaches of adding fault tolerance to workflow systems have concentrated too much on database ideas and did not respect the special conditions of process support contexts. The extension of exception mechanisms towards an inter-process communication facility (chapter 8) has never been proposed in this form and is an important feature for many PSS applications. Finally, replication concepts for process availability were discussed only theoretically before. The thesis provides a discussion of implementation strategies and an evaluation based on performance measurements. Moreover, this thesis shows how these ideas can be extended to support process migration facilities, which allow the flexible administration of large PSS clusters and are a basic prerequisite for scalability.

The OPERA kernel has already proven its usability in a number of follow-up projects. Some important extensions are GEO OPERA [AH97] and WISE [AFH+99]. The former specializes the kernel towards a system supporting scientific modeling. The latter is a system supporting inter-enterprise electronic commerce.

1.3. Document structure

The rest of the thesis is structured as follows. Chapter 3 gives a detailed introduction into the topic of process support and the concept of a process support kernel. Chapter 3 gives an overview of related work and defines a number of basic notions and concepts which will be used later. The rest of the thesis is divided into two parts.

*Part one* discusses the aspects related to the modeling language: Chapter 4 describes the basic modeling language. Chapter 5 discusses the problem of fault tolerance and presents a concept for atomicity in process support systems. Chapter 6 extends atomic spheres with exception handling mechanisms, and chapter 7 presents a validation algorithm for fault-tolerant processes. Chapter 8 describes a concept for event-based inter-process communication.

*Part two* is concerned with aspects pertaining to the system architecture: Chapter 9 describes the basic architecture of the OPERA kernel. Chapter 10 presents a distributed architecture and evaluates the concept of guaranteeing continuous availability through replication methods. Chapter 11 discusses how the distributed architecture can be utilized to provide scalability of the system. Chapter 12 concludes the work.
1. Introduction

2.1. Introduction

The last few years have seen a proliferation of distributed computer systems that has changed many computing paradigms and is still changing them at a rapid pace. Until recently, for instance, information systems were mostly implemented around centralized databases running on mainframes. Today, they are migrating rapidly towards distributed, cooperative architectures. Papazoglou and Schlageter [PS98] characterize the next generation of information systems as reconfigurable aggregations of disparate problem-solving components and their underlying information sources. This definition is not restricted to information systems. It nicely captures the most salient properties of many contemporary distributed applications: Applications execute as interactions between spatially distributed, cooperating components rather than as invocations of stand-alone programs.

The computing environment is heterogeneous by nature, with computers being manifold in terms of architectures and operating systems and applications being manifold in terms of programming languages and interfaces. The single components provide a variety of services, ranging from data storage over specialized data processing to presentation services. The distributed application structure is flexible, can be adapted to actual requirements, and is likely to change frequently. This can have important advantages like increased availability and high performance through parallel processing, but it also requires that applications can deal with varying system configurations.

Because of the heterogeneous and changing nature of the new environments, building distributed applications on top of them is complicated. This is especially the case if the components were not designed to cooperate in the first place. Deploying middleware such as transaction processing monitors [GR93, BN97] or distributed object management (DOM) environments [OHE96] does not solve the problem if many of the existing components are stand-alone programs which lack the interfaces needed for these products. The only solution is to write wrapper programs which mediate between the program interface and the interface required by the middleware. Developing wrappers, however, is costly. Moreover, none of the existing middleware products provides the full range of functionality needed for distributed applications [AHST97]. CORBA environments, for example, lack support for distributed transactions which is needed if an application changes multiple information repositories. This functionality is provided by TP monitors which, however, lack many of the CORBA features. Combining both types of systems, like this is advertised by system vendors [ION97], leads to a large system footprint, is expensive, and still lacks important functionality such as the ability to control the state and progress of running processes.

A Process Support Kernel as it is advocated in this thesis can solve this problem by concentrating on process-based applications. Integrating the various middleware concepts into a PSK provides a platform for reliable distributed computing that can handle heterogeneity, guarantee transactional properties, and provide a rich set of additional services. To some degree, process support systems can be seen as very sophisticated successors of job control systems, with the im-
important difference that they work over a heterogeneous set of computers and provide a much richer set of functionality. This idea will be further explained in the following sections.

The aim of this chapter is to provide a motivation and describe the fundamental ideas behind Process Support Systems. To this end, section 2.2 presents a motivating example. Section 2.3 introduces the concept of a process support kernel, and section 2.4 gives an overview of OPERA, the process support kernel developed as part of this dissertation. Section 2.5 illustrates the concept by outlining the functionality to be expected from a process support system.

2.2. A motivating example

Research in applied geographic disciplines such as environmental sciences is, to a large degree, concerned with the modeling of complex phenomena in order to be able to predict future developments. The developed models, usually series of mathematical transformations, are applied to spatial data which have been collected through various methods, such as measurements, field exploration, or satellites. Fig. 2.1 shows a set of transformations together with their respective data sets as they are used for predicting information like soil erosion patterns and vegetation change in a specific geographic region [AE94]. Some of the data sets are supplied externally, like the rainfall records which are delivered by measurement stations positioned throughout the area of interest. Most of the data sets are derived through transformations, like the spatial rainfall data which are generated using an interpolation model based on the rainfall records that provide information only for the geographic positions of the points of measurement. Some data sets, like the hydrograph describing the change in the level of waters over time, are generated by complex series of transformations applied to a large number of initial and intermediate objects. The transformations themselves are either stand-alone programs or executed by specialized applications like mathematics packages or simulation systems. Due to the experimental character of the application, data as well as algorithms change frequently, leading to the need for recomputations of dependent data sets as well as the necessity of bookkeeping about the various file and algorithm versions that were produced.

![Figure 2.1: Example of a set of geographic models computing the hydrographie characteristics of a region [AH97]](image)

From a technical perspective, what the graph shows is the description of a distributed process, prescribing the order of execution of a set of applications which are distributed over a computer
2.2. A motivating example

network. This geo process [AH97] is characterized by a high degree of distribution and a high
degree of heterogeneity. The data is distributed over different systems using various media (for
instance, databases in the case of rainfall records, GIS for the topographic maps, plain operating
system files for the vegetation samples, and tertiary storage for the satellite images). Likewise, the
process contains programs with special demands in terms of hard- or software and resources. This
is not surprising given that today’s distributed environments consist of a mix of architectures and
operating systems and that most software is not portable, leading to the fact that most programs
will run only on specific machines. Furthermore, there are programs which need specialized re-
sources, like mathematic libraries or output devices. Finally, computing the various data sets in a
distributed heterogeneous environment requires not only to invoke the right programs on the right
machines in the right order, but also to transport data between nodes and convert it between the
formats required by the various programs.

Today, the execution of these types of scientific applications is controlled mostly manually.
Tasks like ensuring the correct order of execution of the various models and keeping track of the
various generated files and modified programs have to be done by the scientists in addition to their
research work. Ioannidis et al. [ILGP96] note that “a small laboratory that can easily generate
and store several megabytes of data per day is still dependent on the good old paper notebook
when it comes to keeping track of the data.” The sheer complexity of the models as well as the
large amount of work necessary for maintaining all files and programs, however, is not only time-
consuming, but also error-prone. An automatic execution is thus highly desirable. Automating
scientific applications, however, goes far beyond simply controlling the order of model execution
(a task for which simple scripting language or even Makefiles would be sufficient). Here is an
overview of all the services which are needed:

Modeling support: The systematic analysis of the models and data objects and their interdepen-
dencies is a prerequisite for its successful automation. An appropriate modeling tool has to
provide a language in which to describe all relevant aspects of a given set of transforma-
tions. The language could be graphical, leading to specifications similar to the picture in
Fig. 2.1. It has to reflect data aspects (which data objects are externally provided, which
are generated through transformations, how can the objects be accessed) as well as process-
ing aspects (which models are used to generate which derived data sets, and how are the
respective programs invoked). In addition to this fundamental modeling support, it is also
important to provide facilities supporting maintenance and re-use of specifications. A spec-
ification repository, for example, would allow to store the specified models and to retrieve
them later based on a variety of parameters, either for modification or to use them as tem-
plates for the modeling of new applications. Such a repository is indispensable especially if
large sets of process models have to be maintained and if many researchers are participating
in the modeling.

Automated execution (Enactment): The specifications developed with the modeling environ-
ment can be used to enact the processes. An enactment service has to take care of all aspects
related to program execution. This includes the automatic detection of changes in the base
data and the subsequent recomputation of dependent objects. It also implies data transfer
and data conversion wherever this is necessary. The experimental nature of the application
requires a high degree of flexibility from the enactment service: Often, researchers need to
apply a series of transformations to multiple versions of the same data object in order to
compare the changes in the derived data. Likewise, the testing of several versions of the
same transformation (if each version uses a different algorithm) with the same set of data
2. Process Support in Distributed Systems

has to be supported. To this end, the enactment service has to keep track of changes in data and algorithms by applying a versioning scheme.

**Automated Bookkeeping:** Recording the history of experiments is a key requirement in all systems that support scientific applications [MVW95, ILGP96]. In the spatial application above, a derived data object (like the hydrograph) can be interpreted correctly only if all aspects of its generation (the object's lineage) are known. This includes the source objects used during its computation as well as detailed information about the algorithms applied in order to generate the object. For instance, a correct evaluation of the hydrograph requires not only knowledge about the time period and geographical scope of the rainfall records used as the basis for the prediction, it is also necessary to know which particular algorithms were used in the storm and discharge models. This means that the enactment service has to store the complete history of a data object’s generation in order to allow its later interpretation. In addition, the system should provide support for recording the development history of data sets and algorithms. An automatic assignment of version numbers to objects that change supports effective bookkeeping and is a prerequisite to functionality which allows to compare different versions of the same object.

**Query functionality:** The metadata stored by the runtime system (version information, lineage data) are an important information resource and have to be made accessible to the researchers through a convenient query interface. Scientists need to know not only the information pertaining to a specific data object (which algorithms have been used to generate object X), but may be interested in predictions concerning the impact of changes to algorithms and data. Questions like “Which data objects are recomputed if I change algorithm X” or “What are the effects of storing a new set of satellite images” fall into this category. Another important category of queries are statistical inquiries, like “How often was data object X changed in the last half year” or “What was the average duration of all executions of algorithm Y”. This information can be used for process optimization, because it enables the detection of bottlenecks. To enable this wide range of potential queries, full-fledged database query languages such as SQL or OQL, potentially with specific data analysis extensions, have to be used.

**Parallel execution and load balancing:** To speed up execution, the enactment service should execute programs in parallel whenever this is possible. Taking into account the dependencies implied by the process semantics, the resource requirements of the particular programs, and the actual load situation in the distributed system, the enactment service will act as a scheduler, trying to minimize execution times. This requires mechanisms for load detection as well as the deployment of appropriate load balancing policies.

This list of requirements gives an impression of the variety of services needed in this particular example. For other application areas, the list will differ, with some functionality being unnecessary and other features being important. In the next section, we will develop the idea of a generic, extensible system for the support of arbitrary distributed applications.

2.3. The idea of a Process Support Kernel

The above example illustrates the demand for suitable programming tools in distributed systems. The closest category of existing software concepts are Process Support Systems like Workflow
2.3. The idea of a Process Support Kernel

Management Systems (WFMS) [GHS95] or Process Centered Software Engineering Environments (PCSEE) [FKN94]. These systems are centered around the notion of a process as the coordinated execution of activities in a distributed environment and provide process-based programming and execution services for a specific application area. Workflow Management Systems automate business processes with a strong focus on the coordination of human activities and the access to information systems. PCSEE automate the various phases of software development in different degrees of comprehensibility, with some systems supporting all stages from initial requirements engineering over development and testing to customer support and software maintenance. Both WFMS and PCSEE combine a modeling environment which allows to specify all relevant aspects of a process with a runtime engine controlling its execution. (Scope and functionality of these systems are described in detail in chapter 3). A problem of existing process support tools is that they are tailored towards a specific domain. Each system provides some functionality only needed in its particular area of application and lacks important services which are needed in other areas. Take as an example workflow management systems. Because of their aim of supporting business processes involving human activities, they have a staff modeling component which allows to specify the enterprise structure and the rights and responsibilities of each employee. This information is then used by the runtime system to assign work to workers using abstract roles describing capabilities of employees and requirements of activities [Bus94]. Such a facility is indispensable for business processes, but it is (at least in this form) more or less useless for scientific environments. On the other hand, services like lineage tracking and program scheduling are missing in workflow systems, a fact which prohibits using them for scientific applications like the one we have described above. Current process support systems are thus not suitable as general platforms for programming distributed systems. Extending existing systems is not a feasible solution either. We have already noticed that because of missing interfaces for extensibility, approaches aiming at extending current workflow management systems have failed [MVW96, Arn96].

The limitations found in existing process support systems call for a more generic approach towards the development of programming environments for distributed systems. The motivation is simple: While each of the various application areas for process-centered distributed computing (like scientific environments, business processes, or software engineering) has specific requirements not shared with others, there clearly is a common set of generic services needed by all of them. It should thus be possible to develop a generic process support kernel (PSK) which provides the basic services and can be tailored towards specific application domains by adding the necessary domain-specific functionality. The advantage of this approach is a considerable reduction of development effort if new process support systems have to be built. At the same time, it is possible to implement the base services in a quality that would be too costly for most ad-hoc implementations. This aspect of the PSK concept is best illustrated by a comparison with database management systems (DBMS). The great success of DBMS is based on the fact that they concentrate on data storage and retrieval services and provide a set of interfaces for application development. This has two benefits. First, applications do not have to implement their own data storage functionality, but can rely on interfaces like SQL [MS92] or OQL [CBB+97] to store and retrieve information. This simplifies application design considerably. Second, the concentration on data storage aspects has allowed DBMS developers to improve the quality of data storage by optimizing storage and retrieval algorithms, exploiting parallelism, and providing a high level of fault tolerance. At the same time, it was possible to offer new services based on the stored data. Examples for these extensions are active database systems which allow the rule-based monitoring of data changes [WC96] or data warehouse concepts which provide analysis environments providing decision support based on the integration of multiple database systems [CD97a]. In addition to the standardized access through interfaces, modern DBMS are highly adaptable by a variety
of means, like analysis and tuning mechanisms which allow to optimize application performance or stored procedure concepts which allow to modify the behavior of the DBMS through built-in programming languages. In the same way that all these benefits were only possible because of the separation of data storage from application design, the separation of distributed programming aspects in the PSK will enable the optimization of process handling and the development of new services.

To support all aspects of distributed process based computing, the functionality of the PSK has to go beyond that of traditional process support systems. As a generic programming tool for distributed environments, it will also have to incorporate concepts which are today provided by other middleware products. Distributed transactions, for example, are the domain of transactional middleware, mostly transaction processing monitors [BN97]. A TP monitor provides a large set of services supporting distributed transactional processing in a client-server fashion. To a large degree, this functionality is similar to what is provided by the PS kernel (programming environment, runtime system, etc). Because of this, coupling a PSS and a TP monitor (this has been proposed for workflow management systems [SJHB96]) introduces a large amount of redundancy which can be avoided if the distributed transaction functionality is integrated directly into the PSS kernel. It is important to note that this does not mean that the services of a TP monitor should not be used if they are available, especially since TP monitors, which were originally developed to support distributed online transaction processing (OLTP), are much more efficient than process support systems which have a larger administrative overhead. It should, however, be possible to execute distributed transactions without the existence of a TP monitor. The same is true for the functionality provided by distributed object middleware like CORBA. The main services of CORBA, providing a coherent namespace and communication mechanisms for a heterogeneous distributed environment, can be easily integrated into the PSS kernel. It is important to note that in both cases, the services integrated into the PS kernel are tailored towards process based computing, which makes the implementation much easier that in the general case, which is handled by the TP monitor or CORBA environments.

2.4. The concept of OPERA

A large part of the work for this dissertation has been the development and actual implementation of a process support kernel, the OPERA system. (The name has two meanings. It can be read as an acronym for Open Process Engine for Reliable Activities or as the plural of the Latin word opus, meaning "labor" or "work".) OPERA aims at generalizing concepts as they are known from workflow management systems or process centered software engineering environments, thereby implementing a wide range of quality enhancements and added functionality based on the ideas depicted in the previous section. The basic architectural principles have to a large degree been inspired by workflow management systems. The principle of the OPERA architecture is shown in Fig. 2.2.

At this place, we will give only a short description of the basic architectural principles. A complete discussion of the OPERA architecture will be provided in chapter 9. OPERA is a server based system. This means that all information for a particular process instance is kept centrally on one server (the only exception is the replication of processes, which is discussed in chapter 10). The main functionality of the server is the interpretation of process descriptions which have been specified in a special modeling language, OCR (Opera Canonical Representation) and are loaded into the server prior to execution. Note that, although the picture shows only one server for simplicity, an actual OPERA system does not have to be centralized. The construction of a server cluster has
2.5. The functionality of a Process Support Kernel

The description in the last section focussed on the architecture and contained only the basic services provided by the PSK. We have already claimed that the concept of dedicated process support...
services provides the chance to implement a wide range of process support services. This section gives a comprehensive overview of the services which should be provided and discusses their most important aspects. An overview of PSS functionality is given in Fig. 2.3. It categorizes the process support services into three layers. At the bottom are the basic support services which form the minimum functionality required. Taken isolated, they do not justify the construction of dedicated process support systems. A justification for process support is only given if the next layer is taken into consideration. It contains quality improvements and additional services which can only be provided using a dedicated PSS. At the top of the triangle are the application-specific services which are not part of the kernel, but have to be added in order to tailor the system towards a specific application area. In the rest of the section, we will describe the different groups of services in detail.

![Functional overview of a Process Support System](image)

**Figure 2.3.: Functional overview of a Process Support System**

### 2.5.1. Basic process support services

The basic services form the core of any process support system.

**Modeling.** The modeling environment has to provide a number of services, some of which are application-specific and can thus not be provided by the kernel. An example is the process modeling language. To enable maximum productivity, it should be as close to the application area as possible. This means that every application needs a different modeling language which is tailored to its specific needs. Business processes, for example, may need a flow-oriented language which is close to the formalisms used in business process re-engineering. For scientific applications like the one described at the beginning of this chapter, what is more appropriate is a language allowing to directly specify dependencies between data objects and algorithms. The only way to enable the usage of various application-specific languages at the user level is the deployment of an “assembler language” at the system level. Such a language should be flexible enough so that all potential applications can be mapped to it. The language can be kept relatively simple because it does not have to provide primitives for data processing. Only language elements related to control flow, data flow, and resource handling have to be integrated.
2.5. The functionality of a Process Support Kernel

**Enactment.** It was already described in the previous section how the PS server uses the process specifications to invoke external applications in the right order and monitor their execution. To enable the execution of remote applications, suitable runtime components have to be provided. There are in general two ways to invoke and monitor an external application. If the external application provides an application programming interface (API), it is possible to integrate calls to it directly into the kernel, which can then communicate with the remote system through appropriate calls. Examples for this functionality are environments such as SAP [Sch98] or CORBA [Sci97]. If no such interface is existing, the control of applications has to be performed through special monitoring programs, so-called program execution clients, which are running on the remote machines and are able to start programs, monitor their execution, and return the results to the engine. This technique has to be used for stand-alone programs, which are usually invoked through an operating system call (usually exec). Using a portable programming language like JAVA [Fla96] avoids the need to write separate PECs for different operating systems.

2.5.2. Advanced process support services

The advanced process support services go beyond plain modeling and execution control. They can be further classified into a number of sub-groups: The class of execution guarantees incorporates quality aspects of process execution, like the guarantee that a process will not be aborted if the process support system should fail temporarily. Resource management services ensure that programs are automatically placed on nodes in the distributed system in a way that optimizes application performance and resource utilization. Monitoring and Analysis features allow to query the state of running processes and perform statistical analysis in order to optimize process structures and resource assignment. Finally, repository functionality means facilities which are important in the modeling phase and which allow the rapid development of applications by re-using previously modeled artifacts.

**Execution guarantees:**

*Persistence:* Storing the state of executing processes persistently provides a basic level of fault tolerance, since it allows to continue with the execution of a process after a failure of the process support server has been repaired. This is inevitable in a process-based environment, especially if processes are long-running and involve a large amount of resources, in which case simply re-starting a process after a failure is not acceptable. Persistent storage adds some amount of overhead to the process management services. This means that there should be the possibility to control whether a process is made persistent or not. Without this option, processes which have tight answer time requirements can not benefit from process support technology.

*Atomicity:* Distributed transactional processing is necessary whenever processes access multiple databases (or other data repositories) and consistency has to be guaranteed. Of the four main properties of the ACID concept (atomicity, consistency, isolation, durability), atomicity is the one with the highest practical significance in distributed environments. (Consistency has to be ensured by the application, and durability is a property that has to be guaranteed by the local repositories. Isolation is in general difficult to achieve, as is described below.) This is also illustrated by the fact that even TP monitors, which are tremendously successful tools for distributed transaction control, do usually only guarantee atomicity. As has been already discussed in the introductory chapter, atomicity is hard to
2. Process Support in Distributed Systems

implement in a process environment because of the autonomy of the systems and programs involved in the execution of a process.

Exception handling: With growing complexity of processes and their environment grows the possibility of runtime failures during process execution. Distributed transactions can provide some degree of fault tolerance, but they have to be accompanied with suitable constructs at the language level if they are to be practically applied. Exception handling concepts, which have been developed originally for programming languages, can provide the missing link.

Synchronization: Whenever multiple processes access the same data objects, data consistency and program correctness can be compromised because of the uncontrolled interleaving of data accesses. This is a known problem in databases, and it has been solved by implementing appropriate isolation mechanisms as part of transactions [BHG87]. In distributed environments, the situation is much more complicated because of the autonomy of the participating databases. The development of appropriate concepts is still an open research topic [AFPS99, SAS99], and it is beyond the scope of this work.

Resource Management:

Scheduling and load balancing: When writing processes for a large distributed environment, the designer cannot anticipate the actual system configuration at the time of process execution. Because of this, it is not advisable to hard-wire the execution site of each program into the process description. Instead, assigning programs to computers should be left to the PS kernel, giving it the role of a scheduler for the distributed heterogeneous system. This includes the incorporation of appropriate load-balancing policies in order to exploit the available system resources in an optimal way.

Data management: In a large heterogeneous environment, it cannot be expected that all computers and programs will use the same data representation. Moreover, spatially distributed computers will not necessarily have access to common data repositories. The PSK has to provide the necessary services for data transport and data conversion. This problem is closely coupled with the problem of optimal scheduling, given that tradeoffs exist between the execution cost on a specific site and the cost for shipping the application data.

Monitoring and Analysis:

Monitoring: For many applications, monitoring the state of running processes is a key requirement. This is due to the fact that processes are often long-running, which may motivate inquiries about the process state by the clients who invoked them. In a centralized system, monitoring the process state is simple since all necessary information is stored as part of the process state in the server's database. Monitoring facilities needs thus only access to this state information. There is, however, the problem that the state is represented in the "internal" representation, which may not be suitable for users who have modeled the process in the application-dependent paradigm. To facilitate application-oriented monitoring, it is necessary to provide a "backward-translation" of the process state from the internal format into the external representation. This translation is part of the application-specific extensions.

Logging: A detailed bookkeeping of the history of all running processes is a requirement in many application areas. In workflow systems, for example, it is necessary for legal reasons to document how certain processes have executed. In scientific applications, there is the need for exact protocols of an experiment in order to make the results reproducible. Logged information can also be a valuable source for process optimization, since it is possible to
analyze the logs in order to detect modeling errors such as bottlenecks or unused resources [AGL98].

**Analysis:** Analysis tools for workflow modeling are an important part of the *metaprocess infrastructure* which supports the procedure of modeling and optimizing processes. Analysis tools can deliver important information based on logged execution data and other sources. Data mining techniques will play an important role in future analysis environments for processes [AGL98].

**Advanced development support:** Supporting the whole life-cycle of a process requires also the provision of suitable repositories which allow to re-use process specifications. Such a repository should have querying facilities which allow to look for process descriptions based on a variety of parameters. The repository functionality has to be provided not only by the kernel, but also as a domain specific extension because the repository should store processes in a form which is comprehensible for end-users. Further important tools for process development are simulation and validation environments which are necessary for the initial testing of processes before they are actually installed. The fact that each process causes immediate effects on the outside world once it is running makes a careful analysis prior to its enactment indispensable.

### 2.5.3. Application-specific extensions

The application-specific extensions tailor the PS kernel for a particular domain. This category subsumes all services and functionality which have to be added to the pure kernel functionality. From the perspective of the kernel, there have to be sufficient interfaces that allow to extend its functionality when this is needed. We will give only two examples of domain-specific extensions here:

**Process representation:** The internal process representation, which can be seen as the “assembler” of the PSK, is not convenient as a modeling language for end-users. Instead, users should be provided with a language which is as close as possible to the problems they want to model. For instance, a modeling language for business processes has to include mechanisms for the specification of rights and responsibilities of employees. A language for scientific applications like in our motivating example should allow to specify the dependencies between models and data sets. This requirement affects not only process modeling, but also process monitoring facilities, since users will want to see in the monitoring tool the same representation which they have specified. This means that adding a new external language to the system requires the provision of two translation mechanisms in order to suffice all requirements.

**Bookkeeping:** It has already been shown in the initial example that scientific applications have advanced requirements concerning the bookkeeping functionality of their PSS. The PS kernel should provide a generic audit interface which allows applications to store process-related information required by users of the system, either for understanding the processes actually executing, or for later interpretation using dedicated analysis tools.
2. Process Support in Distributed Systems
3. Basic Concepts and Related Work

The aim of this chapter is to give a compact overview of related concepts. This description will serve two purposes. First, it aims at describing related areas from which solutions have been integrated into the PSS concept, like transactional middleware or workflow management systems. Second, it tries to point out the main differences between the PSS concept and approaches with similar goals. We will give an overview of these aspects for the following classes of systems and applications:

- **Process-based middleware** (section 3.1), which aims at controlling distributed applications through metaprogram-like approaches. Our notion of process support can be seen as a generalization of the functionality provided in these systems, which encompass, among others, workflow management systems and process-based software engineering environments.

- **Transactional middleware** (section 3.2), like transaction processing monitors, which aim at establishing transactional guarantees for distributed applications.

- **Distributed computing platforms** (section 3.3), like DCE, CORBA, or DCOM, whose main purpose is to provide an abstraction layer on top of existing heterogeneous systems and to offer a variety of services needed for distributed computing.

- **Coordination languages** (section 3.4), like LINDA or ConCoord, which aim at complementing programming languages with features needed in order to effectively control distributed and parallel applications.

## 3.1. Process-based middleware

We have already noted that currently, no universal process-based systems exist. Instead, a number of specialized solutions have been developed, each one tackling a specific area of application. The existing approaches can be further classified in the following way:

- **Workflow Management Systems (WFMS).** Originally conceived for the automation of business processes, the functionality of these systems has a strong focus on office and administrative tasks.

- **Process centered software engineering environments (PCSEE).** These systems aim at automating all aspects of software construction, from requirements engineering to testing. Although their history is older than that of WFMS, and strong similarities in terms of the provided functionality exist, PCSEE have not gained as much popularity. This is mainly due to the fact that they address a very specific application area (the software engineering process), which makes it complicated to adapt them to other problem domains.
3. Basic Concepts and Related Work

- **Embedded Process Tools.** Process control functionality in an embedded form can be found in an increasing number of applications. Examples are standard software packages for business and administration, help desk support systems, or scientific visualization systems. The main difference to the PSS concept is the inflexibility of these systems, which are tightly coupled to a particular environment.

3.1.1. Workflow Management Systems (WFMS)

Workflow Management Systems (WFMS) [GHS95, JB96, DKÖS98], which support the automation of business processes, have started to proliferate around 1990. Unlike their predecessors, the office automation systems of the late 1970's and early 1980's [FK86], they have become very successful. WFMS play an important role especially in conjunction with business process reengineering (BPR) tools, which aim at analyzing, formalizing, and streamlining an enterprise's business processes.

The functionality of a workflow management system is best described by using the characterization of the Workflow Management Coalition (WfMC), a standardization consortium consisting of WFMS vendors and users [WMC94b]. The WfMC defines a workflow as "the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules." A workflow management system is then defined as "a system that defines, creates, and manages the execution of workflows through the use of software, running on one or more workflow engines, which is able to interpret the process definition, interact with workflow participants and, where required, invoke the use of IT tools and applications."

In spite of this basic definition and some standardization efforts by the WfMC, a commonly agreed precise definition of the term workflow is still missing, and, consequently, there is no common notion of the functionality and architecture of a workflow management system. An approach to classify WFMS and related systems can be found in [SB96], where workflow management systems are characterized by a high degree of structure and a high degree of flexibility (Fig. 3.1). This distinguishes them from groupware tools (which support unstructured tasks such as teleconferences), embedded workflow solutions (which usually provide only limited modeling facilities), and process-oriented office systems, which are oriented towards automating a single user's flow of work between different office automation tools.

![Figure 3.1.: Classes of process-support systems according to [SB96]](image-url)
3.1. Process-based middleware

Another classification of different types of workflows was given in [McC92] and is commonly used. Here, workflows are divided into three classes, depending on the nature of the processes they support and the value these processes have for the enterprise:

- **Ad hoc workflows** have no predefined structure and are used when no common pattern of behavior among multiple workflow instances exists. Because of the lacking process specification, workflow support is naturally limited to the provision of communication mechanisms allowing to route case data between workers and (at best) some support for logging and state tracking.

- **Administrative workflows** "involve repetitive, predictable processes with simple task coordination rules, such as routing an expense report or travel request through an authorization process" [GHS95]. Because the structure of such processes is known in advance, they can be automated and the WFMS will take responsibility of ensuring that all steps are executed in the required order.

- **Production workflows** are like administrative workflows in that they are predictable, however, they incorporate access to multiple information systems in order to make most of the decisions during execution. Production workflows pose the highest demands on a workflow management system, because of the multitude of systems involved and the complex process model required.

In comparison with the PSS concept, production workflows are the closest category. A production workflow system allows to model and execute workflow processes and provides a number of interfaces allowing to integrate programs and information sources. Hence, it contains some of the functionality needed in a process support system. However, due to inherent restrictions and a narrow focus on business processes, workflow systems are not suitable as general purpose process support tools. We will discuss this in more detail below.

3.1.2. Process Centered Software Engineering Environments (PCSEE)

Process centered software engineering environments (PCSEE) started proliferating in the 1980’s, when it became obvious that using sophisticated tools for software engineering has limited advantages unless the complete software development process is taken into consideration [Hum89]. This process comprises the complete product lifecycle, starting with the initial requirements engineering, incorporating the complex phases of design, development, and testing, and also capturing maintenance and refinement aspects. The establishment of quality standards for software development has fostered the construction of systems that automate the development process. These systems aim at providing an integration layer on top of existing isolated software engineering tools like editors, compilers, and configuration management systems.

The range of services provided by actual systems varies widely. To give an idea of the diversity, [Hum89] describes the following examples of functionality to be supported by a PCSEE in addition to the plain automation of the process:

- Applicability checks for activities. For instance, the validation that all failures found in unit tests are fixed before integration tests are started.

- Forward or backward chaining in order to maintain implementation consistency. For instance, the identification of all program versions to which a given fix applies.
3. Basic Concepts and Related Work

- Checks for the side effects of potential actions. For instance, the determination of all modules that are affected by a change in data structures.

- Support for building logs and data records of process transactions. For instance, recording of the time spent, defects found, or changes made during a particular process.

- Provision of symptomatic assessments of software process data. For instance, the generation of summaries of the error or change history for all modules.

The list shows that PCSEE add a wide range of specialized services to the functionality which is provided by a workflow system. The application area is, however, even more restricted, which makes it difficult to deploy these kind of systems as general process support systems.

3.1.3. Embedded workflow

Embedded workflow solutions integrate process support functionality into existing products, with limited functionality supporting only the specific area of application. Examples are:

- The workflow module of SAP R/3 [SAP97]. It allows to define workflows involving the various modules inside an SAP system. There is also support for incorporating external applications. They have, however, to be linked against SAP runtime libraries in order to provide the necessary interfaces for interaction.

- Help Desk Systems, such as Remedy [Rem96] incorporate workflow functionality implementing the “business process” of routing a help request or trouble ticket between the different parts of a company’s customer support department.

- Scientific visualization systems such as Khoros [RK94] require the application of a multitude of complex transformations to input data in order to produce comprehensible output. To organize the flow of control between the algorithms, workflow engines are incorporated which can in some cases even control the distributed execution (by allowing to assign hostnames to tasks). There are, however, no means for scheduling or load balancing.

This list could be continued. Due to the growing importance of the notion of process, more and more vendors integrate workflow functionality into their products. This functionality is, however, restricted and can hardly be adapted to other applications.

3.1.4. Comparing existing process-based middleware and the PSS concept

Reviewing the functionality provided by existing process-based systems, we find that they are not suitable as the basis for general process support solutions, mainly due to the following reasons:

- Narrow focus. All systems described concentrate on a specific area of application. WFMS, for instance, are dedicated to business processes to a degree that makes it difficult to use them for other areas of application. This has been reported, for instance, in [MVW95], where an approach has been taken to use a commercial workflow management system for scientific process support. The effort failed because of the inflexibility of the system which made it impossible to adapt it to other than business semantics. There are a number of restrictions which make adapting a WFMS a complicated task. As an example, consider the resource model to be found. Most WFMS provide enhanced facilities for the modeling of
staff hierarchies, capabilities, and responsibilities. This functionality, together with a role-

based concept for work assignment, turns out to be a powerful load balancing mechanism

as long as only human collaborators have to be integrated. If it comes to load balancing be-

tween execution sites for programs, the mechanisms cannot be used and there are no equiv-

alent means to be found. Other important restrictions are the limited modeling language

(often only suitable for simple control structures) and the limited number of interfaces for

application invocation (which has often a strong focus on office tools). Similar restrictions

can be found in software process environments and in embedded workflow tools.

- **Limited extensibility.** Tightly related to the previous problem is the lack of “hooks” that

would allow to enhance existing systems in order to adapt them to new kinds of applications.

Many important aspects like the process modeling language and the interfaces to external

applications are hard-wired into the products.

- **Lack of execution guarantees.** The existing systems lack execution guarantees such as atom-

icity or availability which are important requirements for a process support system. The sta-

bility and robustness of database systems, for instance, can hardly be found in process-based

environments [AS96]. Furthermore, support for automatic failure handling is usually very

restricted. Most workflow systems, for instance, rely on the administrator to resolve fail-

ures that occur during process execution and lack support for the specification of automatic

failure handling strategies.

## 3.2. Transactional middleware

The main objective of transactional middleware is to enable distributed transactional computing,

i.e., ensure that the transaction concept, which was originally developed for centralized databases,

holds even if multiple disparate and heterogeneous components are involved in a computation. By

using appropriate mechanisms, it is guaranteed that no inconsistencies are introduced in both the

results returned by the application and the participating data stores.

### 3.2.1. Overview

Transaction Processing monitors (TPM) constitute complex environments for online transaction

processing (OLTP) in a distributed fashion [BN97]. The general architecture of such a system,

which is shown in Fig. 3.2, consists of three tiers: The presentation server is responsible for

communicating with clients and handling all aspects of data input and data output. Clients issue

requests to the system, which are then sent to a workflow controller. This component encapsulates

the application logic, translating the request into sequences of calls to several distributed transaction servers, each of which is managing a particular set of resources. Transaction servers can be

database management systems or transactional file systems.

The TPM environment provides a wide range of services supporting distributed transactional computations, with the most important being synchronization services for distributed transactions. Part of each TPM is a distributed transaction coordinator which executes a two-phase commit protocol (2PC) [GR93] at the end of each distributed transaction, ensuring the mutual consistency of the various participating transaction servers. Other functionality provided includes such diverse services such as name serving, load balancing, and automatic server restart. Hence, the TPM is almost a distributed operating system for transactional applications, with some systems, like Encina [Tra95], coming very close in their functionality to distributed computing platforms such as CORBA.
In spite of the fact that the middle tier of the TPM architecture is called "Workflow Controller", it has little in common with the functionality usually provided by a workflow management system. The "workflow" is usually implemented in a scripting language or in a programming language which has been extended with a specific application programming interface. Some systems provide a remote procedure call (RPC) mechanism for the interaction with transaction servers, while others rely on queues as the communication mechanism. Hence, the programming model is fundamentally different from the "meta-programming" approach we envision for a process support system. Moreover, there is no provision of the modeling, monitoring, and logging services which are essential parts of a process support system.

3.2.2. Comparing transactional middleware and PSS concepts

The functionality provided by transactional middleware constitutes only a small fraction of the services needed in a process based environment. The following issues are especially important:

- As we already noted, the programming model is fundamentally different. Writing applications in a TPM environment requires, in most systems, the development of C programs with embedded RPC calls to the transaction managers. Even in systems with a dedicated workflow language, there is no support for re-use based modeling, online monitoring, or history logging.

- The range of systems which can be incorporated into a distributed computation is restricted to those providing a dedicated interface enabling them to participate in a 2-phase-commit protocol. This so-called XA-interface [GR93] is usually only provided by DBMS and related systems. In a generic PSS, a much wider range of applications has to be integrated.

- The transaction model used is very limited. As we will point out in chapter 5, very flexible transaction structures are needed in order to effectively support distributed processes. Because TPM are focussed on online transaction processing, their transaction model is not suitable for many applications with other characteristics. If applications are of long duration or contain non-transactional components, the traditional transaction model is not suitable anymore.
3.3. Distributed computing platforms

Distributed Computing Platforms aim at alleviating the task of building distributed applications in heterogeneous environments by providing a set of tools that simplify the interaction between distributed components and allow for easier systems management.

3.3.1. Overview

The proliferation of distributed environments in the last 15 years led to the development of toolboxes to support the development of distributed applications. One of the first commercially accepted approaches in this regard was DCE, the *distributed computing environment* which has been standardized by the *Open Software Foundation* [Ope95]. The principal architecture of DCE is shown in Fig. 3.3. DCE provides an additional layer between distributed applications and heterogeneous operating systems and communication services. The services provided are centered around an RPC mechanism that is the core communication and coordination paradigm for both applications and the DCE components. It is accompanied by a number of other abstractions hiding the heterogeneity of the underlying platform from applications, like a network-wide naming service, a distributed file system, a global time service, and systemwide security mechanisms. Programming is also simplified through the introduction of threads. Application programmers can use the mechanisms provided to write RPC-based distributed applications, which can be arbitrarily distributed over heterogeneous computers. DCE provides both programming support (for instance, through an interface definition language and stub generators accompanying the RPC mechanism) and administration modules allowing to start components, administer security information, and manage the system configuration.

CORBA [Obj92] has goals similar to DCE, but it is based on objects and has a more open design (Fig. 3.4). CORBA's architectural concept, the *Object Management Architecture (OMA)*,
defines a basic framework rather than a complete system, with a strong focus on extensibility. The
design is centered around the object request broker (ORB) which is an “object bus” allowing to
invoke methods of remote objects. Communication is, like in DCE, based on remote procedure
calls. The set of possible services has been largely enhanced, with the option to add new services
when needed. All services reside in the common object services layer which is accessible to all
system and application components. Note that one of the services defined is the transaction ser¬
vice, which provides functionality similar to a TP monitor. In addition to these commonservices,
common facilities are defined, which are mainly standardized interfaces to external system com¬
ponents, like user interfaces or databases. It is important to note that, at present, many of the services
are not yet implemented and some are not even fully defined. Hence, the CORBA architecture can
be seen as a blueprint rather than a complete system definition.

A further distributed computing platform, Microsoft’s DCOM, pursues similar goals as CORBA,
but with a strong focus on homogeneous, MS Windows-based environments. Because of its func¬
tional similarities to CORBA, we will not discuss it in detail here.

3.3.2. Comparing distributed computing platforms and PSS

Distributed computing platforms such as DCE or CORBA provide a number of useful services
that could be used in a process support system. They fail to address, however, most of the prime
concerns existing in such a system. While a possible option would be the implementation of a
PSS on top of these platforms (which has been proposed, for instance, in [MSKW96]), this would
still require the implementation of most PSS services. The main limitations of the platforms with
regard to process support are:

- Like in the case of TP monitors, there is no support for high-level programming. The RPC-
  based programming model allows for the easy implementation of client-server applications,
  but becomes complicated if other programming models are to be used. Consider our initial
  example of the GEO application. Modeling it using RPC would require a large number of
  “workarounds” leading to a hardly recognizable application logic.

- Many of the services needed in a PSS environment (load balancing, scheduling, logging,
  transaction mechanisms) are either not part of the architecture (like in DCE) or are only
defined as system extensions (like in CORBA). Adding these extensions comes near to
implementing a further middleware component. This is, for instance, the case with the
3.4. Coordination languages

Coordination languages aim at enhancing traditional sequential programming languages with elements allowing to control distributed and parallel computations. The idea behind this is that traditional programming languages are well-suited to describe computations, but not for the specification of coordinations, i.e., the interaction between parallel and distributed threads participating in a computation.

Consequently, coordination languages have been developed as orthogonal extensions to programming languages, providing only coordination and communication facilities. In this, they are similar to process modeling languages which also focus on describing the control and data flow between tasks, but do not perform computations themselves.

The perhaps first coordination language developed was LINDA [GC92]. Its main contribution is a sophisticated communication mechanism which is based on the concept of a tuple space, a global data area through which processes exchange information [Weg96]. Synchronization of processes is realized only by means of the tuple space, by blocking processes when they access certain data items. Linda is a coordination model rather than a coordination language. For practical use, the primitives for accessing the tuple space are embedded into computation languages, and a runtime system is responsible for establishing the inter-process communication. Today, LINDA is available as a commercial product, and several extensions towards various directions exist, including computer supported cooperative work (CSCW) (Sonia) [Ban96] and persistence/transactions (Persistent Linda, [AS91]).

The underlying concepts on which coordination languages have been built are manifold, ranging from the aforementioned tuple space over Petri Nets to linear logic [Weg96]. A coordination model closer to the needs of a process support system is that of ConCoord [Hol96], which is based on centralized control and is thus more suited for megaprogramming. ConCoord's coordination language CCL provides language constructs for instantiating components (which are sequential programs written in a computation language) and controlling their state. In addition, the model provides communication primitives allowing the exchange of information between tasks. Hence, the concept of ConCoord is similar to that of OCR (described in the next chapter), OPERA's process modeling language.

Comparing the concept of coordination languages and process support systems, we can state that some of the goals of the PSS concept have also been pursued by coordination languages. In particular, the focus on coordinating distributed and parallel computations is inherent to both classes of applications. A number of important differences remain:

- The type of system we envision as a process support environment goes far beyond what is provided by a coordination language. Process based environments require not only means for modeling the interaction of parallel threads, but also a large number of additional services (like monitoring, logging, and optimization).

- Nevertheless, a coordination language could be used as the base for a process support system (replacing, for instance, the core modeling language OCR in OPERA). The problem with existing languages (like LINDA or ConCoord) is, however, that they require the integration of special primitives into programs. This contradicts one of the main postulates of process support, which aims at re-using existing programs without modification.
Hence, if a coordination language was to be used for process support, a number of important modifications would be necessary. In addition to the aforementioned need to integrate existing programs as-is, an important requirement is centralized control, since it is the basis for many of the advanced services provided by the process support system (like persistence and availability). Of the two languages sketched above, for instance, ConCoord seems to be a candidate, while LINDA relies too much on distributed control.
Part I.

A Process Model for Reliable Distributed Computing

We dissect nature along lines laid down by our native languages.
Benjamin Lee Whorf, in [Who56]

Real programmers can write assembly code in any language.
Larry Wall, developer of the Perl language
4. Basic Process Model

The modeling language impacts most of the functionality of a process support system. It defines the system's view of the "real world", describing programs, data objects, computers, and resources in a distributed system. It serves as a command language specifying the order of execution and data flow between the tasks. It plays the role of a canonical representation on which the exchange of process information between distributed components is based. A poor language design will necessarily result in poor system performance and will limit the functionality which can be provided. This is very important for a PS kernel which has to be as extensible as possible. The aim of this chapter is the description of a language concept for a PSK supporting all these aspects. The model developed in this chapter will serve as the basis for extensions which are presented in subsequent parts of the thesis.

The chapter is structured as follows. In section 4.1, we give a catalog of requirements that lead to the design of OCR (Opera Canonical Representation), which is described in detail in section 4.2. Section 4.3 provides an example, and section 4.4 demonstrates the mapping of application-oriented modeling languages to OCR. Section 4.5 concludes the chapter with a discussion of OCR's properties.

4.1. Requirements

4.1.1. Multiple process representations

It is important to note that postulating an "universal process modeling language" would not be a good idea. The application areas for process support differ too widely to allow this: each area has specific characteristics and problems which have to be reflected in the language. As an example, compare the differences between business processes and scientific applications described in chapter 2. In a business environment, it is important to capture properties such as the organizational structure of a department or the business rules to be enforced. This information is used as the basis for control flow and scheduling decisions when a process is executed. In contrast, in a geospatial application, the control flow is based on the dependencies between data objects and the scheduling has to take into consideration the actual load on the different machines as well as the resource requirements of the programs. Hence, the "view of the world" that has to be encoded with the language differs widely. The provision of a single language for all potential domains would require a language design too generic to be useful, a problem which is similar in programming languages: Assembler languages are expressive enough for all possible programs yet, nobody uses assembler for every-day programming because higher programming languages support a much more comprehensible way of programming, allowing for more efficient development and maintenance of programs.

Hence, there are good reasons for a variety of problem-oriented modeling languages which allow process designers to specify a process and its requirements in a familiar way. The support for manifold languages requires, however, the existence of a common low-level language in the
4. Basic Process Model

Kernel to which the problem-dependent specifications can be mapped prior to execution. Such an "assembler language of the PS kernel" has to be flexible enough to express most of the possible applications.

Fig. 4.1 shows the different process representations used in a process support system. The graph shows a three-tier hierarchy of paradigms. At the top are the problem-specific representations as they are available for users to model their processes in an adequate and familiar way. The nature and number of these languages cannot be determined a priori, and the system should allow to add new representations for new application domains. At the intermediate layer is the canonical representation, used by the inner system components for navigation and other core services. The external representation of a process is mapped into the canonical format through a compilation process. In most applications, there will also be the need for a "feedback loop" from the conceptual level to the application-specific representation. This is required in all cases where external monitoring tools are deployed to control the state or progress of processes as they are executing. Such a monitoring tool should be based on the problem-oriented representation rather than the internal format, since for effective modeling, users need to have a familiar look of their processes.

The bottom layer of the language hierarchy comprises the database-dependent process representations. PS servers store the state of their processes persistently to provide a basic degree of fault tolerance. Keeping the process states in a database guarantees that they survive server crashes. This is important given the potential long duration of processes, which may very likely exceed the mean time between failures of process servers. To allow a maximum degree of flexibility, the persistent storage should not be tied to a particular format. Instead, the system should allow to use arbitrary databases or (if relaxed consistency is acceptable) even file systems for the persistent storage. The rationale behind this approach is that, in many cases, the installation of
a PSS has to be based on the existing infrastructure. It is unrealistic to assume that a team of scientists who want to use a PSS for experiment support will buy multiple expensive licenses of a commercial database management system. Instead, a solution that uses file systems or public domain databases is needed. On the other hand, it is perfectly acceptable in a business environment to base the PSS on commercial DBMS, if the money to buy them (and the expertise needed for administration) is available. As a general rule, the choice of the persistent storage should be based on three factors: the reliability needed, the existing infrastructure, and the financial means available. A database-independent PSS concept is a necessary prerequisite for allowing these choices.

4.1.2. Canonical representation: requirements

Of the languages described above, the canonical representation is the only format intrinsic to the kernel. Because of its critical importance for the functionality of the PSS, it has to be expressive enough to represent all relevant aspects of a process, and at the same time provide the necessary hooks for adapting it for specific applications. Analyzing the various ways in which the canonical representation is used, we postulate the following requirements:

• **Flexibility**: Due to the fact that the application-specific modeling languages are likely to vary widely, it is indispensable for the canonical representation to be general enough to allow mapping of many different languages. This means that the model for control and data flow specification should be as flexible as possible so that the representation of programs and resources can be easily adapted to each particular application.

• **Extensibility**: The concept of the PS kernel as an extensible system requires to have the possibility of enhancing the process representation whenever this is required by a specific application. For instance, in the example of chapter 2, it is necessary to maintain version information about externally stored data objects. This is application-specific functionality, and the system should make it easy to add it. Extensibility complements flexibility: Even the most flexible language design will not allow the mapping of all possible applications—at least not if its complexity has to be kept at an comprehensible level. In cases in which the flexibility is not sufficient, the system should provide the necessary “hooks” for the incorporation of extensions.

• **Simplicity**: A simple design is an indispensable requirement for maintainable systems, and the complexity of the canonical representation has a direct relation to the overall complexity of the PS kernel. The persistent storage of processes, for instance, needs interfaces translating between the canonical representation and the database-specific formats. The development of new translators can be kept in affordable bounds only with a simple language design. A simple design also alleviates the task of extending the language.

Note that, similar to a coordination language (cf. the discussion in section 3), a process modeling language needs no elements to express computational aspects since all data processing is performed externally. This greatly helps in keeping the language simple, as it allows to concentrate on nonfunctional aspects such as control flow, data flow, and resource requirements.

• **Support for runtime analysis**: Given that monitoring is one of the most prominent features of a process support system, it is important to have the ability to inquire about process states at run-time in an adequate way. The internal process representation has to represent all relevant information in a way that makes it easy to access it.
4. Basic Process Model

- **Support for persistence:** Storing processes persistently leads to a significant run-time overhead. The magnitude of this overhead depends to a large degree on the design of the access mechanism, which in turn depends on the language design. A language design which distributes the information about a process over a large number of small objects will have a poor performance since each object has to be fetched separately from the database. In server-based databases, the number of communication rounds needed to retrieve an object is the dominating factor in terms of performance. Hence, a small number of large objects can be faster retrieved than a large number of small objects. The design of the process model has to take this into account whenever possible.

4.2. OCR: the OPERA Canonical Representation

The key to satisfy the above requirements, especially extensibility, is an object-oriented design based on a framework of classes representing the various components of a process. Using such an approach allows to easily adapt the model by either modifying existing classes or by adding new classes through type extension (inheritance) mechanisms. We will now describe OCR (OPERA Canonical Representation), the internal representation used in the OPERA system, which is based on an object-oriented structure. The main entities of a process (and thus the main components of the process model) are tasks, programs, and resources. The purpose of this section is to give a basic overview of OCR and its semantics. We start with a static view of the various classes which are used to represent processes, and continue with a dynamic view describing how control and data flow are represented.

It is important to note that OCR has two facets: The conceptual representation is a model to represent processes and their components. It can be expressed in terms of a class structure and a description of the dynamic behavior. In addition, a textual representation is needed for practical work. It is generated by modeling tools, parsed by the OPERA system, and directly transformed into appropriate sets of objects and relations. In the following, we will first discuss important properties of the conceptual representation. After this, we will give an example showing how processes are expressed in the textual representation and how they are mapped into an object structure.

4.2.1. Representing tasks

The notion of a task as it is used in OCR encompasses everything that has a behavioral dimension, i.e., tasks are entities which can be executed. This includes whole processes as well as blocks (which are used for the logical structuring of a process) and activities (which are the basic execution steps). Fig. 4.2 uses the UML notation [Rat97] to show the basic class hierarchy used to represent the different task types. The various classes have the following meaning:

*Abstract Classes:*

- **Task:** The abstract root class of the hierarchy, representing all tasks. It defines a set of basic attributes which are included by all other task types through inheritance. These generic attributes are an input and an output parameter list, a variable representing the current execution state of the task, and a description field in which annotations concerning the semantics or pragmatics of a task can be stored for documentation purposes. The contents of this field can be used at runtime, e.g., to inform users about the purpose of a task, and it can be useful during process modeling, for example to support the retrieval of processes from a repository.
4.2. OCR: the OPERA Canonical Representation

- **ComplexTask**: This class represents all tasks which have structure, i.e., consist of other tasks. Two attributes are added to those inherited from Task: The tasklist is a set of references to the component task objects. The whiteboard is a set of data objects, serving as a communication area for the component tasks.

- **ComponentTask**: ComplexTask and ComponentTask are complementary in the sense that the latter represents all tasks which can be included in the former. ComponentTask adds two important attributes. The taskid identifies a task uniquely. The guard is used to specify when the task is executed with respect to other tasks or external events. OCR uses a rule-based mechanism for control flow specification which is discussed in section 4.2.3. There are two new attributes introduced in ComponentTask which pertain to data flow. The inBoxBinding is a list declaring the values to pass to each input parameter. The outBoxBinding declares which of the output values have to be copied into the whiteboard.

**Concrete Classes:**

- **Activity**: Represents the basic execution steps of a task. Each activity has an association to an object describing what has to be executed. This external binding information is included in objects of the class Program, which is discussed in section 4.2.2.

- **Process**: This class is derived from ComplexTask and represents all “top level” processes. The only new attribute is a process identificator which identifies the process uniquely. Process identifiers consist of two long values. The idea behind using a tuple is to support the easy generation of unique identifiers in a distributed system. The first value is an identifier of the node where a process instance is created and the second value is unique within this node. This ensures global uniqueness of identificators without the need for coordination protocols.
4. Basic Process Model

- **SubProcesses** represent one of two possibilities to introduce nesting into the process model. A SubProcess references a proper process, which is instantiated when the SubProcess is to be executed. This variant of nesting provides a late binding mechanism, since the definition of the referenced process can be modified until it is really started. This can be useful if dynamic modifications are required. This behavior can be found in many workflow modeling languages, such as FlowMarks FDL [LA94].

- **Block** is the second possibility to introduce nesting into process models. The class is derived from ComplexTask as well as ComponentTask. This means that a block has structure and can be part of processes or other blocks. The concept of a block is very similar to the concept of blocks in programming languages. They are important for structuring large process descriptions. In addition, blocks can be used to introduce special behavior into the system. While loops, for example, can be realized as blocks with a special termination condition. Blocks will also be used as the basis for the introduction of atomicity concepts in chapter 5.

4.2.2. Representing programs

Program objects are used to represent the external binding of activities, i.e., the “real world” action to be started when an activity has to be executed. These actions are generally referred to as “programs”, although they do not need to correspond with the execution of a single program. It may well be that a program represents the initiation of a human activity or an SQL query sent to a database system. The distinguishing characteristic of a program is that, from the perspective of the PS kernel, it is indivisible.

A program object encodes all information needed to start the external programs, pass necessary information to it, and receive the data returned. The type of this information can differ widely, depending on the type of program and the environment in which it is to be executed. Starting a program on a UNIX computer requires different information than what is needed to invoke a CORBA method, which again is different from the description of a human task. To cope with this heterogeneity, OCR contains a class hierarchy for programs (Fig. 4.3).

The root class of this hierarchy, Program, defines a set of generic attributes, including a unique name, an identifier for the actual class, and parameter lists for input and output data. The parameter lists of a program and the activity that references it have to be identical. The figure shows three examples of derived program classes for different types of applications. A human task may be described by the person responsible for executing it and a textual description of the goal to be
accomplished\textsuperscript{1}. The UNIX program is specified by giving a description of the host on which to execute the program and of the command line describing the invocation. The CORBA method uses yet another set of parameters. They correspond to the information needed by the CORBA dynamic invocation interface [Sei97].

\subsection*{4.2.3. Control flow}

To guarantee flexibility, a rule-like mechanism is used to specify control flow. It is based on the concept of \textit{ECA rules}, which were originally developed for active database systems [PDW+94, WC96] and are descendants of earlier approaches to the rule-based languages developed for artificial intelligence applications.

The control flow inside a process is encoded in \textit{guards} which are attached to each component task. A guard can be seen as a description of \textit{when} a task has to be executed. It is a tuple $(A, C)$, where $A$ is an \textit{activator} and $C$ is a \textit{condition}. The activator is a predicate over the state of the process, its components, and the outside world. It defines when a task has to be considered for execution. The condition part of the task is a predicate over data objects which are visible to the task. The following example, showing the definition of a task in the textual representation of OCR, illustrates the concept:

\begin{verbatim}
TASK t2 ()
  ...
  ACT finished (t1)
  COND t1.returnValue > 7
  ...
;
\end{verbatim}

Here, the activator refers to the final state of task $t1$ (see below for a discussion of task states), meaning that $t2$ is to be executed when $t1$ has successfully finished its execution. In addition to referring to states of sibling tasks, it is also allowed for an activator to refer to external events, for instance a certain point in time. It is also possible to extend the concept to arbitrary events sent by other processes, a mechanism which is described in detail in chapter 8. The condition part of the example above refers to a returned value of the task $t1$. Conditions may restrict the execution of a task based on the value of data objects. The primary purpose of conditions is the modeling of conditional branching.

The main advantage of dividing the control flow specification into activators and conditions is efficiency. The activators are based only on states and events and can be checked using efficient methods like finite state automata or Petri Nets [Gat94]. This prevents having to check all data-dependent conditions each time a data object changes and allows to implement fast navigation.

Another advantage of the guard concept is that it allows to "attach" control flow constructs to the task objects. This avoids to model them as separate entities and keeps the number of "first class" objects small. We have already argued that a small number of entities is important since it simplifies the construction of new interfaces.

Rules as a general programming paradigm have been considered harmful [SK95], especially because it is hard to predict the behavior of large sets of rules. This is mainly due to the fact that in traditional rule systems (like most trigger mechanisms for database systems), possibilities to structure rule sets are missing. This makes predicting the behavior of large rule sets nearly

\textsuperscript{1}Note that this is a very simple form of user specification. In a real business environment, role concepts would be used which allow a smarter assignment of work to people. At this point, we will abstract from these aspects in order not to confuse the reader.
impossible. This problem is avoided in our approach because the scope of a guard is restricted to the enclosing process or block. Hence, the number of rules interacting which each other is small.

Loops are implemented through specialized blocks which have an attached termination predicate. Similar to a while loop in programming languages, the loop is repeatedly executed until the termination predicate becomes true. The whiteboard (see below) provided as part of a block allows to store the necessary data between iterations.

4.2.4. Task states

The basic state diagram given in Fig. 4.4 describes the minimal set of states and the events triggering state transitions. This diagram will be extended in the next chapter, when failure handling is discussed. In its basic form, its ability to reflect failure semantics is very limited, allowing only to capture the fact that a program has failed or was aborted on a user's request. It is important to note the distinction between the initial, the ready, and the running states. A task is initial when it was created, which is usually the case after the creation of its enclosing process. The task becomes ready as soon as its guard evaluates to true. The effect is that the PS kernel will attempt to start the program referenced by the task. Hence, ready indicates that the start of the program has been initiated. Recall that programs execute on external systems, and the PS kernel does not know whether a program has really been started before an acknowledgment is received from the remote system. It is also important to note here that some systems, like FlowMark [LA94], provide an additional state indicating that an application was assigned to multiple users and has to be accepted by one in order to move into the "starting" mode. We have omitted this state here, since it is particular to human activities. Once it has arrived, the state of the task object changes to running and stays in this state until a notification of the program's end arrives (which changes the state to either failed or finished), or the program is aborted by the user. In the case of an abort, the system has to wait for an acknowledge until it can really know that the abort was successful. Note that the state may change from aborting to finished or failed in the case where the user aborts a task while at the same time the program ends at the remote system [Nus98]. The dead state is used to capture the case in which a task's activator was true, but the condition evaluated to false. If this happens, the task is not considered for execution anymore, even if its activator becomes true again. A task is also considered to be dead if all of its predecessors are dead. To detect this case, an algorithm called dead path elimination is used (based on a method described in [LA94].

![Basic state diagram for tasks](image-url)
4.2.5. Data flow

Data inside a process is represented either as part of the whiteboard or as part of a parameter list. To facilitate the exchange of information between programs, it is possible to specify the transfer of data between objects. There are two basic forms of data transfer: The transfer into an input data structure (from an output data structure or the whiteboard), and the transfer from an output data structure (into the whiteboard). The two methods of data transfer are best described by looking at the life cycle of a specific task. When a task becomes ready (i.e., its guard becomes true) its input parameters have to be assigned actual values. Which values they take is specified in the task’s inBoxBinding. A binding specification for an input parameter can either be a constant or a reference to another data object, either in another task’s output parameter list or in the whiteboard. All input parameters must be specified. The following shows the header of a task declaration in the textual representation of OCR. Here, the parameter declaration and the initialization are integrated in the task’s parameter list.

\[
\text{TASK t4 (pi} = 42, \text{p2 = t2.ret1, p3 = } \_\text{WHITEBOARD}_.\text{customername})
\]

\[
\text{STORE (rvl > estimatedAmount)}
\]

Here, the first parameter is initialized with a constant value. The second parameter is initialized with a parameter returned by task t2, and the third parameter is initialized with a value taken from the whiteboard. The example also shows a specification of data transfer to be executed after the task’s execution. If some of the task’s results have to be copied to the whiteboard, this can be specified in the so-called storelist which is evaluated when the task has successfully finished. The storelist consists of tuples, where the first element is the name of an output parameter and the second element is the name of a whiteboard component. Both components must have the same data type.

The mechanism described above allows the specification of all necessary data exchanges inside a process. It has to be noted, however, that this mechanism does not check for correctness. It is possible to specify data flow from objects which have not yet been assigned a value (for example, the output parameter list of a task which was not yet executed). In such cases, the system uses default values. It is entirely up to the process designer (resp. the compiler creating OCR representations) to make sure that no incorrect data transfers occur. This is acceptable given that OCR is not intended as an end-user programming language.

4.2.6. Data objects

OCR has a comparatively simple type system consisting only of a small number of scalar datatypes. Two generic extension mechanisms are provided in order to adapt the language. First, is is possible to extend the set of existing data types by defining new subtypes of the Parameter class. Fig. 4.5 shows the currently existing class hierarchy, which consists of the abstract class DataObject and a number of derived classes for the various data types, including ReferenceObject. This type is the key to the second possibility for capturing new types of data. A reference parameter can store binary data which are not further interpreted by the process support system and can be used to let the PSS handle arbitrary information. This facility was originally introduced to store CORBA object references (which have a textual representation allowing to store and exchange them), but it can of course be used to transfer arbitrary information. The disadvantage of handling parameters through the reference mechanism is that the PSS has no knowledge about the structure of the data, and consequently no information about the transferred values can be stored in the log. It is also
4. Basic Process Model

4.2.7. The whiteboard

The whiteboard is the equivalent to local variables in programming languages. It is a set of data objects which can be arbitrarily written and read by all tasks. This provides a convenient mechanism for the storage and exchange of temporary data, since the whiteboard can be used as a repository for common state information as well as a way to implement counters. To this end, the whiteboard allows simple operations on its data objects, including decrement and increment operators for integer values, and an append operator for string data types.

4.3. Example

The canonical model is accompanied by textual representation used to insert new processes into the system. We will now illustrate the structure of OCR using an example. The application used was borrowed from the WISE project [AFH+99] and represents a process occurring in an insurance company when a new claim arrives. The structure of the process in a graphical form is given in Fig. 4.6. It consists of 5 steps, starting with the ReceiveClaim step which captures the claim data given by the customer. After this, the claim is classified. In this simple example, we assume only two claim classes. In the case of a simple theft or burglary, the payment history of the customer has to be checked before the payment is initiated. In the case of a fire damage, an assessment is
necessary which requires to visit the site and perhaps to consult external experts.

```java
PROGRAM ReceiveClaim()
  RETURNS (name: String, policyNo: String, docRef: String) {
    SYS Unix,
    HOST ik6.inf.ethz.ch,
    COMMAND "/home/opera/receiveClaim.tcl"
  }

PROGRAM ClassifyClaim (name: String, policyNo: String, docRef: String)
  RETURNS (claimType: String) {
    SYS Unix,
    HOST dblab5.inf.ethz.ch,
    COMMAND "/home/opera/classifyClassification.tcl %name% %policyNo% %docRef%"
  }

PROGRAM ValidatePayment (name: String, policyNo: String, docRef: String)
  RETURNS (amount: Integer) {
    SYS SAP,
    HOST sapr3,
    CLIENT "800",
    USER wise,
    PASSWORD opera,
    WORKFLOW "CheckPayment"
  }

PROGRAM DamageAssessment(name: String, policyNo: String, docRef: String)
  RETURNS (amount: Integer) {
    SYS Flowmark,
    HOST dbpc,
    USER admin,
    PASSWORD "password",
    SERVER exmsrv,
    DATABASE insuredb
  }

PROGRAM PreparePayment(name: String, policyNo: String, docRef: String, amount: Integer)
  RETURNS () {
    SYS Unix,
    HOST ik6,
    COMMAND "/usr/bin/doPayment %name% %policyNo% %amount%"
  }
```

Figure 4.7.: OCR textual representation of example, part 1: program registrations

The representation of this process in textual OCR has been divided into two parts, the registration of external programs (Fig. 4.7) and the specification of the process structure (Fig. 4.8).

External applications are registered with the system using the `PROGRAM` clause. An entity described through such a clause will be internally represented by an object of the `Program` class or one of its subclasses. Which subclass to select is determined by the `SYS` keyword in the declaration. In the example, the program classes `Unix`, `Flowmark`, and `SAP` are chosen, leading to the generation of respective objects. Note that the structure of the data provided varies between the program types: The declaration of the Unix program requires only a host name and a program name, while in the case of SAP, it is necessary to pass authorization data such as user name and password as well. The required parameters are defined by the respective `Program` subclass.
4. Basic Process Model

PROCESS Insurance ()
  RETURNS ()
  WHITEBOARD (policyNo: String, name: String, claimType: String,
  docRef: String, amount: Integer),

TASKS
  TASK receive: ReceiveClaim ()
    STORE (policyNo -> WB.policyNo),
    ACT initial (STARTUP),
    COND true
  
  TASK receive: ClassifyClaim (name = WB.name, policyNo = WB.policyNo,
  docRef = WB.docRef)
    STORE ()
    ACT finished (startup),
    COND true
  
  TASK validate: ValidatePayment (name = WB.name, policyNo = WB.policyNo,
  docRef = WB.docRef)
    STORE (amount -> WB.amount),
    ACT finished (classify),
    COND WB.claimType = "SimpleTheft"
  
  TASK assessment: DamageAssessment (name = WB.name, policyNo = WB.policyNo,
  docRef = WB.docRef)
    STORE (amount -> WB.amount),
    ACT finished (classify),
    COND WB.claimType = "FireDamage"
  
  TASK payment: PreparePayment (name = WB.name, policyNo = WB.policyNo,
  docRef = WB.docRef, amount = WB.amount)
    STORE (),
    ACT finished (validate AND assessment),
    COND true

END TASKS
END PROCESS

Figure 4.8.: OCR textual representation of the example, part 2: process specification
4.4. Mapping external languages to OCR

In addition to the invocation information, the input and output parameter lists are declared for each application. It is easy to see in the declaration of ClassifyPayment how parameters are passed to an application by embedding the names into the command string, enclosed between "%" characters. At runtime, these parameter names are replaced with the actual values passed from the execution context. Once a program is registered with the system, it is persistently stored and accessible under the specified name. Programs can be re-used in an arbitrary number of processes.

The specification of the process itself (Fig. 4.8) consists of two parts: a process header and the task declaration. The header defines the interface of the process (which is needed if it is embedded into other processes through the subprocess mechanism) and the contents of the whiteboard. In the example, the whiteboard contains all necessary case data, like customer name and policy number and a reference to additional documents describing the claim. The example nicely illustrates all relevant aspects of OCR:

- **Program invocation**: Each task declaration contains a specification of the program to be invoked. The name must refer to a previously declared Program object. Referencing programs by name allows for a late binding mechanism which permits to replace program definitions independently of processes which may use them.

- **Control flow**: The keywords ACT and COND are used to specify the execution order according to the rule-based model used in OCR. For instance, in the declaration of the validate task, the activator refers to the end of the classification task, while the activation condition makes the actual invocation dependent on the claimType variable of the whiteboard. Only a value of SimpleTheft leads to an execution of validate. On the other hand, assessment is executed if the value is FireDamage. The special activator value initial (STARTUP) used in the receive task indicates that this task has to be started on process instantiation.

- **Data flow**: Data to be passed to a program is specified as part of the task declaration directly after the name of the program to be invoked. In the example, all data flow occurs from or to the whiteboard (which is referenced by the reserved symbol WB), although it would also be possible to access the output parameter list of a task directly. The STORE clause of a task declaration defines which return values to store in the whiteboard after task completion. In the example process, all tasks but the last one return a value, which is stored in order to complete the case data.

The example shows how the textual representation of OCR provides a flexible language for specifying all aspects of process execution. The rule based mechanism for control flow and the flexible data flow enabled through the whiteboard allow to express a variety of situations. Again, it is important to recall that OCR is not conceived as an end-user language, but as an intermediate format to which graphical specifications or other problem-oriented process descriptions can be mapped. The textual form is simple, yet flexible and can easily be produced by modeling tools. This has already been demonstrated in the WISE project [AFH99], where a commercial modeling tool based on a Petri Net formalism has been extended to produce OCR representations of the processes.

4.4. Mapping external languages to OCR

As a proof of concept, we will show in this section how an external process modeling language can be mapped to OCR. Of the many languages and paradigms which may be used in OPERA extensions, we use a workflow modeling language. The Workflow Management Coalition, an
4. Basic Process Model

association of workflow vendors and users trying to standardize workflow concepts and system interfaces\(^2\), has defined a generic workflow model [WMC94a]. This model is, however, too generic to be used for actual process modeling. A model that comes close to the reference model is that of the commercial Workflow Management system FlowMark [LA94], which will be used in this section. We start by giving a short overview of the model and will then describe how it can be mapped to OCR.

4.4.1. The FlowMark process model

Fig. 4.9 presents the main building blocks of the language. A process consists, similar to OCR, of activities, blocks, and subprocesses. The mapping of these entities to OCR is thus straightforward and will not be described in detail here. What is different is the way in which control and data flow are modeled. Control flow is based on three types of components, control connectors, transition conditions, and start conditions. A control connector (CC) is a directed edge linking two tasks (the source and target task) and specifying the order of execution between them. Each CC can be marked with a transition condition restricting the execution of the target task. This can be an arbitrary boolean condition over the output data of task which are backward linked through control connectors. The target activity is executed only if the condition evaluates to true. Start conditions are used if an activity has multiple incoming control connectors. The start condition determines in this case which of these connectors have to evaluate to true in order to activate the target activity.

Data flow is modeled through data connectors which link data containers. A data container is equivalent to an OCR parameter list, and each task has an input data container specifying its input data and an output data container specifying its output data. The data connector specifies the flow of information between containers. Upon connecting the output container of one activity with the input container of another activity, it can be specified which parameters have to be mapped to each other. If both containers have the same data type, the mapping is done automatically.

\(^2\)A good entry point for general information about the WfMC is their web page, to be found at \url{http://www.wfmc.org}
4.4.2. Mapping FlowMark's model to OCR

Generating an OCR specification \( S_O \) from a FlowMark specification \( S_F \) can be done based on the following rules:

1. For each activity, block, and subprocess in \( S_F \), a corresponding element in \( S_O \) is created.

2. If a control connector exists in \( S_F \) between \( T_1 \) and \( T_2 \), and \( T_2 \) has only one incoming connector, the activator of \( T_2 \) is set to the predicate \( \text{finished}(T_1) \).

3. If a task has multiple incoming control connectors, the corresponding \( \text{finished}(T) \) predicates are connected by logical connectors according to the start condition of the task in \( S_F \).

4. Transition conditions in \( S_F \) are mapped to conditions in \( S_O \). If a task has multiple incoming connectors, the conditions are concatenated by logical operators in the same way this was done for the activator.

5. Input containers and output containers in \( S_F \) are mapped to input and output data structures in \( S_O \).

6. Data connectors are mapped to startup data bindings in a straightforward manner. Care has to be taken only if at a join node (one with multiple incoming data connectors), multiple source attributes from different output containers are mapped to the same target attribute. This can be specified in the FlowMark system, although it is not documented. In \( S_O \), it is not possible to initialize an input parameter with multiple values. The problem can, however, be solved by using the whiteboard. The multiple data objects can be stored in the whiteboard and read from there by the task upon startup. Note that if multiple values are written to the field, the order is completely dependent on the execution order of the respective tasks. The correct behavior in FlowMark is, however, not documented.

The rules given above can be used to map arbitrary FlowMark flows to OCR specifications. Obviously, a complete implementation of the Flowmark model would also require to extend the task objects in order to represent all the attributes that are attached to a task in FlowMark in order to model all aspects of a business process.

4.5. Discussion

The modeling language presented meets the design goals listed at the beginning of this chapter. To illustrate its extensibility, which is the most prominent requirement given the kernel property of OPERA, we will discuss the various means by which it is possible to extend OCR. There are several interfaces which allow the easy extension of the system. We will give short examples for each of these interfaces in this section. The extensions are all based on extensions of the object-oriented specification of the OCR representation.

- **Changing process and task structures:** Since all information about processes, blocks, and tasks is encapsulated in object-oriented classes, it is simple to extend the representation of tasks by adding new attributes. This also requires to modify the database and network adapters which translate the data types into external formats. Due to the comparatively simple class structure, this does not require much effort.
• Adding new datatypes: The set of available data types has been kept simple in order to keep the system design comprehensible. Moreover, the set of existing data types seems to be sufficient for the vast majority of applications. If more types are needed, they can be easily added by defining new subclasses of the DataObject class. This simply requires the definition of the data components and the implementation of a comparatively small set of operations which are used by the system to manipulate parameters. In addition to the introduction of this new types in the system kernel, it is necessary to write transformation methods that allow to store objects of the new types in databases and to send it to other machines.

• Adding new types of external applications: Because of the representation of the set of external programs as a class hierarchy, it is possible to add new program types by creating new subclasses of the Program class. These subclasses have to implement a number of generic operations which are defined only as abstract methods in the superclass. In addition, it is necessary to add new code capable of communicating with external applications. This is discussed in detail in chapter 9.

• Adding blocks with new semantics: The block construct can be used as the basis for many semantic extensions to the current model. For instance, if the existing loop construct (while loops) is not sufficient, it would be possible to add a new type of blocks that implements for loops. This can be done by creating subtypes of the Block type and redefining the behavior of this new block. To this end, some of the inherited operations need to be overridden.
5. Atomic spheres

5.1. Introduction and motivation

Distributed processes are susceptible to failures because of the sheer complexity of the environment in which they are executing. Communication problems, computer outages, or program failures are some of the technical sources for errors during process execution, and the probability that such an error occurs grows with the number of components participating in a process. Erroneous process specifications, unexpected changes in the system configuration, or mistakes while performing tasks are “man-made” errors, which are also more likely to occur the more complex a system becomes. With this high probability of failures, it is desirable to have processes that are able to continue executing in spite of failures.

In their general classification of system dependability aspects, Laprie et al. [Lap92] distinguish two means for failure procurement in dependable systems, fault prevention and fault tolerance. While the former is concerned with “how to prevent fault occurrence or introduction”, the latter deals with “how to provide a service complying with the specification in spite of faults” [Lap92]. Fault prevention is to a large degree a design issue – it requires the existence of suitable design methodologies and construction rules which help to avoid introducing failures into a system. In a process support context, validation facilities like process simulation can support the prevention of faults. A complete avoidance of failures is, however, not possible. Hence, there is a need for fault tolerant processes, i.e., processes that are able to continue execution even if failures occur.

In this and the following chapters, we will discuss the problem of fault tolerant processes and extend the basic process model described in the previous chapter with features allowing the specification and execution of failure handling. We begin with a motivating example and a general discussion of the problem of fault tolerance in process support contexts. The remainder of this chapter will then present the concept of atomic spheres as a first step towards fault tolerance. In the following chapter, we enhance the sphere concept with exception handling mechanisms. Chapter 7 introduces a validation mechanism for fault-tolerant processes and discusses our solution for fault-tolerant processes as a whole.

5.1.1. Motivation

As a running example, consider a process incorporating the reservation of various flights, rental cars and accommodations as well as sending the invoice to the customer and the storage of the result in the travel agency’s internal database (Figure 5.1). The programs and services incorporated in the process are executed by different autonomous systems: Flight reservation is done through a CORBA gateway to a booking system. Sending the invoice as well as reserving a hotel are manual tasks handled by the travel agency’s personnel. Record keeping in the local database takes place via a TP monitor, and the reservation of a rental car is done through a legacy system. In such a scenario, possible failures can be classified into several categories: Program failures, design and system failures, and semantic failures.
5. Atomic spheres

- **Program failures** lead to the abort of an external application. For instance, the flight reservation step can fail because no seats (or no seats in the requested category) are available; manual tasks like Invoice may have an associated time limit which may be violated; Record Keeping may fail because of a transaction abort, and so forth. A fault tolerant process has to provide handling strategies for all these situations. In the case of unavailable seats the solution may be to ask the customer to select a different flight, to accept a different category, or to cancel the whole reservation process. In the case of temporal constraints, tasks could be automatically dispatched to a different employee or a supervisor notified. The transaction failure may be solvable by simply retrying execution.

Note that often the external application provides some level of fault-tolerance. A database application, or the database system itself, may automatically reschedule an aborted transaction. While this simplifies the work to be done by the PSS, it does not solve the problem of more complex failure handling strategies. For instance, using a train reservation as a contingency plan for the flight/car reservations needs a combination of backward and forward recovery (we assume here that no rental car is needed if a train is booked since the railway station is central enough): If a car has already been reserved, but no flight is available, recovery requires that the reservation be canceled before the train is booked.

- **Design and system failures**, such as incorrect parameters for program invocation or impossible program executions because of configuration changes, must also be considered. If the process programmer has erroneously provided an incorrect program name, object id, or host address, or if an application has been moved from one host to another, the invocation will fail. Such failures must be captured by the process model in order to allow the specification of flexible reactions. The same is true for communication failures, which may prevent the invocation of a program. The difference between the two is that design errors usually lead to aborting the process, while communication failures may be resolved by re-trying execution at a later point in time or by choosing a different host.

- **Semantic failures** are the most interesting kind of exceptions. The execution of a program without errors does not always mean that it was successful, since there may be additional constraints. In the example given, the checkAvailability service returns successfully even
5.1. Introduction and motivation

If no available seat has been found. This case is, however, an exception since without available seats no reservation can be made. It is up to the process code to detect this and invoke appropriate measures. Thus the process system needs mechanisms to define what is regarded as a semantic exception in order to incorporate them into the general exception handling scheme. We will discuss these issues in section 6.3.

5.1.2. Fault-tolerance in process support systems

There are three aspects of fault tolerance that must be taken into consideration. Fault tolerance of the PS kernel requires that the system can continue to work even if some components fail. This aspect is discussed in chapter 10. Fault tolerance of the underlying components, i.e., of the computers, communication lines, and programs in the distributed system, is an important issue as well. The deployment of fault tolerant components will contribute to the overall dependability of a distributed process since it reduces the probability of failures becoming visible to processes. The process support system has thus to offer mechanisms through which dependable processes can be built over undependable components. Finally, fault tolerance of processes requires the enhancement of process descriptions with specifications how to react to failures. This is the focus of this and the following two chapters, where we will concentrate on language aspects of process fault tolerance, i.e., on linguistic abstractions that aid the process designer in modeling fault tolerant processes.

Aborting a process in the case of an error leaves it in an undefined state, with only parts of its goals met and external resources possibly inconsistent. This is particularly undesirable for processes which are very long, consist of expensive tasks, or access multiple external data repositories. To allow a process to continue after a failure, it is necessary to extend its specification by including failure handling directives. Such directives can specify, for instance, the compensation of already executed activities in order to restore the consistency of external resources and the execution of contingency plans instead of failed activities.

One way to do this is to integrate the necessary steps for handling failures directly into process descriptions using conventional modeling languages [AKA+96, Arn96]. This, however, is not advisable. Figure 5.2 shows a slightly simplified representation of our initial travel planning process using the modeling language of FlowMark [LA94], a workflow management system. On the left side, we display the pure “application logic” without failure handling, while on the right side, an attempt has been made to include all necessary steps for handling the failures mentioned above. The model includes recovery mechanisms like compensation (the Cancel_Flight activities) and contingency tasks (Hotel2 is a replacement for Hotel1, ReserveTrain replaces BookFlight...
5. Atomic spheres

and RentCar) as well as partial rollback (compensation of BookFlight if no car is available and subsequent booking of train).

The interleaving of the original tasks with the recovery steps makes the fault tolerant version very complex and the original process logic hardly recognizable. The complexity explosion in this very small example points out the drawbacks of including exception handling as part of the normal process specification. Mixing business logic and exception handling logic makes it difficult to keep track of both, complicating the verification of processes as well as later modifications. Moreover, such an approach makes it almost impossible to reuse components since they will lack meaning out of the context for which they were originally designed. One of the key features of process systems is the ability to reuse subprocesses, very much in the same way that libraries are used in programming languages. This can only be accomplished if there is some form of system support for exception handling that allows to separate it from the normal code.

The need for fault tolerance is of course not particular to process specifications. The lack of suitable abstractions that alleviate the task of writing fault-tolerant code has been recognized early, and a number of concepts have been developed over the past 30 years. Of particular interest for the PSS context are failure atomicity as it is used in database transactions and structured exception handling mechanisms as they were developed for programming languages. The concept of atomicity in its basic form guarantees data consistency in the presence of failures, since it is based on the rollback of work until a previous, consistent state is reached. Structured exception handling mechanisms tackle the problem of interleaving application logic and failure handling in process specifications. They allow to write fault tolerant, yet comprehensible processes. Integrating both techniques delivers a flexible failure recovery mechanism for distributed processes that allows to model a large variety of failure handling strategies.

5.2. Overview: The notion of atomicity

The idea of grouping sets of operations into atomic units dates back to early operating systems research in the 1970s, where concepts like Recovery blocks [HLMSR74] and Spheres of Atomicity [Dav78] were developed. The original idea of a programming abstraction to be supported by the operating system, however, had no success. Ensuring atomicity becomes expensive if concurrent activities have to be taken into account, and any reasonable mechanism would lead to a poor operating system performance. At the same time, many applications may not even need this functionality. Hence, instead of implementing it as a general service, the concept was applied at the application layer, and integrated into database management systems. One of the main purposes of DBMS is to guarantee the consistency of data, and a certain overhead is accepted. Both factors were important prerequisites for the successful integration of atomicity into the transaction concept.

5.2.1. The original transaction concept: strict atomicity

The transaction concept as it is used in database systems [BHGR, GR93] defines a set of abstractions which simplify the construction of correct applications (transactions) accessing shared resources. Its main properties are usually summarized in the ACID property, meaning atomicity, consistency, isolation, and durability. From the perspective of fault tolerance, these notions have the following meaning: Durability protects from the system failures that may occur after a transaction has completed. The system guarantees that no effects of completed transactions disappear, even if the DBMS or parts of it fail. This is an abstraction that frees the programmer from the need to perform their own bookkeeping about completed transactions. Isolation protects from failures
that may be introduced because of concurrent access to shared resources. It is usually imple-
mented through locking mechanisms, and can be seen as an abstraction freeing the programmer
from the need to coordinate data accesses explicitly with other transactions, which is an impor-
tant improvement over other methods for the synchronization of concurrent processes, such as
semaphores, critical sections, or monitors. Consistency requires a transaction not to cause failures
if the underlying system is failure-free, that is, a transaction must be consistent per se.

Finally, atomicity protects against failures that occur while a transaction is executing. It is
based on the simple principle that a transaction that encounters a failure is aborted and rolled
back, leaving the database in the state it had before the transaction was started. This abstraction
receives the programmer from the need to program recovery steps. Instead, the system is responsible
for purging the data and bringing the database into a consistent state every time a failure occurs.
Many techniques for achieving this have been developed, with the most common being the logging
of a transaction's operations and the undo of these operations if a failure occurs [BHG87].

In contrast to isolation, atomicity can be enforced in distributed environments with compara-
tively little overhead. Distributed commit protocols such as two- and three-phase commit [BHG87]
allow to implement distributed transactions by providing the means to reach consensus about
whether to abort or commit a distributed transaction. The two-phase commit protocol has also
been standardized by the Open Group [Gro92] and is used in many commercial database manage-
ment systems and TP monitors. It requires, however, the data repositories to be able to defer the
commit of changes until the distributed nodes have reached their decision about commit or abort.
This makes the protocol hardly usable for process support environments where the participating
systems are not necessarily databases and hence might not provide the required functionality.

The transaction concept in its original form is applicable only for online transaction process-
ing (OLTP) applications, where a large number of short transactions is issued against one or more
databases. It was recognized very early that the concept is not suitable for many other applications
because of a too strict notion of isolation and atomicity [Gra81]. From the perspective of atomi-
city, the main drawback is that in the case of a failure, all work has to be undone. This becomes
unacceptable with long running activities, since an undo of everything already performed and the
subsequent re-execution would be too expensive. With the original transaction concept, however,
the scope of a rollback can only be limited if a series of operations is partitioned into multiple
transactions [KP92]. This has the advantage that in the case of a failure only the last small trans-
action has to be undone. It does not allow, however, to undo multiple small transactions if this
should be necessary, since they already committed.

5.2.2. Advanced transaction models: relaxing atomicity

Because of the known deficiencies of the transaction concept, various advanced transaction mod-
els have been developed in the late 1980s and early 1990s [Elm92]. They aim at weakening some
of the transactional properties in order to achieve a higher performance, a better comprehension
of application semantics, or more flexibility. Of the models relaxing the atomicity requirement,
Sagas [GS87], closed nested transactions [Mos85] and flexible transactions [ELLR90, ZNBB94]
will be discussed in detail here.

The Saga concept [GS87] allows to partition long running activities into smaller units. A saga
is the execution of a set of transactions. In the case of a failure, it is possible to abort only the
currently running transaction, preserving the results of its predecessors. If more work has to be
undone, previous transactions can be logically undone by executing compensating transactions.
If necessary, a whole saga can be rolled back by executing the compensating transactions for
its components in reverse order. Compensation as a means to undo work that has already been
5. **Atomic spheres**

committed is a concept to be found in most transaction models that relax isolation, since it allows to commit partial results of a long running activity, while at the same time preserving the atomicity of the whole computation. The ability to support *partial rollback* is an important building block for flexible recovery schemes.

*Closed nested transactions* [Mos85] can be seen as a generalization of earlier ideas for weakening atomicity, like chained transactions or checkpointing concepts. The original flat transaction model is extended to a nested model, allowing to specify an activity as a nesting of transactions, where each subtransaction can be aborted independently of the others. Subtransactions are dependent on their supertransactions, meaning that when a transaction aborts, all of its subtransactions are aborted as well. The closed nested transaction model has the disadvantage that a complete transaction tree is isolated against other transactions, which leads to performance problems if transactions are long running. To alleviate this problem, open nested transactions were proposed [WS91] which relax isolation and atomicity. Preserving correctness in open nested transactions is, however, difficult to achieve [ABFS97]. Nesting was also introduced into the Saga concept in the form of nested sagas, which can be seen as a special case of open nested transactions.

Most transaction models assume implicitly that the tasks of a transaction are atomic, i.e., can be undone in the case of a failure. In real-world environments, however, irreversible effects may be caused. These effects may often not even be compensatable. To cope with this problem, the concept of *flexible transactions* was developed. It provides also mechanisms for the specification of forward recovery mechanisms. The model is described below.

### 5.2.3. The FLEX transaction model: exploiting application characteristics

The FLEX transaction model [ELL90, ZNBB94] was originally designed as a means for controlling multidatabases, i.e., database systems composed out of autonomous database management systems. The problem in such a system is similar to that in process support systems: the participating sites might not support the notion of atomicity. The model does not presume that any of the participating systems has such properties. Instead, it tries to enforce atomicity based on the actual properties of the component activities.

Flexible transactions are flat, and their component tasks are assumed to be transactions themselves (i.e., they are atomic). Component tasks are classified into a number of categories: *Compensatable* tasks can be logically undone by executing an inverse task. *Retriable* tasks can be rescheduled for execution in the case of a failure and will eventually succeed. *Pivot* tasks are neither compensatable nor restartable.

A flexible transaction consists of a set of tasks and two relations between them: The *precedence order* defines the “normal” control flow between tasks. The *preference order* relates equivalent tasks and defines alternative executions in case of failure. Take as an example the flexible transaction in Fig. 5.3, borrowed from [ZNBB94]. Each box is a task, and the arcs represent the precedence order, i.e., T3 has to be executed after T1, T5 after T3, etc. The precedence order is represented by the numbers attached to some of the arcs. For instance, after T3 has finished, T5 and T6 are started since they have the highest priority in the precedence order. If a failure occurs during the execution of T3, T6, or T7, the transaction is rolled back to the end of T3 (if T5 or T6 have already finished, they are compensated), and the next element in the preference order, T8, is executed. Flexible transactions weaken the atomicity requirement by introducing *semi-atomicity* as a new criterion. According to semi-atomicity, the successful execution of a transaction does not require that all tasks have to succeed. It is sufficient if one execution path is successful. This allows to try several paths according to the preference relation until one that leads to a successful termination of the transaction is found.
A well-formed flexible transaction is one which preserves atomicity in all cases. The definition of well-formedness is given in [ZNBB94], and we will not repeat it here. It is based on the principle that after the successful execution of a non-compensatable task, the transaction is doomed to succeed, i.e., there must be an execution path that leads to a successful termination of the transaction.

5.2.4. Discussion of the FLEX model

Compared to other transaction models, the flexible transaction model has two properties which make it a good candidate for the definition of fault tolerant processes. First, it takes into account varying characteristics of component tasks. Second, it incorporates a recovery mechanism that allows the specification of various failure handling strategies, including partial backward recovery and the execution of contingency tasks. There are, however, a number of limitations which show that the model was primarily designed as a theoretical means to study the behavior of complex applications. These limitations need to be solved to render the model useful in practice:

- The set of available task execution characteristics (compensatable, retrievable) is very restricted. In “real” applications, there are more properties that have to be taken into account.
- The recovery mechanism distinguishes only between failure and success, i.e., it is not possible to declare multiple failure handling strategies depending on the type of failure that occurred.
- The method for modeling recovery (preference order) can lead to an unclear design, and the interleaving of precedence order and preference order is quite confusing.
- The flex transaction model does not support nesting.

In the sequel of this chapter, we will develop a model for atomic processes that provides flexible structures while avoiding the drawbacks described above.
5. Atomic spheres

5.3. Definition: spheres of atomicity

Atomic units are introduced into OCR in the form of spheres of atomicity. The notion of a sphere as it is used here is inspired by Davies [Dav78], and the sphere concept can be seen as an approach to decompose the transaction concept by allowing to assign to a process only those transactional properties which are actually needed. To this end, it is possible to distinguish between spheres of atomicity, spheres of isolation, and spheres of persistence [AHST97]. In this chapter, we will concentrate on spheres of atomicity.

In OCR, an atomic sphere is a set of activities that is required to either execute successfully or leave no effects in the outside world. To achieve maximum flexibility, no strict atomicity is required, but a weakened notion similar to semi-atomicity as it is defined in the FLEX model. Spheres are created by marking activities, blocks, or processes as atomic. Hence, it is possible to make only parts of a process atomic, while other parts have weaker execution requirements. There are, however, restrictions on the allowed combinations of atomic and non-atomic elements. For instance, it is obvious that an atomic process is not allowed to contain non-atomic components. We will discuss these issues later in this chapter.

The interpretation of an atomicity declaration differs from activities to blocks. If a process designer declares an activity to be atomic, he assures to the system that the program associated with the activity leaves no side-effects if it is aborted. The atomicity of a block (or process) has to be guaranteed by the process support system.

5.3.1. A classification scheme for tasks

Similar to the distinction between compensatable, retrievable, and pivot tasks in the FLEX model, we introduce a task classification scheme that allows to specify the execution characteristics of programs, blocks, and processes. It is the basis for scheduling decisions of the runtime-system as well as for validation of correctness at compile-time. In this scheme, task execution characteristics are classified in three dimensions, as displayed in Fig. 5.4. The following dimensions and

Figure 5.4.: Classification of task execution characteristics

58
5.3. Definition: spheres of atomicity

characteristics exist:

- **Atomicity:**
  - *Atomic*: These tasks have no side effects if they fail. Transactions issued to a database or to a distributed environment through a TP monitor fall into this category. This class can also be used for activities that do not cause any changes, like read operations.
  - *Non-atomic*: Activities belonging to this class may have side effects if they are aborted. Obviously, the system cannot enforce atomicity if a process contains non-atomic activities.
  - *Quasi-atomic*: Quasi-atomic tasks do not perform automatic rollback in case of a runtime failure. They do, however, keep enough information to allow an undo after the failure has happened. Examples are certain CAD systems that perform logging during operation, or the SAP R/3 workflow component [Sch98]. After a failure, recovery can be initiated through an explicit call to the system. When an activity is declared to be quasi-atomic, a *rollback-method* has to be registered which can be invoked by the system to invoke the undo. Quasi-atomicity can also be declared for blocks or processes by specifying a rollback-method which is executed instead of compensating each single task.

- **Recoverability:**
  - *Compensatable*: Tasks within this category can be undone after their successful completion. The concept of compensation was already discussed in conjunction with transaction models. For this class, it is necessary to register a *compensating method* that will be called by the system when an activity has to be undone.
    
    If a block or process is declared to be compensatable, it has to be specified whether it is to be compensated by a compensating-method or through the compensation of its component tasks. (This distinction between integral and discrete compensation was proposed in [Ley95].) Note the difference between the rollback-method and the compensating-method for processes and blocks. Rollback applies when the sphere has to be aborted (i.e., did not complete successfully), while compensation is necessary if the sphere was successful and has later to be undone. It will often be necessary to have separate methods for these two tasks.
  - *Deferred Commit*: This category applies to activities which support a two-phase commit, i.e., can be rolled back after they have finished, but before they receive a final commit signal. The most common example for this class are database transactions which are controlled through an XA interface [Gro92]. Upon declaring this property, three methods have to be registered. The *prepare-method* and *commit-method* are used for the respective messages of the 2PC protocol. The *abort-method* initiates the undo of the activity.
  - *Non-compensatable*: A non-compensatable activity cannot be undone after it has finished successfully.

- **Termination:**
  - *Retriable*: If this sort of task fails, it can be executed again and will eventually succeed. The concept was adapted from the FLEX model. Examples are many programs that may fail due to operating system failures or communication problems, but will succeed
5. Atomic spheres

after a sufficient number of tries. Think of a word-processor running on a Windows
PC. To make this category more practical, the number of retries can be bound by a
positive integer. The system will then try to restart the execution only a limited number
of times and signal a failure if no success was reached afterwards.

- Guaranteed Success: This is a category that will usually not apply to activities, but to
blocks or processes. A process that consists only of retrievable activities is guaranteed
to succeed.
- Non-retriable: The standard case, in which a task failure can not be resolved by a
re-execution.

5.3.2. Well-formed composite spheres

Based on the declared properties for the basic steps, OPERA enforces atomicity of composite
spheres, i. e., processes and blocks. A process is guaranteed to be atomic if it either is declared to
be quasi-atomic (in this case, the process designer has to provide a backout method that performs
rollback), or if (a) every component task is atomic or quasi-atomic and (b) every component task
is compensatable or the process has a structure that conforms to the well-formedness rules for
flexible transactions. An important difference is that in the OPERA model, there is no preference
order. Hence, the rules for well-formedness can be simplified as follows.

Definition 1 (Basic notation) We use the following terms and abbreviations:

- A sphere $S$ is a tuple $(T, \rightarrow)$.
  - $T = t_1, t_2, \ldots, t_n$ is a set of tasks.
  - $\rightarrow \subset (T \times T)$ is the precedence relation defined on the tasks.

- Let $PROP$ be the set of task execution characteristics, i. e.,
  $PROP := \{atomic, quasiatomic, retrievable, guaranteed\_success, compensatable,$
  $commit\_deferred\}$.

- The function $prop : T \rightarrow 2^PROP$ assigns to each task a set of execution characteristics.
  As a shortcut, we introduce test predicates allowing to determine certain properties of tasks
directly: The predicate $atomic : T \rightarrow P$ is true if and only if the task to which it is applied
is atomic. Respective test predicates exist for all task execution characteristics.

- To simplify matters, the test predicate $comp(t)$ returns true if $t$ is either compensatable or
  allows a deferred commit. Likewise, the boolean predicate $retr(t)$ returns true if $t$ is either
  retrievable or guaranteed to succeed.

- $pivot(t)$ returns true if $t$ is a pivot.

Definition 2 (Well-formed atomic spheres) We can now give a definition for well-formed atomic
spheres:

1. A sphere $S$ is allowed to have at most one pivot task.
   $\forall t_1, t_2 \in T : pivot(t_1) \land pivot(t_2) \Rightarrow t_1 = t_2$

2. Every possible execution path to this pivot $t_p$ must consist solely of compensatable tasks.
   $\forall t \in T : (t \rightarrow t_p) \Rightarrow comp(t)$
3. Every possible execution path from a pivot or retriable task must consist solely of tasks which are retriable or guaranteed to commit.
\[ \forall t_1, t_2 \in T : (t_1 \rightarrow t_2 \land (\text{pivot}(t_1) \lor \text{retr}(t_1))) \Rightarrow \text{retr}(t_2) \]

4. Parallel tasks must have the same type.
\[ \forall t_1, t_2 \in T : (\neg (t_1 \rightarrow t_2 \land t_2 \rightarrow t_1)) \Rightarrow (\text{comp}(t_1) \land \text{comp}(t_2)) \lor (\text{retr}(t_1) \land \text{retr}(t_2)) \]

Note that these rules are much more restrictive than the usual FLEX rules, mainly because no alternative executions can be specified. In OPERA, contingency executions and other recovery strategies are implemented through the exception handling mechanism which will be introduced in section 6.

5.3.3. The rollback algorithm

Aborting a sphere has to take into consideration the task execution characteristics of the sphere as well as of its component tasks. Given that the sphere is well-formed, the following algorithm can be used:

1. **Stop execution**: Abort all tasks that are currently running. Depending on the execution characteristics of these tasks, it is necessary to invoke their rollback-method (for quasi-atomic tasks).

2. **Rollback**: We distinguish between the atomic and the quasi-atomic case. If the sphere is atomic, perform single-step backout: Compensate each task, thereby using the reverse order of the original execution\(^1\).

   If the sphere is quasi-atomic, invoke its rollback-method. The component tasks are not compensated. This strategy can speed up the recovery significantly and can be applied in situations in which no work needs to be undone but some notification of the abort has to be written to a log or sent to an administrator. We note again that the holistic backout should not be confused with holistic rollback, which is performed for compensatable composite tasks for which a compensating method was defined.

   Note that for flex structured processes, an atomic abort is only possible while only compensatable activities have been executed. Once a pivot or repeatable activity has succeeded, these processes become quasi-atomic in the sense that they can only be aborted through a backout method. The process designer has the possibility to determine the behavior of flex structured processes in specifying whether holistic or single-step backout is preferred when there are only completed compensatable tasks.

5.3.4. Extended task state model

With the new task execution characteristics taken into consideration, the task state diagram changes into the one shown in Fig. 5.5. Here, the gray states are the ones already existing in the simple state diagram of Fig. 4.4. The new states, which have white background, are needed to capture the new execution semantics. The state transitions which are drawn as dotted lines apply only to specific types of tasks. The following changes are important:

\(^1\)Note that it is possible to compensate all steps in a sphere in parallel if it is known that there are no conflicts between the compensating activities. However, we have no knowledge of possible conflicts. The separate rollback of steps in reverse order is always safe.
5. Atomic spheres

Figure 5.5.: Task state diagram for atomic tasks

- After the failure of a quasi-atomic activity, its rollback is initiated, which finally leads to the Undone state. If an atomic activity fails, it changes into the Undone state directly.

- Retriable activities re-enter the ready state, either from the Failed state (if they are non-atomic) or from the Undone state (if they are atomic or quasi-atomic).

- A compensating task changes into the Compensating state while its compensation method is executing. After the successful undo, it changes into the compensated state.

5.4. Discussion

Modeling a process as a nesting of atomic spheres gives the process designer the possibility to limit the amount of rollback in the case of a failure. In this, our model is similar to the traditional nested transaction model. Extended, however, by incorporating the idea of different task types, which was taken from the flex transaction model. In addition, the range of available types was extended in order to better reflect actual application semantics. The introduction of holistic backout and holistic compensation is another contribution towards capturing application semantics. It can speed up recovery times significantly.

The model described so far is limited in the possible recovery strategies. It is only possible to perform (partial) backward recovery through the compensation or abort of spheres. A mechanism for the specification of alternative execution paths is missing, since we dropped the idea of the preference order which was present in the flexible transaction model. In the next chapter, such a mechanism will be introduced.
6. Exception handling

While atomic spheres provide a basic degree of fault tolerance, they need complementing mechanisms for specifying how to recover from a failure. This problem is not particular to PSS, it also exists, for instance, in database applications. The transaction programmer can rely on the atomicity as a basic abstraction guaranteeing database consistency, but she has to program the necessary steps to maintain consistency on the application side. This is usually done by detecting transaction aborts and executing appropriate recovery operations, like the re-execution of the failed transaction, the execution of alternative operations, or the abort of the application. All these strategies have to be implemented in the host programming language since the transaction concept itself provides no support for the direct specification of recovery strategies. In this section, we opt for an integrated recovery mechanism that combines atomicity with a powerful mechanism for the specification of failure handling. This mechanism is based on the well-known and approved principle of structured exception handling.

This chapter is structured as follows. In section 6.1, we review the concept and its history. In section 6.2, we integrate exception handling mechanisms into process modeling languages, and in section 6.3 the problem of exception detection is discussed. Section 6.4 contains an example illustrating the concepts.

6.1. Structured Exception Handling in Programming languages

The idea of structured exception handling dates back to early programming language research. In conjunction with the development of principles like modular programming, it was noticed that the benefits of structured program design are useful only if failures are taken into consideration. This requires concepts that allow to detect and handle failures in a structured way. A general outline of such a system was first given by Parnas [Par72], who proposed the separation of failure detection and failure handling and presented a basic concept in which it would be possible to pass failure notifications between different levels of a layered system design.

This concept was later refined by Goodenough [Goo75], who described a detailed concept for exception handling in higher programming languages. This concept is based on the following principles:

- Exceptions are typed signals reporting an error in an invoked operation to the invoker.
- The invoker of an operation associates with the call one or more exception handlers which are automatically called when the operation signals an exception. An exception handler has to be declared for each exception type that may be signaled.
- Exception handlers can be attached not only to single operation invocations, but to whole blocks of operations, which simplifies the specification significantly.
- An exception handler can decide how the control flow continues after its operation. It can terminate the operation, in which case control is returned to the caller. It can resume the
failed operation’s execution, which can then normally finish. Or it can pass the exception to the next higher level in the invocation hierarchy, in which case an exception is signaled to the “invoker of the invoker”.

- Exceptions can enforce a specific behavior of their handlers. To this end, three types of exceptions are defined. An escape exception requires termination of the operation raising the exception. A notify exception forbids termination and requires resumption of the operation’s execution. A signal exception allows either resumption or abort.

- Explicit declarations of exception-related aspects, like the exceptions a procedure may signal, the exception a handler reacts to, or the type of an exception (escape, notify, signal) are essential. They allow to implement compile-time checks which reduce the probability or runtime errors.

This concept provides well-defined language primitives that enable the separation of exception detection and exception handling. Furthermore, a wide range of strategies can be implemented which allow the treatment of various types of exceptions. The mechanisms have been adopted in many programming languages, including CommonLisp [Ste90], Standard ML [Pau91], C++ [Str91], and Java [Fia96]. They are also an integral part of communication standards for distributed systems like CORBA [Obj92], and exception handling has even been integrated into Windows NT [Cus93], where it is used to handle system and user-defined exceptions in a uniform way.

The replacement model described by Yemini and Berry [YB85] provides sound semantics for the exception handling mechanism. It has its name from the principle that a failed operation is logically replaced by its exception handler. The code below is a modified example from [YB85]:

```c
char* strtoupper (char *str) throws Badcode;
{
    char result[1000];
    int i = 0;
    while (str[i] != '\0' && i < 1000)
    {
        if (isprint (str[i]))
            result[i] = toupper (str[i]);
        else
            result[i] = (throw Badcode (str[i]));
    }
    return result;
}
```

The function, written in C++ with small extensions which are described below, converts the characters of a string into uppercase letters. If a character has no valid type (i.e., is not a printable character as defined by the C isprint function), an exception of type Badcode is signaled, which receives the incriminated code as a parameter. We assume here that the exception handling mechanism allows for resumption (which is normally not possible in C++). Hence, it is possible to use the result of the throw directive. This allows for the passing of parameters to the exception signaler if it is resumed. Another assumption we make here is that the scope of exception handling is a function (in contrast to the try blocks normally used in C++): an abort due to an exception aborts the function in which the exception was raised.

An exception handler for this example can either resume the execution of the function (by supplying, for instance, the character '?' as a replacement for the non-printing character), or it
can abort the whole function, in which case it has to provide a proper result that conforms to its declared return type. In the example, a string has to be returned which could indicate that an error was encountered ("Invalid input" or something similar). Note the replacement semantics in both cases. In the case of a resume, the `throw` directive is replaced and a proper return value is provided which allows to continue execution. In the case of an abort, the call to `strtoupper` is replaced, and a proper return value is provided that allows to continue operation in the calling procedure. It is important to note that an exception handler has to declare two sets of return values, one for the resumption case and one for the abort case.

The replacement semantics allow to embed exception handling directives seamlessly into the structure of a program, without the need for explicit `try` blocks or other syntactical constructs. The mechanism is especially useful for our PSS fault tolerance mechanism. Recall that we were looking for means to combine atomic spheres with forward recovery facilities. With an exception handling mechanism, forward recovery can be specified through exception handlers, while backward recovery is implemented by aborting spheres. The mechanism is seamless in the sense that the return value of a sphere is always defined, since it is either delivered by the normal code or by an exception handler replacing it. In the next sections, we will describe the OPERA exception mechanism in detail.

### 6.2. Adding exceptions to process modeling languages

Applying exception handling ideas to OPERA requires to integrate appropriate language features into OCR and to define the semantics of exception handling if used in conjunction with the atomic sphere concept. We will start by describing the language integration. To show that the concept is feasible not only for the canonical representation, but also for external modeling languages, and to give intuitive examples, we show also how the constructs could be integrated into a graphical language like the FlowMark model described in section 4.4.

#### 6.2.1. Exceptions

We introduce exceptions as new entities into the canonical representation. An exception is a quadruple \((N, T, I, O)\), where \(N\) is a unique name, \(T\) is the exception type (notify, escape, signal), \(I\) is an input parameter list, and \(O\) is an output parameter list.

Two classes of exceptions have to be distinguished: **Synchronous exceptions** are signaled by explicit language primitives (similar to the `throw` primitive in C++), and the programmer controls the time of exception detection by placing them in the control flow. **Asynchronous exceptions** are predicate-based and allow to detect failures based on the process state, which allows, for instance, to detect deadline violations or invalid values of global variables.

The same data flow mechanism used for normal activities is used to handle the data flow during exception handling. Since an exception has data containers, when the exception occurs, its input container is used to pass information to the handler. Similarly, the handler has the possibility to return data to the signaler using the output container of the exception.

Asynchronous exceptions are similar to synchronous exceptions except for the fact that they do not take part on the normal control flow. Instead, they have an associated exception condition instead of the guard, which can be an arbitrary predicate over the data objects in a process or block. Conceptually, they could be seen as activities to which all other activities are connected through a control connector that gets activated when the exception occurs. The advantage is that now this control flow towards the exception is implicit and the workflow designer does not need to construct it explicitly. Note that in current systems the only way to achieve similar functionality is
6. Exception handling

to actually treat the exception as an activity and add control connectors between all activities that could possibly raise the exception and the exception activity. The same would need to be done with the data connectors. Many actual implementations actually resort to this very inelegant, very inefficient solution to be able to provide a minimal failure handling capability.

Activities and processes have to declare the exceptions that they will possibly raise. Exceptions become thus an integral part of a task's interface. It has been argued in [Goo75] that this helps to avoid programming errors, like the omission of handlers for certain exceptions. With the exception specification, it is possible for the workflow system to verify a correct embedding of tasks. This puts an additional burden on the workflow designer, but it simplifies the integration of existing definitions into new workflows and is thus essential for reusability. Note that in a workflow development environment, declaration of exceptions can be supported by the system, which can automatically scan a process or activity specification and generate the declaration of exceptions.

6.2.2. Exception handlers

Exception handlers are modeled as specialized blocks in OCR. In addition to the normal block construct as it was described in chapter 4, they are enhanced with an exception declaration and a second parameter list. The exception declaration specifies which exception can be handled by the handler, and the second parameter list is used to pass information back to the signaler if its operation is resumed. The format of the parameter lists is defined by the context in which the handler is embedded. The input parameter list must conform to the input parameter list of the exception it is handling. The output parameter list must conform to the output parameter list of the syntactical unit (activity or sphere) it is attached to, and the resumption parameter list must conform to the format of the exception's parameter list.

Each possible exception a task may signal has a corresponding default handler, which is either system-provided or defined when the task was registered. The system default handler matches every exception without specified handler, aborts the signaler and then propagates an exception to the caller. A process designer can, however, provide user-level default handlers where this is appropriate. For each task integrated into a process, the designer can provide override handlers for those exceptions where the default behavior needs to be modified. The advantage of this approach is that it facilitates modular design: Reusing components becomes easier since they will either cope with any possible exception themselves or will pass the exception up to the caller. This is a significant advantage over existing systems in which exception handling is entirely hard-wired and needs to be modified every time a process is used in a different context. By combining default and override handlers, the designer can let the system take care of exceptions and specialize the behavior when necessary.

An example for the graphical representation of an exception handler is given in figure 6.1. The entry point to a handler process is always a so-called proxy activity that can be seen as a placeholder for the exception that occurred. The output container of the proxy contains the data that have been passed by the signaler together with the exception. This makes the case-dependent data accessible inside the handler. In our example, two predicates, $P_1$ and $P_2$, are defined on these data. This allows to take different execution paths depending on the information provided by the signaler. Terminator proxies define the endpoints of a handler. They determine how the control flow has to proceed after termination of the handler. Different types of terminators can be used depending on whether the signaler has to be aborted or resumed and whether an exception is to be propagated.

In addition to the functionality provided in ordinary processes, OPERA provides special con-
6.3. Detecting exceptions

Figure 6.1.: An exception handler

Structs that can only be used in exception handlers. They are syntactical shortcuts that facilitate the convenient specification of recovery-related tasks:

- Retry: If recovery requires to retry the execution of the task that caused an exception, retry proxies are introduced. They refer to the task that raised the exception currently handled and can be marked with a time interval to specify a delay before the re-execution is to be scheduled. Since this mechanism could lead to an indefinite number of recursive invocations, the following semantics are defined for the retry mechanism: If during the retrial of T the same exception is raised again, the system does not call the exception handler again. Instead, the control flow returns to the first invocation of the exception handler. Note that repeated invocations of T are still possible with this rule, but need to be explicitly specified in the exception handler.

- Human interaction: Due to the complexity of workflow processes and the rules that determine their behavior, it may not be possible to determine the strategy to follow with exceptions at the time the process is defined. In these cases, human intervention is necessary. We provide a special notificator proxy that allows to transfer control to a user responsible for dealing with the exception. Notificators have input and output containers that allow to pass information concerning the failure to the users and are used to send information back to the handler once the failure has been resolved.

6.2.3. Example

Figure 6.2 shows several examples for the flow of control in OPERA during exception handling, depending on the decision of the handler. In diagram (a), the exception handler resumes execution of the signaler. In diagram (b), the signaler is aborted and control returns to the process that invoked it. Diagram (c) shows a two-level nested execution, where the innermost process (p2) raises an exception which is propagated by the exception handler, enforcing the abort of p2 and the invocation of an exception handler associated with p1. This handler resumes the operation of p1.

6.3. Detecting exceptions

In addition to the problem of modeling exception handling strategies in a comprehensible way, PSS have to capture failure signals returned by external applications. There are several ways in which these exceptions are signaled. In the simplest case, explicit exceptions are used, as
6. Exception handling

![Figure 6.2: Control flow during exception handling](image)

illustrated in Figure 6.3 which shows an interface description in CORBA’s interface definition language (IDL). CORBA services communicate failures to their callers through exceptions that are returned instead of the defined return values. In our example, two potential exceptions are declared: invalidData and noSeats. Another approach for communicating failures is the more traditional one of using return codes, usually some integer representing the failure. In the example, the record keeping activity (implemented in Encina [Tra95, BN97], for example) signals failures this way. A third important signaling method is the use of explicit failure channels like they are used in persistent queuing systems or queue-oriented TP monitors like Tuxedo [Sys96, BN97]. The process management system must translate these heterogeneous failure signals into an internal format that allows to treat them uniformly.

If programs that are part of a workflow do not use OPERA’s exception mechanism, but rely on proprietary ways of exception signaling, OPERA translates these external exceptions into its internal format at run-time based on mapping information provided when the programs are registered. Program registration ensures that the engine has the necessary information to invoke a program or notify a user, like IP addresses and command lines in the case of programs or role specifications and descriptive text for human activities. In OPERA, exceptions also have to be declared. This includes the declaration of an exception translation function defining which external signals result
interface flightSystem {
    exception invalidData {string reason;};
    exception noSeats {string reason;};
    int checkAvailability (in date depDate, in time depTime,
        in string depCity, in string destCity,
        out string category, out string flightNumber,
        out int noOfSeats)
        raises (invalidData);
    int bookFlight (in string flightNumber, in string category,
        in string number, out string code)
        raises (invalidData,noSeats);
};

Figure 6.3.: Sample IDL definition including exception declarations for flight reservation tasks in the example process

in which internal exceptions. The format in which this function is given is dependent on the type of the external application:

- **Workflow-aware applications** [SJHB96] are applications that have been specifically designed to be used with OPERA by incorporating calls to the OPERA application programming interface (API). The OPERA API is a library of procedure calls which contains calls to directly signal OPERA exceptions. Hence, there is no need for the translation of exceptions in workflow-aware applications.

- **Legacy applications** are programs that are not aware of the workflow system. Being stand-alone applications invoked through the operating system, they signal failures through special return codes that are converted to OPERA exceptions by the runtime system. The conversion is based on a *translation table* registered with the program that maps specific return values to appropriate exception types.

- **Standard environments**. Distributed environments like CORBA [Obj92] provide their own exception mechanisms. Exceptions returned by their applications are directly converted into OPERA exceptions. The exception declarations for a service are parsed from its IDL file that has to be provided when the service is registered. Note that these application do usually not allow signaler resumption, i.e., programs are always aborted when they send an exception.

- **Manual activities**. Humans communicate with the PSS through worklists, which are graphical user interfaces. The signaling of exceptions by human agents is supported through the worklist by a suitable GUI. In the failure case, it will return an appropriate exception signal. The user may be presented a list of possible exceptions, from which she can select the appropriate one.

The exception mechanism is also used to detect and signal semantic failures. OPERA provides two options: synchronous exception raising, based on special *signal proxies* embedded into the control flow description, and asynchronous exception raising, which is based on predicates over process-internal data.

*Signal proxies* can occur anywhere inside a process. A signal proxy is associated with an exception name, data containers, and an exception category. If the flow of control in a process reaches a signal proxy, control is passed to the appropriate exception handler. Figure 6.4 gives an example of explicit signalisation. If after the execution of Activity1, the value of the parameter R
is negative, an exception is raised. Since the exception type (cf. Section 4) is *Escape*, this leads to the termination of the process.

![Figure 6.4.: Explicit signaling of an exception](Image)

For implicit signaling, the workflow designer has to provide a set of predicates that define under which circumstances a given exception must be raised. The variables for these predicates are those in the containers or in the whiteboard associated with each process. The designer can for instance define *startup predicates* and *termination predicates*. The former evaluate the parameters passed to the process upon startup. They allow to check if these values allow the process to execute correctly. If incorrect values are encountered, a user can be informed through a *Notify* exception. This user can then correct the parameters and resume the process execution. Termination predicates check the return values of the process and raise exceptions if incorrect values are detected.

### 6.4. Discussion

![Figure 6.5.: The travel example modeled with the new primitives and the control flow when handling a failure of activity A2](Image)

To illustrate the approach, a specification of the introductory example (Figure 5.2) using the new primitives is given in Figures 6.5 and 6.6. The left hand side of Figure 6.5 shows the graphical representation, the right hand side displays the control flow for the case that the car rental activity fails, and figure 6.6 shows the textual representation in OCR. (Remember that the OCR representation is compiled from the graphical specification, hence a process designer does not write OCR code but uses a graphical design tool that hides much of the OCR syntax.)
6.4. Discussion

Program registrations (user-provided):

PROGRAM BookFlight
    INPUT: DepTime, DepCity, ArrCity, Category : String;
    OUTPUT: BookingCode: String;
    PROPERTIES: COMPENSATABLE;
    COMPENSATING: CancelFlight (BookingCode);
    EXCEPTIONS: NO_SEAT, NO_SEAT_IN_CATEGORY;
    SUBSYSTEM: ...
    INVOCATION: ...

PROGRAM BookHotel
    INPUT: City, Hotel, Begin, End: String
    OUTPUT: BookingCode: String;
    PROPERTIES: COMPENSATABLE;
    COMPENSATING: CancelHotel (BookingCode);
    EXCEPTIONS: NO_SEAT
    SUBSYSTEM: ...
    INVOCATION: ...

PROGRAM ReserveTrain
    INPUT: DepTime, DepCity, ArrCity: String;
    OUTPUT: BookingCode: String;
    PROPERTIES: COMPENSATABLE;
    COMPENSATING: CancelTrain (BookingCode);
    EXCEPTIONS: NO_SEAT;
    SUBSYSTEM: ...
    INVOCATION: ...

PROGRAM RentCar
    INPUT: City, Begin, End: String
    OUTPUT: BookingCode: String;
    PROPERTIES: COMPENSATABLE;
    COMPENSATING: CancelCar (BookingCode);
    EXCEPTIONS: NO_CAR;
    SUBSYSTEM: ...
    INVOCATION: ...

EXCEPTION NO_ROOM
    INPUT: City, Begin, End: String;
    OUTPUT: Void

EXCEPTION HANDLER EH1
    INPUT: Begin, End, City: String;
    COMPONENTS:
    TF: PROXYTASK_FAILED;
    N: NOTIFY(INITIATOR), ACTIVATOR: TF;
    OUTPUT: Continue: Boolean;
    T1: TERMINATORABORT, ACTIVATOR: finished(N), CONDITION: (TF.Continue = TRUE);
    T2: TERMINATOR RESUME, ACTIVATOR: finished(N);
    CONDITION (TF.Continue = TRUE);

EXCEPTION HANDLER EH2
    INPUT: Begin, End, Destination: String;
    COMPONENTS:
    TF: PROXY TASK FAILED;
    A: ACTIVITY ReserveTrain (Destination, Time), ACTIVATOR: TF;

EXCEPTION HANDLER EH3
    COMPONENTS:
    NR: PROXY NO_ROOM;
    A5: ACTIVITY BookHotel (NR.City,'Central', NR.Begin, NR.End), ACTIVATOR: NR;

Process representation (Compiled from graphical specification [Fig. 10]):

PROCESSTravel
    INPUT: Begin, End, City: String;
    OUTPUT: Success: Boolean;
    BACKOUT: SINGLE;
    COMPONENTS:
    Transport: BLOCK;
    A3: ACTIVITY BookHotel (City, 'Hilton', Begin, End), ACTIVATOR: finished(Transport), HANDLERS: EH3 HANDLES NO_ROOM;
    HANDLERS: EH1(Begin, End, City) HANDLES TASK_FAILED;

BLOCK Transport
    PROPERTIES: ATOMIC;
    BACKOUT: SINGLE;
    COMPONENTS:
    A1: ACTIVITY BookFlight (Begin, 'Zurich', City, 'Any');
    A2: ACTIVITY RentCar (City, Begin, End), ACTIVATOR: finished (A1);
    HANDLERS: EH2(Begin, End, City) HANDLES TASK_FAILED;

Figure 6.6.: OCR representation of example workflow
The process description has been decomposed into a process (*Travel*), shown in the center of the graphical process representation, and three exception handlers. Note that the process itself contains only the business logic, plus a sphere (*Transport*) that indicates that flight booking and car renting are regarded as atomic with respect to failures. The graphical representation shows the elegance of the proposed approach, especially if compared with the process in Figure 5.2, which is the only possible way to cope with exceptions in most current systems. Furthermore, modifying the process becomes straightforward. Consider the addition of another task in the booking process (e.g., reserving theater tickets). While in the conventional design this would require embedding several new nodes and arcs to the graph to avoid violating the failure semantics, in OPERA only one new task has to be added to the process description, since all recovery-related steps are taken care of by the system. Thus the OPERA approach to exception handling guarantees reusability of process descriptions, since existing specifications can easily be used as a basis for new processes. Moreover, reusability of tasks is improved, since the exception mechanism can be seen as a form of parameterization of activities and processes allowing to use a once-declared task in a large number of contexts. Consider the BookFlight program as it is shown in Figure 6.6. It has the required seat category as a parameter and raises the exception NO\_SEAT\_IN\_CATEGORY if the flight has free seats, but none in the requested category. Assume that the default exception handler for this exception aborts the program and propagates an exception. While this is the appropriate behavior in many cases, it is possible to change it by providing an override exception handler that resumes the execution of the program, allowing it to reserve a seat in another category than originally requested. Hence the criteria of reusability and flexibility are met by the approach.

The NO\_ROOM exception is registered with the system, which ensures that its parameters are known to all processes and exception handlers. Note that the exception TASK\_FAILED needs not to be declared, since it is a system default exception. All activities have associated default exception handlers that propagate this exception. The sphere has an associated override handler (*EH2*) that catches the propagated exception. If either A1 or A2 fails, this handler gains control and calls the train reservation task (*A3*) while the sphere is aborted, and the backout mechanism cancels reservations already made (the sphere’s backout mode is declared as Single Step). Should the train reservation fail as well, its exception is propagated automatically to the process itself (remember that we do not allow recursive handler calls). This activates the process’s handler, which notifies the process’s invocator. This person has the possibility to either abort the whole process or to perform appropriate actions (for example, organize a travel by private car) and resume execution. If the execution is resumed, the process continues with the hotel reservation. This activity has an associated exception handler (*EH3*) that invokes the reservation of another hotel if no rooms are available. If this activity fails, too, an exception is propagated to the process and its handler gets control again, informing the invoker of the process.

As an illustration of the forward and backward navigation performed by the system if a failure occurs, the right hand side of Figure 6.5 shows the control flow if activity A2 (RentCar) fails. First, the default handler for A2 is invoked, which propagates the standard exception TASK\_FAILED to the next higher level, which in this case is the sphere S. This leads to the invocation of *EH2*, an exception handler associated with the *Transport* sphere, which calls activity A4 (ReserveTrain) in order to handle the exception. After the completion of A4 the sphere is aborted (because of the single step backout method the system automatically calls the compensating operation for A1, canceling the flight), and operation continues with P’s next regular operation A3. This example shows how, based on the failure semantics specified through spheres, exceptions and exception handlers, flexible recovery is enforced.

In the OCR representation in Figure 6.6, all exception handling related parts have been marked through boldface typing. The program declarations in the top region contain the declaration of
their transactional properties, a reference to a compensating process (if it exists), and a declaration of the exceptions they may raise. Exceptions themselves and their exception handlers are declared in the middle region. Note that the structure of an exception handler is similar to the structure of an ordinary process (e.g., the Travel process in the bottom region), with the difference that the handler contains a proxy activity that represents the exception to be handled. Due to space limitations, the descriptions of the compensating tasks (they are programs in this example, but they could be processes of arbitrary complexity) as well as the specifications of subsystems and external references are not included in the figure. It is important to note that most of the OCR code is generated automatically from the graphical process description. The process designer must only provide information about the external programs (shown in the upper part of the figure), like the program file to execute and the parameters to pass to and returned by the program.
6. Exception handling
7. Validating process specifications

Atomicity and exception handling mechanisms give process designers the ability to specify fault tolerant processes by deliberately creating spheres as units of joint compensation and defining appropriate failure recovery strategies through exception handlers. The combination of the mechanisms, however, complicates the detection of failures in process specifications. The rules for well-formed processes given in section 5.3.2 are not sufficient, since new violations of atomicity can be introduced through inappropriate use of exception handlers. Hence, what is needed is a new correctness criterion and a validation mechanism that aids process designers by checking process specifications, detecting failures, and proposing changes that lead to correct processes. The aim of this chapter is to define the notion of a well-formed process specification and to describe an algorithm for correctness checking. In doing so, we use the well-formedness criterion for atomic spheres (section 5.3.2) as a starting point. It will, however, be extended in order to take the exception handling functionality into account.

The chapter is structured as follows. Section 7.1 contains a motivation. Section 7.2 gives necessary notation. Section 7.3 presents a correctness criterion, which is used in the validation algorithm described in section 7.4. Section 7.5 evaluates the complexity of the algorithm, and section 7.6 discusses a number of possible variations. Section 7.7 contains a discussion of all fault-tolerance solutions presented in this and the previous two chapters.

7.1. Motivation

The well-formedness definition of section 5.3.2 is valid only as long as a process contains no exception handlers. If handlers exist, it is possible that the execution characteristics of a task are changed through the execution of an handler. Recall that, due to the replacement model semantics, it is possible that a compensatable task is replaced by a non-compensatable exception handler, which may lead to an incorrect process structure. The following examples will serve as a starting point for the further investigation of these problems.

Consider the four examples for process structures in Fig. 7.1. The process of example (A) has a correct ordering of tasks if the exception handlers are not taken into consideration. The handlers, however, have inappropriate execution characteristics, leading to non-atomicity if one them is executed. The non-compensatable handler attached to a compensatable task, for instance, leads to unrecoverable effects, because it cannot be undone if the pivot task should fail. The same effect occurs for the retriable task at the end of process (A). It is replaced by a non-retriable handler, which again can lead to non-atomicity if the exception handler fails.

Example (B) shows the reverse effect, i.e., the correction of an ill-structured process through the introduction of exception handlers. The compensatable task at the end of the sphere has an exception handler which is retriable. Hence, it is guaranteed that the sphere completes after a successful termination of the pivot, even if the compensatable activity should fail. Note that the reverse case is not possible: A retriable activity before a pivot is not acceptable, even if it has a compensatable exception handler.
7. Validating process specifications

Another important observation is that all exception handlers attached to a task have to be retriable in order to "repair" a non-retriable task. Hence, a process case like that in example (C) is not correct, since, if the third task fails with exception \( E_2 \), the non-retriable handler is selected. If this handler fails, the process is blocked and no rollback is possible.

Example (D) shows incorrect behavior due to the fact that a sphere is aborted at the wrong time. The third task has an exception handler which propagates the exception up to the next higher level in the invocation hierarchy. Here, it is caught by a higher-level exception handler. In the case that this handler decides to abort the sphere, atomicity is violated since the pivot activity has already been executed. This is an important example since it shows that the correctness of the exception handling mechanism has to take into account the state of a sphere at the time an exception handler is invoked, and a sequence of exception handler invocations over several levels.

7.2. Notation

Our goal is now to derive a well-formedness criterion for atomic spheres enriched with exception handling elements. We start by introducing the necessary notation and will then present conditions that have to hold for correct process definitions.

Definition 3 (New notion of spheres) We start by giving a modified version of Definition 1 (Section 5.3.2). The main difference is the inclusion of exception-handling related aspects.
A sphere $S$ is a quadruple $(T, H, <, e)$. $T$ is the set of component tasks, which may be activities, blocks, or subprocesses. $H$ is the set of exception handlers used in the block. $< \subseteq T \times T$ is the control flow relation. It is derived from the activator part of the guards attached to each task). $e \subseteq ((T \cup H) \times E \times T)$ is the exception handling relation, where $E$ denotes the domain of all exception types. Hence, the exception handling relation relates tasks to exception handlers based on the exceptions that may occur. Note that this relation can be determined based on the declarations which have to be provided by the process designers. Note also that exception handlers can raise exceptions as well. Hence, they are also included in the relation.

As a shortcut, we will denote with $T^+$ the union of tasks and exception handlers, $T \cup H$.

The function $\text{prop}$, mentioned in Definition 1, is extended to work on $T^+$ instead of $T$. Likewise do the test predicates, i.e., atomic has now $T^+$ as its input set.

**Definition 4** (Transitive task execution characteristics)
The examples (A), (B), and (C) above show that the execution characteristics of a task can change if an exception handler is executed. To capture this fact, we introduce a new function $\text{prop}^* : T^+ \rightarrow 2^{\text{PROP}}$ and a new set of test predicates $\text{compensatable}^*$, $\text{retriable}^*$, etc, which are based on $\text{prop}^*$.

$\text{prop}^*$ is defined recursively:

1. If $t$ has no associated exception handler, then $\text{prop}^*(t) := \text{prop}(t)$

2. If $t$ has a set of associated exception handlers $H_t$, then
   (a) $(\forall h \in H_t : retr^*(h)) \Rightarrow (\text{prop}^*(t) := \text{prop}(t) \cup \{\text{retriable}\})$
   i.e., if all exception handlers of a task are retriable, the task can be considered being retriable.

   (b) $(\exists h \in H_t : \neg \text{comp}^*(H)) \Rightarrow (\text{prop}^*(t) := \text{prop}(t) \setminus \{\text{compensatable}\})$
   i.e., if at least one exception handler of a task is non-compensatable, the task has to be considered non-compensatable.

**7.3. Correctness**

The correctness criterion for fault-tolerant processes is enclosed in the notion of well-formedness given below.

**Well-formed process specifications:**

A well-formed process specification has to suffice the following three requirements:

1. **Atomicity of components.** The first necessary condition for block atomicity is that either all component tasks have to be atomic or quasi-atomic, or the block has to be quasi-atomic. There is one exception to this rule. If a task is retriable, is is allowed to be non-atomic. We assume that the re-execution of a retriable activity after a failure cleans up possible effects of the previous erroneous execution.
2. **Flex structure under replacement.** As a second necessary condition, we postulate that a sphere must adhere to the well-formedness definition of section 5.3.2, with the difference that well-formedness must hold for the transitive task execution characteristics as they were defined in section 4. This correctness requirement captures cases such as the ones described in the examples (A), (B), and (C) of Fig. 7.1.

3. **No abort after critical point.** A sphere that contains non-compensatable activities can be rolled back only until a certain point in its execution is reached. This point, which is defined by the successful termination of the first non-compensatable activity, is called a critical point [Can98]. The specification of a sphere can have multiple critical points due to parallel branches.

It has to be avoided that the sphere is aborted after the critical point, since rollback is not possible anymore. Obviously, there are two cases in which an abort of a sphere is possible: (1) The sphere contains a signal proxy raising an exception, and handling this exception leads to an abort of the sphere. (2) The sphere contains a regular task which can signal an exception, the handling of this exception leads to its propagation, and the resulting handling of the exception on the next higher level leads to an abort of the sphere.

Hence, the atomicity of a sphere can only be guaranteed if after a critical point no exceptions can be raised that could ultimately lead to an abort of the sphere. An algorithm for testing this condition will be given in the next section.

**Deriving execution characteristics of complex tasks:**

<table>
<thead>
<tr>
<th>Execution characteristic</th>
<th>Definition</th>
<th>Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic</td>
<td>Well-formed structure (see section 7.3)</td>
<td></td>
</tr>
<tr>
<td>quasi-atomic</td>
<td>Rollback-method provided</td>
<td></td>
</tr>
<tr>
<td>compensatable</td>
<td>Compensating method provided</td>
<td>All components compensatable</td>
</tr>
<tr>
<td>commit_deferrable</td>
<td>All components commit_deferrable</td>
<td></td>
</tr>
<tr>
<td>retriable</td>
<td>All components retriable</td>
<td></td>
</tr>
<tr>
<td>guaranteed success</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1.: Deriving execution characteristics of complex tasks

If a sphere contains blocks or subprocesses, it is necessary to compute the task execution characteristics of these complex tasks in order to assess the correctness of the sphere. The characteristics of a block are based either on the characteristics of its component tasks or on specifications given by the process designers. Table 7.1 defines for each execution characteristic the necessary conditions which have to hold. A process is compensatable, for instance, if either all of its components are compensatable, or if a compensating method has been provided which allows to compensate the whole process without regard to its components. Quasi-atomicity, on the other hand, can only be achieved through definition, by specifying a rollback method. A property that is only based on structure is the deferrable commit: Only if all components of a block are commit-deferrable, the commit of the block can be deferred.
7.4. A validation algorithm for well-formed atomic spheres

We will now describe a validation algorithm for atomic spheres that tests for the three necessary conditions described above.

7.4.1. Preliminaries

Given that we want to validate a sphere $S$, the algorithm assumes the existence of the following information:

- The set of component tasks $T_S$. We assume that each task $t$ is represented by a tuple $(E_t, C_t)$, where $E_t$ is the set of exceptions it can possibly raise, and $C_t$ is the set of task execution characteristics of the task which have either been declared or are (for complex tasks) derived according to section 7.3. We assume further that the set of task contains as a subset the set of signal proxies. The execution characteristics of a signal proxy are always “atomic, compensatable, guaranteed_success”, given that a signal proxy has no side effects, but is rather a syntactic element used to model exception signaling.

- The set of exception handlers $H$. Each exception handler $h$ is represented by a tuple $(C_h, T_h)$, where $C_h$ is the set of task execution characteristics and $T_h$ is the set of possible terminations (out of resume, abort, and propagate) of this handler. This information can be derived from the specifications of the exception handlers. We denote with $T^+$ the union of exception handlers and tasks.

- The control flow relation $<_S \subseteq (T_S \times T_S)$ which describes the relative order of the component tasks. It can be derived from the activators which are part of the process specifications.

- The internal exception handling relation for the component tasks, $e_i \subseteq (T^+ \times E \times H)$. It defines which exception handler has to be invoked if a particular exception is signaled by a particular task. Note that this relation does not cover exceptions raised by signal proxies, since those exceptions are not handled internally, but have to be treated by exception handlers of the sphere.

- The external exception handling relation of the sphere. It declares the exceptions the sphere may signal (if a signal proxy is executed or if an exception handler propagates its exception) and the exception handlers to be invoked. It is represented as $e_e \subseteq E \times H$.

- To simplify matters, we will omit indices whenever no ambiguities are possible because of the context. For instance, instead of $<_S$, we will often use $<$ to denote the control flow relation if it is clear which sphere is meant.

![Figure 7.2.: Steps of the validation algorithm](image)

The validation algorithm is graph-based and runs through five phases, which are executed according to the graphical representation of Fig. 7.2. In the following, they are described in detail.
7. Validating process specifications

7.4.2. Step 1: Constructing the internal exception handling graph

The internal exception handling graph $G_I(S)$ for a sphere $S$ is a directed, multicolored graph that has as its nodes the elements of $T^+$, i.e., the tasks and exception handlers used in $S$. The graph has two types of edges, reflecting the control flow relation $<$ and the internal exception handling relation $e$. For each element $(t_1, t_2)$ of $<$, a $<$-edge is inserted between $t_1$ and $t_2$. Likewise, for each element $(t, E, h)$ of $e$, an $e$-edge is inserted between $t$ and $h$. This edge is labeled with $E$. The graph is not allowed to have loops.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Characteristics</th>
<th>Activator</th>
<th>Exceptions</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Activity</td>
<td>atomic, compensatable</td>
<td>initial</td>
<td>E1</td>
<td>-</td>
</tr>
<tr>
<td>T2</td>
<td>Activity</td>
<td>atomic, retrievable</td>
<td>finished(T1)</td>
<td>E1,E2</td>
<td>-</td>
</tr>
<tr>
<td>P3</td>
<td>Signal Proxy</td>
<td>-</td>
<td>finished(T2)</td>
<td>E3</td>
<td>-</td>
</tr>
<tr>
<td>H1</td>
<td>Handler</td>
<td>atomic, compensatable</td>
<td>-</td>
<td>E2</td>
<td>abort</td>
</tr>
<tr>
<td>H2</td>
<td>Handler</td>
<td>atomic, compensatable</td>
<td>-</td>
<td>-</td>
<td>abort, propagate</td>
</tr>
<tr>
<td>H3</td>
<td>Handler</td>
<td>atomic, compensatable</td>
<td>-</td>
<td>-</td>
<td>resume</td>
</tr>
<tr>
<td>H4</td>
<td>Handler</td>
<td>atomic, compensatable</td>
<td>-</td>
<td>-</td>
<td>abort</td>
</tr>
</tbody>
</table>

Table 7.2.: Example Sphere. Top: Components, bottom left: internal exception handling, bottom right: external exception handling

![Internal exception handling graph for the example](image)

An example sphere is described in table 7.2. It consists of the two activities T1 and T2 and the signal proxy P1. Two exception handlers, H1 and H2, are attached to the activities. Furthermore, two exception handlers, H3 and H4, are attached to the sphere itself. The properties of the various tasks and handlers can be seen from the table. The other two tables describe the internal and external exception handling relations.

The exception handling graph for this sphere is shown in Fig. 7.3. Note that no edge is leaving P1, since its exception impacts the external exception handling of the sphere, not its internal exception handling.

7.4.3. Step 2: Deriving execution characteristics

In a first step, the prop* relation is computed by first computing the execution characteristics of composite tasks (according to section 7.3) and then deriving the transitive execution characteristics.
1.4. A validation algorithm for well-formed atomic spheres (according to section 4). For the first part, each composite task has to be analyzed recursively. For the second part, the subgraph containing only e-edges is traversed in reverse order, starting with exception handlers that throw no exceptions themselves and ending with plain tasks. During the traversal, the transitive execution characteristics for each node can be computed as a function of the characteristics of its successors.

In our example, it is easy to see that the prop* function equals the prop function. None of the two conditions given in section 4 are given, since all exception handlers are compensatable.

7.4.4. Step 3: Validating component atomicity and FLEX structure

Now that the prop* function is known, the first two necessary conditions of section 7.3 can be validated. In addition, the critical points have to be identified.

To achieve this, the graph is traversed depth-first from left to right. Each visited node is first checked for atomicity or quasi-atomicity. The structure rules for well-formed spheres are checked by appropriately marking visited tasks and ensuring that no path contains non-retriable activities after the first non-compensatable activity occurred. It is also checked that parallel branches have the same type. Critical points (i.e., a non-compensatable tasks directly following compensatable ones) are marked when they are visited.

In the example, T2 is a critical point, since it is the first non-compensatable task. It is easy to verify that component atomicity and FLEX structure are not violated, because each task is atomic, T1 is compensatable, and T2 as well as P1 are retriable.

7.4.5. Step 4: Identifying critical exceptions

Now we modify the graph in order to validate the third correctness criterion of section 7.3. First, all nodes are removed that do not follow a critical point (this includes the critical points themselves). The removed nodes are those from which a rollback is possible. Hence, the remaining nodes enforce a successful termination of the sphere. Due to the removal of tasks, there may be exception handlers which are not reachable anymore. Those are removed as well, together with all edges connected to them.

In a second step, we add as new nodes all exceptions that may be signaled by the sphere itself. New edges are inserted. These so-called e-edges (critical edges) reflect the external exception handling relation. They are generated by the following rules:

1. If an exception handler h is reached by an e-edge labeled with exception E, and if h has propagation as a possible termination, we draw an c-edge from h to the node for E.

2. If a signal proxy p can raise an exception of type E, we draw an c-edge from p to the node for E.

Now we can derive the set of critical exceptions for S. A critical exception is represented by a node which has an incoming c-edge. The rationale behind the introduction of critical exceptions is that they are signaled by the sphere while it is in a critical state. Hence, correctness requires that the handling of a critical exception cannot lead to an abort of the sphere.

In our example, the removal of nodes leads to the elimination of the nodes for T1 and T2, since T2 is the critical point. The exception handlers H1 and H2 are also removed, since they were only reachable from the removed tasks. The resulting graph, together with the newly introduced external exception information, is shown in Fig. 7.4. The two external exceptions of the sphere, E2 and E3, are added as well as a c-edge from P1 (the signal proxy) to E3. Hence, the only critical exception in this example is E3.
7. Validating process specifications

7.4.6. Step 5: Constructing the external exception handling graph

In a last step, the impact of handling the critical exceptions is evaluated. To this end, we construct the *external exception handling graph* based on the external exception handling relation. It contains as nodes the sphere \( S \) itself and all exception handlers which are transitively reachable from \( S \) through \( e \)-nodes.

The external graph for the example is shown in Fig. 7.5. \( S \) signals two possible exceptions, \( E_2 \) and \( E_3 \), which are caught by the handlers \( H_3 \) and \( H_4 \).

All handlers which cannot be reached through a path starting with a critical exception (as computed in the previous step) can be removed from the graph. (In the example, this is \( E_3 \).) The termination options of the remaining handlers have to be examined. If they include an abort, the correctness of the sphere cannot be guaranteed, since it is possible to have an execution of the sphere in which an exception which occurs after a critical point leads to an abort of the sphere. In the example, it is easy to see that correctness is violated, since the handler \( H_4 \), which is invoked for the critical exception \( E_3 \), aborts the sphere.

If in step 5, no critical handler with an abort option is detected, the correctness of the sphere is validated.

7.5. Complexity of the algorithm

We will now evaluate the complexity of the algorithm described above. To this end, we analyze the different steps.

- **Step 1:** It is easy to see that the size of the internal exception handling graph is linearly dependent on the number of tasks and exception handlers in the sphere, since it has exactly these elements as nodes. Inserting the edges has a complexity which is bound by \( n \times m \), where \( n \) is the number of tasks and exception handlers, and \( m \) is the number of edges listed in the internal exception handling relation, which we assume is represented as a table.

- **Step 2:** If the reverse traversal of the graph is performed breadth-first, each node has to be visited exactly once. Hence, the complexity of this step is linear.
• **Step 3**: All validations of step 3 (atomicity, FLEX-structure, critical points) can be performed in one depth-first left-to-right traversal of the graph. Hence, the complexity is linear.

• **Step 4**: In this step, nodes and edges are removed and inserted based on information which can be found directly in the process descriptions. Hence, what is needed is a traversal of the graph and a lookup of task properties for each node. The complexity of this is bound by $n^2$, if $n$ is the number of nodes in the graph.

• **Step 5**: This step requires a traversal of the external exception handling table and the generation of the external graph based on this table. The complexity is bound by $n^2 + m^2$, where $n$ is the size of the table and $m$ is the size of the exception information for the exception handlers. The size of the former as well as the size of the latter are bound by the number of exceptions in the system.

The considerations above show that the algorithm for the correctness validation of spheres described above has a complexity which is $O(n^2)$, where $n$ is the number of tasks, exceptions, and exception handlers used in the sphere.

### 7.6. Practical considerations

The validation algorithm described above allows to control the correctness of sphere definitions and is thus an important tool for the construction of fault-tolerant processes. Process designers can invoke the algorithm to verify the correctness of their specifications and receive hints on possible improvements. These hints can be easily derived from the algorithm. In the simplest form, the algorithm could present the internal and external exception handling graphs to the user, giving them the chance to comprehend the decision of the system and allowing them to detect possible improvements themselves. A more sophisticated mechanism could make propositions for improvements autonomously. If, for instance, a critical exception is detected that leads to an exception handler which aborts the sphere, the system can provide the designer with a set of possible changes, which include (1) changing the exception returned by the respective task or signal proxy, (2) changing the termination options of the handler which incorrectly aborts the sphere, or (3) if a chain of multiple handler invocations leads to the abort, change exceptions or termination options of one or more of the handlers in the chain. The user can then select the most appropriate out of the options presented.

A general problem of the correctness notion underlying the algorithm is its restrictiveness. The algorithm evaluates the worst case, detecting all possible violations of atomicity. Such a violation can, however, be caused by an exception that is very unlikely to occur, which means that a process is rejected although the probability that atomicity will be violated is very small. For practical reasons, modifications to the algorithm are possible that take a more pragmatic approach:

1. **Ignore exceptions in the validation process.** The user can instruct the system to check only the normal process without considering the exception handling mechanism. Given that exceptions are unlikely to occur, this allows to validate a basic level of correctness, without the guarantee that the system can enforce atomicity in all cases. Instead of compile-time checks, the system can detect atomicity violations caused by exception handling at runtime, and signal a specific exception in this case which calls for human intervention. Relaxing the mechanism in this way can be valuable during the test phase of a new process, since it allows to detect which violations are likely to occur.

2. **Allow to ignore certain violations explicitly.** Ignoring all correctness violations caused by exceptions may be a too relaxed solution in some cases. Instead, an interactive mechanism is
possible, which presents the user with the correctness violations found, asking them to correct the failures, but permits to install a process even if some of the failures are not fixed. This weakening of correctness has to be confirmed by the user explicitly, and the run-time mechanisms described before have to be in place in order to detect violations at run-time.

(3) Statistical support: Given that the process support system contains a history component which logs all relevant information during process execution and allows to analyze the stored data, it is possible to provide statistical support for the decisions of the modeler. If, for instance, a process is composed out of predefined components which have already been used in other processes, the system will have information about the probability of certain exceptions and the typical decisions of the exception handlers. This information can serve as the basis for deciding which correctness violations can be ignored (because they are very unlikely to occur) and which have to be corrected. The history information can also be used to gradually improve process specifications, for instance by analyzing a process after a number of instances have been executed and finding exceptions which are raised very frequently. An exception that is raised often should be incorporated in the normal control flow. Likewise, branches in a process which are almost never executed, can be changed into exceptions in order to keep the process as near to the actual application semantics as possible.

7.7. Discussion

The purpose of this section is to discuss the properties of the complete approach to fault-tolerant processes presented in the last three chapters.

7.7.1. Advantages of the solution

The language extensions presented allow the elegant specification of fault-tolerant workflow processes. These extensions are not present in existing systems. The language extensions, along with the corresponding system support, will result in cleaner process specifications and less overhead when designing fault-tolerant workflow processes. Regarding the system support, much of the implementation details can be directly derived from the descriptions above as the new primitives are to be treated as standard constructs and will not require significant changes to the engine (for instance, an exception can be treated as an activity which is triggered when another activity raises an exception. This involves minimal changes to the control flow logic).

To summarize, the extensions presented meet important criteria for a flexible process exception mechanism:

- **Support for flexible recovery strategies**: Processes and spheres provide natural boundaries for partial backward recovery. Semantic recovery mechanisms like compensation and holistic backout ensure the necessary flexibility of backward navigation, while exception handlers guarantee forward progress.

- **Reusability**: It has been shown in the example of section 6.4 that the proposed primitives improve the reusability of process descriptions (because of the separation of business logic and failure handling semantics) as well as the reusability of tasks in different contexts (because of the parameterization realized through override handlers). This is a significant advantage over what is possible in current systems. Furthermore, failure handling strategies can be re-used since exception handlers are registered with the system and can thus be applied in various processes. As a result, it is possible to reuse defined processes and activities without further modification and without having to redo the exception handling procedures entirely.
• **Openness:** The mapping mechanisms we have described in Section 6.3 allow the incorporation of arbitrary applications, ranging from legacy applications over CORBA-like services to workflow-aware programs that can utilize the OPERA API. Hence it is possible to achieve a seamless integration of external applications and environments with the fault tolerance mechanisms present in the system.

Our model requires only minimal modifications to the representation of the business logic. The above shows that it is only necessary to add spheres in order to specify atomicity. Since all other recovery related information is described separately in the exception handlers, the process description remains comprehensible.

The validation mechanism described is an important contribution towards the rapid development of fault-tolerant processes. It provides process designers with the ability to control relevant properties of their process specifications and can be seen as a first step towards more sophisticated verification mechanisms for distributed processes.

### 7.7.2. Related work

Several research projects have focussed on the integration of databases and non-databases into distributed computing environments, including work on extended transaction models (ETM) in distributed object management (DOM) [BOH+92, GH94, GHM96], where a framework for flexible transaction structures in workflows was developed, and the ConTracts project [WR92], which focussed on long-running applications and provided relaxed atomicity based on compensation and forward recovery. Recently, the work on recovery in transactional workflows has been extended in [CD96] and [CD97b], where recovery mechanisms have been developed for transactional workflows that consist of transaction hierarchies with arbitrary deep nesting.

In addition, a considerable amount of work towards flexible recovery has been done in the context of advanced transaction models [BDS+93, ELLR90, Elm92, GHM96, GS87, WS91, WR92]. In particular, the EXOTICA project [AKA+96] investigated the mapping of advanced transaction models to workflow description languages and showed how some of the concepts used in transaction management can be used in workflow environments. The approach presented in here differs from this work mainly because of its strong focus on modeling language aspects and because we do not assume a transactional environment.

There is no support for exception handling in currently available commercial WFMS. The problem of recovery from activity failures has, however, been considered in a number of research projects. [Ley95] introduces the notion of a sphere of joint compensation, which is a subset of a process’ activities. If one activity of a sphere fails, the whole sphere has to be backed out (either by compensating each step that succeeded so far or by executing a special higher-level compensating activity). The special properties of the concept (spheres do not have to form a connected graph, and multiple spheres can overlap) leads to very complex and incomprehensible semantics. [EL96] describe recovery facilities in WAMO, a research WFMS. Processes are treated as workflow transactions. Component activities can be declared vital for a process. A running process is aborted and compensated if a vital activity fails. Failures are classified into two categories (system failures and semantic failures). A further distinction of exception types is not possible. The system does thus not support the specification of different handling strategies for different classes of failures. Resumption of failed processes is not possible.

The work on exception handling in programming languages [Par72, Goo75, YB85, vZBC96, Seb96] has provided the basics for our approach. While it provides constructs for the seamless integration of exception handling into workflow descriptions, it lacks the consideration of practi-
7. Validating process specifications

cal aspects that become important in workflow systems such as the participation of autonomous, heterogeneous legacy systems and the strong impact of human intervention.

A problem we did not consider in this work is the correct handling of multiple exceptions that occur concurrently in multiple parts of a system and can only be handled together. Specification and failure resolution techniques that can be used for this case have been developed in the context of complex concurrent and realtime distributed systems [JC86, Rom97, XRR98]. They do mostly rely on a notion of atomic actions that is very similar to the notion of atomic spheres used in workflow contexts.
8. Event-based Intertask Communication

8.1. Introduction

In chapter 6, exceptions were characterized as typed signals raised by blocks or processes to indicate failures to the enclosing sphere. The exception concept can be seen as an inter-process communication mechanism, albeit reserved for specific situations (failures) and allowing only vertical communication between a task and its sphere. Generalizing the idea of inter-task signals can lead to a communication mechanism allowing the arbitrary exchange of information between activities and processes. The existence of such a mechanism can widen the scope of a PSS significantly, since it allows to model processes which are cooperative, i.e., rely on the exchange of information between running tasks. The purpose of this chapter is to develop such a communication service, describe its integration into the process model, and discuss its semantics in the context of atomic spheres. It is well-known that the integration of communication mechanisms and atomic units generates a number of consistency problems. This is not a problem particular to the event-based scheme we present here – it can be found, for instance, in databases too, where the concept of "recoverability" [BHG87] covers the possibility to undo work of a transaction without affecting other transactions. We will discuss the problem of recoverability for event-based mechanisms in this chapter.

The chapter is structured as follows. Section 8.2 presents an example motivating the need for inter-task communication. Section 8.3 discusses communication paradigms and their applicability in a PSS context. Section 8.4 presents an extension for process modeling languages allowing event-based communication. Section 8.5 discusses the problem of recoverability and presents a solution based on various recovery-modes giving the process designer control over the behavior of the process. Finally, a discussion of the approach can be found in section 8.6, and section 8.7 compares it to related work.

8.2. Motivation

As a motivation and running example, we will use a business process involving the cooperation of workflow systems in an administrative environment. The example we use has been taken from a real application scenario [SST98] and represents a typical administrative environment with several heterogeneous systems. In this environment, administrative information is stored and maintained in SAP R/3. SAP R/3 has its own workflow component [SAP97] allowing to model sequences of operations to be executed in different modules of the system. The workflow functionality is, however, restricted to R/3 activities, and the integration of external programs is only possible if these programs have been linked against SAP runtime libraries. To interact with the rest of the system, a PSS is used, which controls the overall process and invokes R/3 workflows as subprocesses.

Fig. 8.1 shows a process executed in such a configuration. The process handles the delivery of products by a retail company and captures all steps from the initial placement of the order by a customer to the final payment. The process has been divided into two sub-processes, warehouse
8. Event-based Intertask Communication

administration and order handling. The company is using SAP R/3 for stock administration, which requires to model the sequence of steps pertaining to warehouse administration as a SAP workflow and leave its execution to R/3. This subprocess, shown in the lower part of the figure, consists of four steps. First, the availability of the requested items is checked. If the item is not in stock, it has to be ordered from the manufacturer. If it is in stock (or when it has been delivered by the manufacturer), it can be ordered from the warehouse, which requires updating the stock databases. Finally, an invoice is produced based on the quantity taken from the warehouse, which completes the process as far as R/3 is concerned. The overall process, shown in the upper part of the figure, contains the initial placement of the order, which includes generating the necessary record of the products ordered. The “Get item” task, executed next, serves as a placeholder for the R/3 process. Upon execution of this task, the workflow tool initiates the execution of the R/3 process. After the products have been extracted from the warehouse, an acknowledgment is sent to the customer announcing the expected delivery date. The final steps of the workflow are concerned with packaging and shipping of the product and with the handling of the payment.

The disadvantage of this way of modeling the application is the lack of cooperation between the two processes. Each process appears as a “black box” to the other. In fact, the invocation of subprocesses resembles the RPC mechanism used in distributed computing environments, leading to the “blocking” of the invoker until the call has finished. In the example, this leads to unnecessary delays since the acknowledge to the customer could be sent as soon as it is known that the product is on stock. Such an early notification is an important contribution towards customer satisfaction. As another example, the shipping could be initiated as soon as it is known from which warehouse the product will be delivered and at which date is will be available. In this way, all preparations could be finished by the time the product is available and shipping could start without further delays.

The examples show the need for a communication mechanism allowing tasks to communicate. Such a mechanism cannot only shorten the response time by increasing the degree of parallelism. It is also a necessary requirement for many types of applications:

- The integration of applications which are characterized by a long duration and the generation of asynchronous events during their execution to which a process should react. A good example is the integration of groupware tools (e.g., conferencing systems) into workflow systems [SL95]. While a conference has no predefined structure and lacks the rigid control flow specification of a process, there are nevertheless certain situations, like an agreement met on a specific sub-topic, which may require to trigger a task (for instance, preparing...
documents concerning this particular topic) or may influence the flow within a running process [SL95].

- For many employees, the strictness with which workflow systems enforce the process specification limits their acceptance. The higher the skills of a user, the higher the probability that she will refuse to follow rigid prescriptions of a particular order of activities. Intertask communication mechanisms allow to implement non-prescriptive processes. Such a process contains a list of tasks to execute rather than a prescription of the required order of execution. The employee can decide autonomously on the control flow and will signal the completion of each particular task using the communication facilities. By this, the ability of the system to log relevant state changes is retained, while at the same time greatly increasing the flexibility.

### 8.3. Communication paradigms

The aim of this section is to give a brief overview of potential communication paradigms and to evaluate their use in building a communication service for process modeling languages. Three principal paradigms can be distinguished, which are also depicted in Fig. 8.2.

- **Shared memory.** Communication through shared memory is a common approach which can be found, for instance, in databases (where multiple transactions access the same data) or in coordination languages like LINDA ([GC92], also described in chapter 3). They are also part of the standard interprocess communication mechanisms offered in most operating systems. In this scheme, information exchange takes place through writing or reading data objects accessible to all communication partners. The sender of a piece of information simply writes it to a common data object, where it can be read by the receivers. While this provides a comfortable abstraction for exchanging even large amounts of data, additional means have to be taken to notify the receiver of the information arrival. To achieve this, mechanisms like message passing, polling, or condition variables are needed. Furthermore, to avoid inconsistencies due to concurrent access to shared objects, shared memory mechanisms are usually accompanied by synchronization services like locks (database management systems), semaphores, and condition variables (operating systems).

  From the perspective of a process modeling language, shared memory mechanisms have the disadvantage that they do not solve the notification problem. Hence, it would also be necessary to add a message mechanism. Furthermore, the integration of appropriate synchronization services further increases the system complexity. Hence, using a shared memory approach for inter-process communication would increase the language complexity significantly.

---

![Figure 8.2.: Communication paradigms](image_url)
8. Event-based Intertask Communication

- **Message passing.** Message-passing schemes for process interaction are offered by most operating systems and are also the base for many distributed programming paradigms like PVM [GBD+94] or MPI [For94]. They are based on direct communication, with the sender knowing all receiver's addresses. Two general cases can be distinguished: Synchronous message passing, where the message exchange takes place at the same time, and asynchronous mechanisms, where the sender's message is put into a mailbox or queue and the receiver can check for arrival without being blocked. A common characteristic of these mechanisms is that the sender has to know the receiver's address in order to deliver the message. Changing the number or address of receivers requires a modification of the sender. This characteristic can be overcome by using group-communication mechanisms based on multicast messages, but only at the cost of higher system complexity.

From the perspective of process modeling, the direct addressing of receivers by the sender becomes a problem because it complicates the reuse of components. Consider a subprocess that is incorporated in different contexts, where each context defines different receivers for messages sent. For each new application, all addresses in all send operations have to be modified. Obviously, the problem can be circumvented by adding appropriate abstraction mechanisms (for instance, use receiver addresses as parameters given to a process at startup), but this increases the complexity of process descriptions. A mechanism that inherently contains appropriate abstractions would be preferable.

- **Event-based mechanisms.** This class of mechanisms is characterized by a publish-subscribe pattern of interaction [KR95, Eis99]. The sender announces a specific set of messages (called event in these models) it may send (signal) during execution. Processes willing to receive messages can subscribe to these events and are notified once such an event occurs. In this type of model, a sender does not need to know the subscribers of its events, and an arbitrary number of subscribers can exist. Because of this, it is very simple to change the patterns of communication, since all subscription information is encapsulated in the event service. No processes have to be modified. Like in message-passing approaches, it is possible to distinguish between synchronous and asynchronous event delivery. With synchronous delivery, events are delivered to subscribers immediately, and a handler in the receiving process is responsible for reacting to the event [Eis99]. In asynchronous schemes, events are stored persistently by the event service. This has the advantage that it is possible for subscribers to react not only to simple event occurrences, but to specify complex predicates over sets of events which trigger specific reactions [GGD94, KR95, GT96].

For the purposes of a process modeling language, event-based communication is the best-suited paradigm because of two main reasons. First, the mechanism is very simple and requires only a small number of extensions to the process model. No synchronization primitives as they would be required by shared memory approaches are needed. The simplicity of the mechanism allows for more comprehensible process specifications and is a significant contribution towards an efficient design. Second, the model fosters modularity since no information about receivers is kept by sending processes. Signaler and subscribers are decoupled to a degree which allows a compositional style of process modeling, where only the event service has to be modified to adapt the communication patterns.
8.4. Adding event-based communication to modeling languages

In this section, we will describe how a process modeling language can be enhanced with event-based intertask communication mechanisms based on publish-subscribe principles as described in the last section. The extension is based on the notion of an event as it is used, for instance, in active database systems [PDW+94]. Hence, events are typed and parameterized in order to allow the transport of context information.

The use of events in a process support application is best illustrated using our previous example. Here, the R/3 subprocess would raise either an event product_available or an event product_not_available as soon as it is known whether the product is in stock or needs to be ordered from the manufacturer. Both events could as well carry the estimated delivery date as a parameter. Upon receiving the signal, the base process can proceed by sending the corresponding acknowledgment to the customer and preparing the shipment based on the estimated delivery date.

8.4.1. Adding events to the process model

Our definition of an event is similar to that of an exception as it was introduced in chapter 6:

- An event type is a tuple \((N, PL)\), where \(N\) is an event name and \(PL\) is a parameter list. Event types are registered with the system as part of a task declaration. During execution, tasks signal events (i.e., create event instances) by providing an event type and bindings for all parameters in the parameter list. Like with exceptions, the parameterization allows to pass arbitrary context data with the event, from simple boolean values to references to external objects. A task is free to signal an event arbitrarily often or even not at all. This gives the event mechanism the necessary flexibility to capture all situations of interest.

- Events are signaled in the same way as exceptions, namely through pseudo-activities similar to the signal proxies described in chapter 6. They can be embedded into a process model but have no external binding. Instead, the system generates an event if such a pseudo-activity is executed. The advantage of this approach is its seamless integration with the control and data flow primitives already present in the language.

- Events have to be declared. At run-time, a task can signal only the events listed in its event-declaration which is, like the input and output parameters, part of the signature describing the task interface. As an example, see the interface structure of the warehouse administration task from our initial example (Fig. 8.3 shows the interface in a tree representation). The interface of a task consists of declarations of the parameter lists (inbox, outbox) and
the event declaration. Here, two event types are defined. Each of them has one parameter of type DATE, which allows to communicate the expected delivery date of the ordered product. While the task is executing (remember that this implies the execution of the R/3 subworkflow), it may at any time raise one of the two events, thereby signaling either that the product is on stock or that is has been ordered and returning the estimated delivery date. Practically, this will happen when the first activity of the subworkflow has finished. (The SAP workflow module, for instance, has a callback mechanism that can report state changes in an workflow to the invoker through its API and that can be used to signal the event.)

### 8.4.2. Extending the representation of tasks

Adding events to the process model requires to extend the task model presented before (Fig. 4.2). To this end, we add an event declaration and an event queue to the Task class representing the root of all task types in OCR. The new interface of the class is shown in Fig. 8.4. The event queue records all events signaled by the task in the order they occurred and forms the basis for the integration of events into control flow decisions.

A process can react to an event by making the start of a particular task dependent on events that occur in other tasks of the same process:

- In OCR, this can easily be achieved since the guard concept provides already an event-like mechanism for control flow determination. Hence, the only extension needed is to allow activators to refer to event occurrences. An activator of the form `signaled(t1, e3)`, for instance, would specify that an activity is to be started when event `e3` appears in the event queue of task `t1`. More complex specifications are possible, as noted in 8.4.3. A reference to an event has always to be accompanied by a specification of the required recovery mode. The recovery mode determines the reaction of the system if the event should be revoked due to the abort of a task. This issue is discussed in Section 8.5.

- In graphical representations like the FlowMark model which was described in section 4.4, a new class of control connectors can be introduced. An event based control connector (ECC) is a quintuple `(T_T, T_S, C_{Ev}, C_{Act}, RM)`, with two new elements, an event condition `C_{Ev}`, which can be an arbitrary predicate over the event queue of the source task, and a recovery mode `RM`. During process execution, the target task is considered for execution as soon as the predicate evaluates to true, which may be any point in time before the source task has finished. An ECC allows, thus, to define the flow of control based on the occurrence of events, while the original, state based control connectors, can only react to the finished state of tasks. CCs and ECCs can be arbitrarily combined in the same process, allowing both event driven and flow driven execution.
8.4. Adding event-based communication to modeling languages

8.4.3. Intra-process communication

Without further specifications, the visibility of an event is limited to the enclosing sphere of the signaler, i.e., to the block or process a task belongs to (similar to local variables in a programming language). This allows for a basic form of communication between a process and a signaling task by using the event references described above. The activators of other tasks belonging to the same process can refer to the event queues of sibling tasks, which allows to start new programs if certain patterns of events occur. References can take the form of arbitrary predicates over the structure of one or more event queues. Note that discussing suitable event algebras for constructing complex events is beyond the scope of this thesis. This topic has been deeply investigated in the context of active database, and an exhaustive treatment can be found, for instance, in [Gat94]. For most PSS purposes, a comparatively simple algebra will be sufficient, with only conjunction and disjunction as operators. The reference to multiple event queues in a graphical representation is difficult because control connectors usually link exactly two tasks. The introduction of multi-edges would be necessary to overcome this limitation.

The principles of the intra-process communication mechanism are illustrated in Fig. 8.5 using the initial example, now described including the event mechanism. To keep the picture comprehensible, we have omitted a number of details like the full definition of all tasks and certain aspects of data flow. The GetItem task is now defined according to Fig. 8.3, declaring two event types. (Remember that this task represents a whole subprocess, executed by the R/3 system.) We have added two event based control connectors (the broken lines) that react to the occurrence of the respective events in the SAP workflow. We have also divided the shipment activity into two steps, where the first one prepares the shipment (by allocating resources, transport facilities, and the like), and the second one triggers the actual shipping. If the requested item is on stock, SAP will signal the PRODUCT_AT_STOCK event, which initiates (through an ECC) the execution of the shipment preparation step. Then the acknowledgment step is started, which informs the customer of the estimated delivery date. If the product has to be ordered, the shipment preparation is omitted. In this case, the Acknowledgment task is directly started, informing the customer of the delay. The actual shipping activity is executed when the subprocess (and with it the GetItem task) has finished, ensuring that the shipment starts properly as soon as the product is available. Note that in this example, the imported event queue of the process is not used since all event exchanges are
among tasks at the same level in the process.

### 8.4.4. Inter-process communication

To enable inter-process communication, it is necessary to make events visible outside of their enclosing block. To this end, a subscription mechanism is used. A process can register for any visible event, using a subscription statement of the form \((T, E, N)\), where \(T\) is the task the event is imported from, \(E\) is the event type, and \(N\) is a nickname under which the event is made accessible to the component tasks of the subscriber. Similar to the event queue of a task, each process has an imported event queue (IEQ) which makes all imported events visible to its component tasks. This mechanism supports the propagation of events over several levels of nesting, since a subscription statement issued by a subprocess can refer to an event of its parent's IEQ.

Similarly, each process has an exported event list which allows to externalize events signaled by its component tasks. It contains declarations of the form \((T, E, N)\), where \(T\) is a component task of the process, \(E\) is an event declared by it, and \(N\) is the nickname under which it is accessible to other processes. The exported event list forms, together with the event declaration of a process, the complete declaration of all events the process may signal. The export mechanism allows events to arbitrarily "climb up" the nested scopes of a process, while the import mechanism lets them "climb down". Together, they allow for inter-process communication while the ECC concept supports intra-process communication only.

Fig. 8.6 illustrates the inter-process communication mechanism. It describes the same application as before, but this time modeled using two parallel subprocesses communicating through events. Upon start of the parent process, both subprocesses are started, and the PlaceOrder activity is invoked (it is the only activity marked as StartActivity, enforcing its immediate begin). When it ends, as well as after the CheckAvailability task and at the end of the StockAdmin subprocess, events are generated and delivered to the respective other subprocess. Note that each of the two subprocesses subscribes to the events declared by the other, and that activities can access all events through the imported event queue using ECCs.

This second example shows the close relation between the interprocess communication mechanism and event based workflow architectures [GT96], giving the workflow designer the option to choose between two flavors of modeling: Either using process/subprocess hierarchies with RPC-like semantics, or combining independent processes that communicate by exchanging events. When to use each one of them depends on design methodology and the particular characteristics of the application environment.

### 8.5. Recoverability

In conjunction with atomic spheres, the event mechanism can lead to inconsistencies. In our initial example, for instance, consider the case that the warehouse administration is aborted (it might turn out after the product is requested from the manufacturer that it is not produced anymore). We can consider this case as an exception that requires, similar to the occurrence of a failure in database transactions, the abort of the process. The interesting question is now how to treat the events that have already been signaled by the aborted process. Clearly, in the above example the obvious solution would be to inform the customer and then abort the overall process too. There are, however, many cases in which to abort the receiver of the event is not the best solution. In many situations, the invalidation of an event can be solved by the execution of some alternative activities which allow to substitute the failed activity. In other cases, only a partial abort might be required, while some of the effects already executed have to be made persistent.
8.5. Recoverability

We will denote the case in which an event that was already observed by a subscriber is undone as event revocation. The resulting effect, possible inconsistencies, is similar to the problem occurring in databases if transactions read uncommitted data. The acceptable solutions are, however, fundamentally different. In a database management system, the strategy mostly used is to restrict the range of allowed interactions in order to avoid non-recoverability. In most practical implementations, cascading aborts have also to be avoided, leading to the fact that transactions are never allowed to read uncommitted data [BH87]. In our context, events are deliberately used for communication, and restricting the allowed patterns would not be acceptable. Instead, we would like to give the process designer control about the necessary recovery strategy, which may include even cascading aborts. Note that recoverability is no problem in our context as long as compensation is possible.
After introducing some necessary notation, we will discuss a family of recovery strategies in the remainder of this chapter.

**Definition 5 (Notation)**

- Let $\text{SET}(t)$ denote all tasks that are active at time $t$. An active task is one whose event queue contains events that may be subject to revocation.

- The causality relation $\text{DEP}_E(t)$ contains all pairs $(\text{PROD}, \text{CONS})$ of tasks where $\text{CONS}$ is causally dependent on the active task $\text{PROD}$ because the execution of $\text{CONS}$ was triggered by an event in $\text{PROD}$'s event queue, or, in other words, a process has an event-based control connector with $\text{CONS}$ as the target and an event of $\text{PROD}$ was referenced in the predicate. Note that $\text{PROD}$ has to be active, but $\text{CONS}$ can be an already finished task. This captures the fact that a consumer may finish before the producer it depends on. In these cases, compensation is necessary to undo the effects of the consumer "post mortem".

- Let $\text{DEP}_E(t, T)$ be a shortcut for $\{\text{CONS}|(T, \text{CONS}) \in \text{DEP}_E\}$.

### 8.5.1. The basic algorithm: exception dissemination

We intend to base our recovery algorithms on the exception functionality which is already present in the system. This has the advantage that as little new concepts as possible have to be introduced. The exception mechanism is an ideal "transport medium" to initiate recovery in the case of event revocation. The base algorithm for recovery is simple (Fig. 8.7): If a task $T$ is aborted, the system determines $\text{DEP}_E(t, T)$. To this end, a registry of all currently existing consumer-producer relationships has to be maintained in a persistent table. In all subscribers determined this way, an exception of a predefined type is generated which, through the usual exception handling mechanism, initiates the customized recovery procedure as defined by the workflow designer. This procedure is executed in two phases. First, if defined, an exception handler is executed which performs contingency and/or cleanup operations. This utilizes only the exception mechanism and requires no new functionality. What has to be added is a termination protocol invoked after the handler was executed. The termination protocol determines how to perform recovery, by either aborting the subscriber completely or performing partial rollback. Practically, the termination protocol to use is determined by the recovery mode which has to be specified with each event subscription.

Note that the order in which the exception dissemination protocol is executed is important. The import and export mechanisms permit a task to have subscribers on several levels of the same
8.5. Recoverability

process tree. In these cases, the exceptions are handled in a top-down style: If a parent process and one of its descendants have a causal dependency on the same task, the parent process has to be handled first. This avoids unnecessary work and speeds up processing, since aborting the parent results in an abort of its children as well, making exception handling in the child obsolete.

8.5.2. Recovery modes and termination protocols

The recovery mode specified by the process designer for each event-based control connector determines the fate of a consumer task after the execution of the exception handler. Based on the recovery mode, a termination protocol is executed. The following recovery modes and termination protocols exist:

**IGNORE.** No further action is performed, which can be useful if (a) an abort is impossible or too expensive because of the amount of work already performed, or (b) if it is known from the application semantics that no recovery is necessary. Consider our initial example, where the Acknowledge task informs the customer. Since this task left unrecoverable side-effects in the outside world, the obvious strategy is not to abort it, but invoke a contingency action (send a new letter) through the exception handler.

**ABORT.** Abort the parent process of the consumer. This is useful in those cases where after event revocation no forward progress is possible and hence a complete abort is the only feasible strategy.

**PARTIAL ROLLBACK.** This is the most interesting recovery mode. It performs a partial rollback, undoing all work causally dependent on the aborted task. After this rollback, the execution continues from the last consistent state before the the original occurrence of the event. While the principle is simple, its implementation turns out to be complicated because of the complex structure of the causality relationship. We investigate this in detail in the next subsection.

8.5.3. The Partial rollback algorithm

Between all tasks in the system, a causal dependency [Lam78] is defined through the ordinary control connectors, data exchanges through the whiteboard, and the event mechanism. The goal of the partial rollback algorithm is to set the system back to a consistent state by undoing all effects causally dependent on an aborted task. To this end, the transitive closure of the causal dependency relation is computed and tasks are aborted in the reverse order of their execution. We define the relation $DEP_{ALL}(t)$ over all pairs of tasks in the following way: $(T_1, T_2)$ are in $DEP_{ALL}(t)$ if (1) $T_2$ was executed after $T_1$ and both were linked by a control connector, or (2) if $T_1$ raised an event that caused the start of $T_2$, or (3) if $T_2$ read data from the whiteboard that was written by $T_1$. This dependency information can easily retrieved, since usually a workflow system logs all information pertaining to process execution in its execution log for legal reasons and to facilitate process optimization based on statistics [AGL98]. We denote with $DEP^*(t)$ the transitive closure of $DEP_{ALL}(t)$. As a shortcut, we define $DEP^*(T, t)$ as the restriction of $DEP^*(t)$ on those tasks causally dependent on $T$, i.e., $DEP^*(T, t)$ contains all tasks transitively causally dependent on $T$.

The goal of the partial rollback algorithm is now to undo all work dependent on an aborted task by aborting all tasks in the dependency set of this task. The algorithm maintains two data structures: $S_{DONE}$, storing all tasks that have already been handled by the recovery algorithms, and $Q_{TODO}$, a FIFO-organized list of tasks yet to be considered for recovery. At the beginning of the algorithm, $S_{DONE}$ is initialized with the task that originally failed and caused the recovery
procedure and with $T$, the task that caused the start of the partial rollback algorithm. The $Q_{TODO}$ is initialized with all consumers of events signaled by $T$. The algorithm proceeds in two phases:

1. First, determine all tasks that are dependent on $T$ because of regular control and data flow. These are tasks belonging to the same process as $T$. Abort these tasks in reverse order of execution. If an aborted task had consumers, add them to the $Q_{TODO}$.

2. Perform a breadth-first traversal of the graph containing all causal dependencies on $T$ caused by events. This is done by traversing the $Q_{TODO}$: An exception is sent to the task at the head of the $Q_{TODO}$, which is resolved through the exception handling mechanism. If the task has the PARTIAL_ROLLBACK option set, its consumers are appended to the $Q_{TODO}$, and the task itself is added to the $S_{DONE}$. Then the algorithm continues by taking the next task from the head of the queue. If a task extracted from the queue is already in the $S_{DONE}$, it is ignored. This algorithm proceeds until the queue is empty.

After the execution of the algorithm, the system is in a consistent state again. In large process sets with many interprocess dependencies, using only partial rollback strategies can cause significant overhead. The process designer, however, has the option to limit the amount of rollback work to be done by deliberately choosing ABORT or IGNORE recovery methods whenever they are applicable.

Fig. 8.8 illustrates the recovery algorithms. There, three processes communicate by exchanging events. At the top, we show the current state of the processes at the point in time when a failure occurs in Process1, which leads to its abort. At this point, the first 2 activities of each process have finished, and the last ones are still running. The abort of Process1 leads to the revocation of E1. Subsequently, the exception dissemination algorithm sends exceptions of type REVOKED.EVENT to T22, which was the only consumer for this event and has the recovery mode PARTIAL_ROLLBACK. If an exception handler was defined in T22, it is executed now.
Then the partial rollback algorithm is started, with $S_{DONE}$ containing $T_{11}, T_{12}, T_{13},$ and $T_{22}$. $Q_{TODO}$ contains $T_{32}$, since it consumed the event $E_2$ signaled by $T_{22}$. The first, local phase of the algorithm leads to the abort of $T_{23}$, which has generated no events. Then, the second phase is started with an exception being generated in $T_{32}$ which is at the head of the queue now. This task had generated event $E_3$, which was consumed by $T_{13}$, so $T_{13}$ is appended to the $Q_{TODO}$. Exception handling for $T_{32}$ leads to an abort of $T_{33}$, and then $T_{13}$ is extracted from the queue. Since this task is already in the $S_{DONE}$, it is ignored. Now the queue is empty, and the system is in a consistent state again.

### 8.6. Discussion

The mechanisms described give the process designer the freedom to choose from a family of recovery strategies in the case of a failure. By combining the exception handler option and the recovery method, it is possible to implement the following algorithms:

- Ignore the failure by providing no exception handler and choosing the strategy IGNORE. This is useful in all cases where a task had no side effects and thus no recovery is necessary.

- Perform additional operations, but no abort (strategy IGNORE, provide an exception handler that invokes alternate activities). This will be used if contingency operations need to be invoked, but an abort is not necessary or possible.

- Abort the process containing the failed task, which is the strategy to choose when no graceful degradation is possible and a complete abort is the only solution.

- Invoke contingency operations that replace the invalidated task (strategy ABORT, provide an exception handler implementing the contingency logic). This is the option to be used if semi-atomic tasks have to be aborted and some cleanup work has to be done to really undo their effects.

- Rollback to the state before the event occurred, and continue work from there (use the PARTIAL-ROLLBACK method).

In addition, the solution has the advantage of integrating event mechanisms into the process server instead of relying on underlying event-based technology. Our language extensions allow to enhance traditional, flow-oriented languages with event mechanisms, giving the workflow designer the option to arbitrarily combine both paradigms. We have especially focused on the recovery problems existing if interprocess communication is used in transactional workflow environments.

The algorithms we have presented are likely to widen the scope of applicability of workflow techniques by allowing to implement new modes of interaction between otherwise isolated processes. "Cutting windows into the black box" by using the event and exception mechanisms proposed will lead to better process models since designers do not have to use workarounds like using flat, unstructured process models if interaction between applications is essential. Our approach, as shown in the examples, also provides the means for the integration of heterogeneous workflow tools.

The combination of interprocess communication, event-based exception handling, and atomic spheres delivers a very powerful programming model facilitating the specification and enactment of robust and flexible processes suitable for mission-critical applications. The algorithms for recovery in the presence of event revocation provide the process designer with the option to choose
from a family of recovery strategies, allowing them to gradually adapt the system behavior to the application needs.

8.7. Related work

It was already mentioned that recently, a number of distributed event environments have been developed [GGD94, KR95, GT96, Eis99]. None of the approaches, however, discusses the problem of recoverability.

[CCPP96] provides a classification of possible workflow interactions if the processes access common data. A workflow integration approach is proposed that aims at identifying and removing interferences between concurrent workflows. The paper does not discuss event-based interaction of processes, and aspects of exception handling and recovery are not evaluated.

Contrary to what might be expected, the work done in the context of active database management systems (ADBMS) [PDW+94, WC96] does not apply directly to what we discuss in this work, although the ideas proposed could be implemented on top of an ADBMS. ADBMS allow to define events as part of the event-condition-action (ECA) rule paradigm: Upon the occurrence of an event, which can (among others) be an operation executed as part of a transaction, the associated condition (a predicate defined on data objects) is evaluated and, if successful, the action, a subtransaction containing arbitrary operations, is executed. While this principle appears to be related to our approach, a closer look reveals a fundamental difference: ECA rules cannot be used to link existing transactions. Instead, the action is executed either as part of the transaction raising the event (in form of a subtransaction), or a new transaction is started. Both cases simplify recovery tremendously because the action can simply be aborted in the case of an event revocation. This may, in the worst case, require cascading aborts, but the isolation provided by the transaction ensures that no interferences with other transactions have to be taken into account. Since we distinguish spheres of atomicity and spheres of isolation as separate concepts (meaning that a set of interacting processes may well consist solely of atomic spheres without any isolation enforced by the system), event revocation is much more complicated. The difference can be illustrated by looking at the the dependency graph defined by the producer-consumer relationship of events. While this is a tree in active databases, it is a mesh in our case.

The same is true for other database-oriented areas of research, for instance the large body of work on controlled interaction of transactions [Elm92]. All of these models rely on common data as the medium of interaction between transactions (or at least on some notion of shared information) and, because of this, cannot be used in event-based workflow environments. Finally, event-like mechanisms have been proposed in the context of groupware frameworks in [SL95, SS96]. While the general idea, exchanging events between isolated applications, is similar to our approach, the mechanisms we have developed go much further because of their tight integration with exception handling mechanisms and the possibility for recovery.
Part II.

PSS Architecture,
Scalability and Availability
8. Event-based Intertask Communication
9. Runtime Architecture

9.1. Introduction

The runtime components of a process support system handle all aspects of process execution (as opposed to modeling and analysis aspects), which encompasses a wide range of services to be provided. The required functionality includes determining the order of execution of programs, scheduling of applications, allocation and management of resources, logging of relevant events, and monitoring the state of selected processes. The primary goal in developing such an architecture for the OPERA PSS kernel was to develop a framework rather than a closed architecture, given that the kernel has to be easily adaptable to add new services. From a research point of view, what is interesting is not so much the basic architecture, but rather the extensions leading to a distributed, fault-tolerant framework. Hence, the presentation in this chapter will cover the basic architecture only insofar as this is needed for discussing the extensions in subsequent chapters.

This chapter is structured as follows. In section 9.2, the general principles of the system design which was chosen for OPERA are described. Section 9.3 presents an example illustrating the interaction of the components. In section 9.4, we discuss extensibility, describing how new types of external applications can be added to the system. Section 9.5 motivates the need for a distributed PSS architecture and presents the concept of a PSS cluster. In section 9.6, the implementation of such clusters is discussed based on experiences with building the OPERA system. Section 9.7 provides a short summary.

9.2. General Overview

A bird eye’s view on the OPERA architecture is shown in Fig. 9.1. Logically, the system consists of a number of modules, each of which encapsulates a specific part of the PSS functionality. (In the actual implementation, each of these logical modules is implemented as an object with a well-defined interface, hence effectively encapsulating information and functionality pertaining to the same functional group at one place). It is easy to see that these modules are logically grouped into three layers. The database services layer provides persistent storage that is used by the higher-level components, like the navigator (which stores the state of processes) or the history manager (which logs process histories). The process services layer contains the main functionality of the PSS, like navigation, data handling, and monitoring services. The interface services layer is responsible for connecting to external systems in order to invoke applications, monitor their state, or access resources. The logical grouping of the modules into the three layers implies orthogonality, and in fact the modules of the layers are to large degrees independent of each other. This means that new database types can be connected in the database services layer without affecting the modules in the interface services layer, or that the interface to a particular system can change without impact on most of the service modules. The different layers are described in more detail in the following sections.
9.2.1. The database services layer

The persistent storage provided by the database services layer is logically divided into a number of spaces, each of which is reserved for a specific type of information. The template space stores process specifications and information about registered programs. Once a process is instantiated, its state is kept in the instance space in the form of a process image containing all information about it. The object space is used to store knowledge about external resources, such as files, database contents, or distributed objects which are needed by workflows. The history space is the key to process analysis functionality. The kernel’s logging components store all relevant information about running processes here. External analysis tools can access the logged information in order to compute aggregated information which can be used, for instance, to optimize process structures. The configuration space, finally, is used to store information about the current system configuration, including the users and programs currently connected, the programs running externally, and other relevant information whose persistent storage enables the system to set up operation again after a server crash.

The logical division of the persistent storage into multiple spaces provides a maximum level of flexibility, because it allows to implement and optimize each space separately. This is an important property given that the different types of information stored in the different spaces have extremely varying access patterns. To illustrate this point, consider the differences in access characteristics between the template space and the instance space. Due to the fact that process templates change...
rather infrequently (the time between modifications to a process template has to be measured in
days or weeks rather than minutes), the template space is mostly accessed when a new instance has
to be generated from a given template. Hence, the access characteristic is almost read-only. Fur¬
thermore, process starts are much less frequent than process state changes, which need read/write
access to the instance space. Hence, the access characteristics for the instance space are read/write,
and this very often.

These differences become important especially in a distributed setting with multiple servers
(like the cluster architecture described in the next section), since it provides the opportunity for
many optimizations. For instance, it is reasonable to use a centralized database as a common
template space for all servers, and provide each server with a local instance space in order to
optimize access to its frequently used state information. In a similar way, access to the other
spaces can be optimized.

The database abstraction layer provides a mapping function between the data representation
used in the upper system components (based on the Opera Canonical Representation described in
chapter 4) and the format used by the connected databases. Using an abstraction layer allows to
replace the underlying databases and is an important requirement for an adaptable system. Thanks
to the database abstraction layer, adding a new database for a specific space requires only the
writing of new mapping functions between the canonical representation and the database-specific
format. For the other system components, the nature of the databases actually used is completely
transparent.

9.2.2. The process services layer

The process services layer contains the modules pertaining to the core process management func¬
tionality. Each module is responsible for a specific group of services, and the various modules
communicate through well-defined interfaces. To add new functionality to the system, modules
can be added to the process services layer. The basic configuration of OPERA consists of the
following modules:

- **The Navigator** is responsible for enforcing the control and data flow between applications.
  It receives notifications of user requests and external events (such as the termination of pro¬
cesses), evaluates process states, and determines the next steps to take. Hence, the navigator
can be seen as an interpreter for the processes given in the canonical representation. At
the core of the navigator is the implementation of the process model itself (i. e., an object-orien¬
ted representation of the OCR elements). Adapting the process model requires in many
cases only modifications to the navigator, while other modules are not affected. At the same
time, the navigator has no knowledge of the nature and configuration of external systems,
which are always accessed through the dispatcher.

- **The Dispatcher** is responsible for the invocation and control of external programs. If a
  program has to be started, the navigator passes its description and the necessary data to the
dispatcher. The dispatcher will then determine the subsystem responsible for executing the
program and contact it to initiate the actual start of the program. The concept of subsystems
encapsulating the specifics of particular applications is discussed below.

- **The Transaction Manager** is responsible for the coordination of all activities inside a server.
  Conceptually, each state transition of a process is treated as a transaction in order to en¬
sure the consistency of the connected databases. The transaction manager has to ensure
atomic commit if multiple databases are connected to a server. It also has to synchronize
9. **Runtime Architecture**

parallel operations in a multithreaded server implementation. In the current implementation of OPERA, which is based on POSIX threads, the transaction manager synchronizes parallel threads in a trivial way by allowing only one thread to be active at a time. This is an acceptable solution given that (1) transactions are comparatively short and (2) implementing enhanced synchronization mechanisms that would allow for more concurrency would increase the administrative overhead.

- **The History Manager** is responsible for the logging of all relevant events to the history space. Because of the kernel characteristic, it is important that a history manager be highly customizable, allowing the definition of new event types whenever this should be necessary for a specific application. A concept for a history manager for the OPERA system has been developed and implemented in [Sei98]. We will not discuss its architecture in detail here.

- **The Exception Manager** is responsible for the handling of exceptions according to the mechanisms described in chapter 6. The exception manager is informed whenever an exception is detected in an external application, and invokes the necessary steps for resolving it by calling exception handlers, aborting tasks, and invoking compensating operations.

- **The Event Manager** implements the event mechanisms described in chapter 8. The range of its services encompasses the registration of events, the management of the publish/subscribe-mechanism for the communication between processes, and the enforcement of the correctness criteria described in chapter 8. The event manager interacts closely with the navigator (since its decisions have impact on the control flow of processes which import events) and the dispatcher (since enforcing atomicity may require to manipulate external applications).

### 9.2.3. The interface services layer

The interface services layer contains all functionality needed in order to connect to external applications. As a general rule, a PSS needs to integrate applications that are not PSS-aware, i.e., lack the necessary interfaces for connecting to the PSS components. Thus, the PSS itself has to provide the necessary functionality to invoke and control the applications. This is a fundamental difference to approaches such as DCE, CORBA, and DCOM and allows the easy integration of existing applications.

The interface services layer consists of *subsystems* that allow to control external applications. For each class of external applications, there has to be a dedicated subsystem encapsulating the algorithms and protocols for communicating with the remote system. A one-to-one relation exists between the *Program* class hierarchy of OCR (cf. section 4.2.2) and the subsystem hierarchy: If the dispatcher has to start or manipulate a program, it always contacts the appropriate subsystem, which is selected based on the particular program subclass. It is then left to the subsystem to interpret the information encapsulated in the program object and to take appropriate measures. There are two general approaches to the implementation of subsystems, depending on the way the communication with applications can be established:

- **Agent-based implementations** consist of two components, a *communication interface* which is part of the OPERA server, and an *agent* which is running on a remote system and does the actual work of controlling the programs. An example for this type of connection is the *UNIX* subsystem. Invoking a program on a remote UNIX host requires the existence of an agent running on this machine. This agent is notified by the communication interface when a new program has to be started, aborted, or compensated, and it informs the kernel of changes in the program's state (like a successful termination or abort).
• **API-based implementations** are used to communicate with systems which provide an appropriate *application programming interface (API)*. Examples for such systems are CORBA, SAP, or many workflow management systems. An API-based adapter needs no agent, but uses the API of the respective system to invoke applications and monitor their state. The development of API-based adapters for OPERA is described in [Sei98] (CORBA) and [Sch98] (SAP). We will not discuss them in detail here.

### 9.3. Example

![UML Sequence Diagram](image)

**Figure 9.2.:** UML Sequence Diagram: Interaction of OPERA components when a new process is instantiated

The interaction between the various components is illustrated in Fig. 9.2, which contains an UML *Sequence Diagram* describing the flow of control between the various modules when a new process is instantiated. At the top of the diagram are the components involved in the instantiation: The three system modules (instance space, template space, and navigator) and two application objects (a process template and a process instance). The users or external components interacting with the server are modeled as “outside world”. OPERA is based on an object-oriented system design, with all system components and all data created and passed between them modeled by classes. All interaction between objects is through method calls.

The arrows represent method invocations, and the order of invocation is top-down. Instantiating a process incorporates the following steps: (Note that we describe a successful execution only. If failures occur, they raise exceptions which lead to an abort of the current transaction.)

1. The `startProcess()` method of the navigator is invoked. It has to be passed the name of a process template to be used.

2. The appropriate template is requested from the template space through a call to the method `getTemplate`. If the template exists in the connected template database, it is returned by the call. In the diagram, the template is represented by the `temp` object right to the navigator.
3. Before an instance can be created, a new unique identifier has to be generated. The instance space provides a method, `getPID`, for this task. The rationale behind placing the ID generation in the instance space is that all IDs have to be unique within the instance space in order to ensure proper operation. Furthermore, many database systems provide functions for efficiently generating unique values (like Oracle’s `SEQUENCE` construct), which can be used for this task.

4. The identifier is passed to the `createInstance()` method of the process template object. Template objects in the OPERA design are self-contained in the sense that they not only encapsulate the information needed to generate new instances, but also the methods to do so. This allows for easy modifications if the semantics of the process model are to be changed.

5. Once a process object has been created, it is requested to store itself in the instance space (through the `store()` method).

6. Finally, the instances of the component tasks have to be stored in the instance space. To this end, the `storeTask()` method is invoked for each task. Tasks are stored as separate items in the instance space for performance reasons. During navigation, normally only some of a process’ tasks are needed by the kernel, hence retrieving a complete process image would waste time and bandwidth. In the OPERA design, the navigator will request form the instance space only those task instances which are currently needed.

7. Now the instantiation is completed and the process can be started. To this end, the process’ `start()` method is invoked which contains all logic for determining the first tasks to start. These tasks are subsequently passed to the dispatcher for actual invocation.

The example shows how the object-oriented design of the kernel allows to guarantee system adaptability. Implementing process instantiation as a method of the process template, for instance, allows to easily adapt this procedure if the process model should change. Likewise, since the process object contains the `start` method, it easy to modify the semantics of starting a process without affecting other system components. The overall objective of designing the architecture has been modularity, i.e., limiting the scope of modifications to a small number of components.

9.4. Extensibility

The possibility to extend the PS kernel by adding support for new types of applications is one of the most salient requirements. We have already described that an application type is represented by two main components: On the one hand, there has to be an OCR subclass of `Program` that describes the data to be passed to the external system in order to start an application. On the other hand, in the runtime system, this class has to be accompanied by an appropriate `subsystem` that is able to decode the information in the program objects. A general principle of the OPERA architecture is that all mechanisms for the communication with remote computers and systems are embedded in the subsystem, which is accessed by the other kernel components through a well-defined interface.

Table 9.1 describes the interface each subsystem has to provide to the other system components. The presentation here is slightly simplified compared to the actual implementation [Nus98]. The interface consists of methods for each request which may be sent by the dispatcher, enabling it to start applications or manipulate their state as required by the atomicity and exception handling
9.5. Distributed architecture: clusters of PSS servers

<table>
<thead>
<tr>
<th>Method name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>Activity, Program</td>
<td>Start an application</td>
</tr>
<tr>
<td>abort</td>
<td>Activity, Program</td>
<td>Request the abort of an activity</td>
</tr>
<tr>
<td>resume</td>
<td>Activity, Program, Exception</td>
<td>Resume the application after an exception was handled</td>
</tr>
<tr>
<td>compensate</td>
<td>Activity, Program</td>
<td>Compensate the application. Subsystem has to have the necessary knowledge on how to do this</td>
</tr>
<tr>
<td>release</td>
<td>Activity, Program</td>
<td>Release all locked resources. Important for applications whose commit can be deferred</td>
</tr>
<tr>
<td>recover</td>
<td>Activity, Program</td>
<td>Used to re-establish the communication with a subsystem after a server crash</td>
</tr>
</tbody>
</table>

Table 9.1.: Standard interface to be provided by subsystems

algorithms. If the system has to be extended to support a new class of applications, a subsystem has to be added which implements the interface described above and translates each call into appropriate interactions with the remote site. All methods are parameterized with the activity and program objects containing the necessary information to select the appropriate programs, pass the correct data, etc.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>started</td>
<td>Report successful start of program</td>
</tr>
<tr>
<td>finished</td>
<td>Report successful termination of program</td>
</tr>
<tr>
<td>failed</td>
<td>Signal the failure of a program</td>
</tr>
<tr>
<td>exception</td>
<td>Signal the occurrence of an exception</td>
</tr>
<tr>
<td>aborted</td>
<td>Confirm the successful abort of a program as requested by the system</td>
</tr>
<tr>
<td>resumed</td>
<td>Confirm the successful resumption of a program</td>
</tr>
<tr>
<td>compensated</td>
<td>Confirm the successful compensation of a program</td>
</tr>
<tr>
<td>released</td>
<td>Confirm the successful release of all locks held on behalf of a program</td>
</tr>
</tbody>
</table>

Table 9.2.: Callback interface available to subsystems

The subsystems communicate with the rest of the kernel through the dispatcher, which provides a set of callback methods allowing the subsystems to signal events of interest. These calls are shown in table 9.2, again slightly simplified compared with the real implementation [Nus98]. The callback methods can be categorized into two groups. They either are acknowledges to requests sent by the dispatcher (like the started or compensated callbacks) or are used to notify the kernel of a state change in an external application (like the finished or exception callbacks). A detailed description of the protocol to be used between the dispatcher and the subsystems can be found in [Nus98].

9.5. Distributed architecture: clusters of PSS servers

Up to now, the description of the OPERA system has quietly assumed a centralized architecture in which all processes are controlled by a single server. In systems that are large in terms of number of connected users, number of processes executing concurrently, and number of external applications running, a single server will soon become the main bottleneck. The obvious solution is to use multiple servers and distribute the workload among them [BMR96, MSKW96, MWW+98]. While this is today state of the art, several limitations can be found in current distributed PSS
9. Runtime Architecture

architectures:

- The distributed configuration is not transparent to clients: clients bind to specific servers, which means that nodes cannot be shut down without affecting connected users or programs.

- Processes are assigned to servers statically: any given process "lives" on a given server for its entire lifetime. This creates availability and scalability problems. If, for instance, a server fails, the execution of all processes assigned to it stops, which is not acceptable for mission-critical applications [KGAM96]. In addition, static assignment makes it impossible to move part of the workload of a busy server to an idle machine, thereby reducing the applicability of load balancing schemes.

- Maintenance of a PSS server and availability exclude each other, since all processes on the server need to be suspended for the duration of the maintenance phase.

A solution is to use multiple servers and distribute the workload among them. In this section, we will describe a cluster architecture for distributed process support systems which was developed to address the problems described above. The COPS (Clusters of OPera Servers) architecture aims at providing cluster properties for process support systems on top of a non-clustered platform. This concept of a PSS cluster is to some degree similar to workstation or PC clusters [Pfi95, BSS+95, VDB+98]: It consists of a collection of interconnected servers which cooperate in order to serve client requests. Clients, however, do not see the particular servers, but perceive the cluster as a logical unit to which requests are sent and from where messages are received. Because of this transparency, the cluster concept is a powerful basis for building fault-tolerant and scalable systems.

The challenge in constructing the COPS cluster was to preserve transparency in spite of the complex administrative procedures involved in providing cluster services. As an example, consider dynamic load balancing policies, which are necessary to make the best use of the resources available in the cluster. Under these policies, processes are migrated to idle servers when the load on a node reaches a certain threshold. Processes can be very complex and may have a large context in the form of open connections to running programs as well as users monitoring the process state. Hence, it is not sufficient to just copy the process state information to the new node. In addition, the open connections have to be moved as well, avoiding loss of information, for instance, in the case where a program finishes (and sends its results) during the migration phase. A second challenge besides these correctness issues is performance. Providing the cluster characteristics on the application layer leads to a certain overhead. An important question to answer is thus whether the performance degradation is small enough so as to make the approach feasible.

COPS does not require a workstation cluster as the underlying platform, but implements "cluster-like behavior" as part of the application layer. This has two main advantages.

1. Workstation clusters are not yet pervasive. They cannot be implemented using commodity hard- and software, but rely on specialized components and require modifications of the operating system. Low-cost approaches are promising, but very restrictive at the moment. An example should support this statement. Microsoft has recently released a clustered version of its Windows NT operating system, the Microsoft Cluster Server (MSCS) [VDB+98].

---

1It is important to note here that of the features provided by cluster architectures, we are primarily interested in transparent scalability and availability. If we wanted to provide high performance through parallelization, specialized hardware like high-performance switches would of course be necessary.
The idea is to provide workstation clusters through software, without the need for specialized hardware components. Several vendors, like Oracle, SAP, or Microsoft itself, offer application programs that exploit the cluster structure and can provide an increased degree of fault tolerance. A closer examination, however, shows that the functionality at present is very restrictive. A cluster can consist only of two computers, which furthermore have to be linked by a common SCSI bus to which a shared disk is connected. The cluster software supports only availability, but no parallelization of tasks [VDB+98]. These severe limitations indicate that the provision of a full-fledged cluster solution as part of the operating system is far from trivial.

2. A second reason is that workstation clusters are usually based on a homogeneous set of machines as the underlying platform (for example, only Linux machines in the Beowulf project [BSS+95], only NT servers in the case of MSCS), a requirement that is too strong for many enterprises investing in process support technology. Platform-independence is a key objective in the OPERA approach, since a platform-independent PSS allows to utilize existing resources whenever possible and reduces the investment effort for PSS deployment. This allows to optimally utilize the existing infrastructure, which is today often a mix of PC- and workstation-based machines running various operating systems.

*High availability* in a PSS cluster can be achieved by using backup mechanisms that allow to replicate the state of a process on multiple failure-independent servers, ensuring that in the case of a server crash at least one copy remains accessible. Such an approach was described as part of the EXOTICA project at IBM Almaden [KGAM96], but without an evaluation of the overhead that occurs because of the necessary synchronous replication.

*Scalability,* in its simplest form, is enabled by the fact that the structure of the cluster is transparent to clients, which allows to add more servers when needed. For several reasons, however, more sophisticated solutions are needed. There is especially the need to incorporate load-balancing mechanisms allowing processes to be transparently migrated from server to server. This is not straightforward since it has to be guaranteed that normal operation of the cluster is not affected during process migration.

---

**Figure 9.3.: A COPS cluster**

---

111
9. Implementing clusters

A PSS cluster integrates multiple servers and provides a suite of mechanisms and protocols that allow to foster scalability and fault-tolerance. In this section, we discuss the cluster architecture chosen for OPERA and discuss the added functionality provided by means of a number of distributed protocols. The principal idea behind the concept is shown in Fig. 9.3. It is based on multiple communicating servers and concentrators are used to connect external components. The concentrators make the cluster appear as a single server to the outside world.

9.6.1. Extending the server architecture

To enable interaction between distributed process servers, the architecture described before was extended by modules allowing the coordination with other servers. Fig. 9.4 shows the extended architecture, which now provides a cluster adapter implementing a number of services needed for cluster maintenance and distributed interactions. The cluster adapter consists of the following components:

- The cluster configuration manager (CCM) cooperates with other nodes in order to implement the cluster configuration service, i.e., it maintains a consistent view of the current system structure and allows to modify it in a controlled way. This requires keeping track of the nodes that are currently in the cluster and of the services that are available on each of
9.6. Implementing clusters

The process replication manager (PRM) implements a number of replication schemes which allow to store the state of a process redundantly in synchronous copies on multiple nodes. For processes that are replicated in this way, continuous availability can be guaranteed since as long as only one node fails, a copy of the current process state remains accessible. Each PRM maintains a persistent open hashtable in which state information about replicated processes is stored. This information is used by the backup server when a failure occurs. The backup/failover algorithms implemented in COPS are described in chapter 10.

- The Failover manager (FM) is closely related to the PRM. It is responsible for reconciling the distributed PSS after a node failure by using the replicated information. To this end, it determines the failed processes of which backup copies exist, locates those copies, and recovers the processes by installing the backup copies.

- The process migration manager is responsible for moving processes from one node to another. This requires not only to copy the process state, but also to update administrative information that is used by clients and applications to locate the process. The process migration mechanism has to ensure consistency, i.e., it must guarantee that no information sent to a process is lost due to a migration in progress.

- The load balancing manager is responsible for all issues pertaining to the equal distribution of the work in the system. Together, all load balancing managers implement a load balancing policy (see section 11.5) by relying on the services of the process migration manager.

9.6.2. Providing transparency

The structure of the cluster (i.e., the particular servers that are part of it at a certain point in time) has to be completely transparent to clients requesting PSS services and programs running as part of a process. Without this property, restructuring the cluster would be very complicated. The necessary transparency is ensured by using concentrators that serve as proxies isolating clients from servers. A client knows only the address of the concentrator through which it interacts with the cluster. The concentrator has always full knowledge of the current cluster structure, which allows it to route client requests to the appropriate nodes. Upon startup, it determines the current configuration (by sending a broadcast which is answered by all running servers). Subsequently, each configuration change (nodes joining or leaving the cluster) is broadcasted to all concentrators by the cluster configuration managers.

Each process is addressed by an identifier which is a tuple \((N, S)\), where \(N\) is the node where the process was started and \(S\) is a sequence number that is unique on that node. This addressing scheme does not only allow to generate system-wide unique addresses without synchronization, it also speeds up the routing of requests as long as no migrations or failovers have occurred. Since the server address is encoded in the process ID, a concentrator can propagate a client request directly.
9. **Runtime Architecture**

If a process is migrated to another node (or if it changes its location because of a failover after a server crash), the new node sends the information about the new location to all concentrators in the cluster. Each concentrator keeps a cache of these process/node assignments that is indexed by a hashtable and kept in main memory. This cache allows to quickly determine if a process has been migrated and to retrieve its new location. Information about migrated process is also stored persistently on each server in the system (as part of the configuration space). Upon startup, a concentrator initializes its cache by contacting an arbitrary server and requesting this information.

The concentrators serve as proxies not only for clients accessing servers, but also for servers contacting clients. If a server needs to contact a particular client (e.g., to send a new work task to a user or to query the current state of a program), it always sends its messages to a concentrator, which propagates them to the client. Servers do never store the actual addresses of clients, but keep a table of client/concentrator assignments. This allows a client to change its location without affecting the servers in the network, a feature that is important especially if clients are nomadic and change their address frequently. As long as they are always connecting through the same concentrator, servers are not affected.

### 9.6.3. Concentrators

It is obvious that the most salient characteristic of the PSS cluster is transparency, i.e., the impression of a single system for external components. In this section, we will have a closer look at the concentrators and the functionality they have to provide in order to maintain the illusion of a centralized system.

![Diagram of API-based communication with a remote system]

The main purpose of a concentrator, as described above, is to serve as a proxy for external agents, routing messages sent to the appropriate servers. An important aspect of this proxy property is that external agents should not even know that they connect to a concentrator instead of a proper PSS server – a requirement which becomes important especially when existing applications are connected through an API, since the APIs can usually not be changed. The usual way of communication through an API is shown in Fig. 9.5 for the example of connecting to an SAP system. The Subsystem adapter, which is part of the kernel, is linked against an API provided by SAP. This interface provides the necessary logic to communicate with a remote SAP installation.

The problem with this way of handling communication is that it is not possible to add a concentrator between the kernel and the remote system, since the communication is controlled by the API functions which cannot be changed. Hence, providing cluster properties for a system using API-based communication is only possible if the architecture is changed in an appropriate way. The solution is to put the subsystem adapters into the concentrators instead of the kernel [Sep98].

114
The resulting architecture is shown in Fig. 9.6. The subsystem adapters have been moved into the concentrator, and they communicate with the kernel through a dispatcher stub providing the standard dispatcher interface, but transferring all communication to the "real" dispatchers on the kernel site. It is important to note that such a dispatcher stub has connections to all dispatchers present in the system. The stub concepts allows to preserve the interface through which subsystem adapters communicate with the system. Hence, no modifications to the adapters are necessary.

9.7. Chapter summary

This chapter has focussed on two important aspects of the OPERA architecture. First, we have presented the general architecture concept for a process support system kernel, with a special focus on extensibility. The object-oriented structure of the design allows to easily extend the system, either by adding new modules that provide new functionality, or by modifying existing components. As an example, we have discussed how external systems can be added to an OPERA framework.

The distributed architecture described in the second part of the chapter forms the basis for advanced mechanisms that can guarantee availability and scalability of a process support system. The transparent cluster concept that has been implemented as part of the COPS version of OPERA is necessary in order to migrate processes arbitrarily between nodes, which is required for both backup and load balancing purposes. In the next chapters, we will discuss availability as well as scalability issues in detail.
9. Runtime Architecture

Seite Leer / Blank leaf
10. Continuous Process Availability

Fault tolerance in a PSS context comprises, in addition to the language aspects already discussed in previous chapters, fault tolerance of the runtime environment itself. In this regard, it is necessary to guarantee continuous availability of the system, i.e., ensure that the runtime services can be provided even if some of the system's components fail. The key to this property is the cluster architecture introduced in the last chapter, since it allows to store process information redundantly on failure-independent sites. In this chapter, we will introduce the problem of process availability and discuss possible solutions. A large part of the chapter is devoted to a performance study comparing different implementation strategies.

The chapter is structured as follows. Section 10.1 contains a motivation. Section 10.2 gives an overview of availability techniques. Section 10.3 describes how to implement backup mechanisms in a process support system. Section 10.4 gives an overview of the mechanisms that were implemented on top of the OPERA/COPS architecture and whose performance is evaluated in section 10.5. Section 10.6 discusses the results.

10.1. Motivation and problem description

The availability provided by traditional, centralized PSS architectures is far from being sufficient if mission-critical processes have to be controlled. In a centralized PSS architecture, all operation comes to a halt if the central process server fails. During the outage of the server, no navigation is possible, leading to the fact that all processes are blocked until the server is up again. For processes whose liveliness has to be guaranteed, this is not acceptable. Liveliness of a process is an essential requirement in a variety of cases, like processes for decision support in stock trading environments or real-time PSS applications with side-effects in the "real world".

As a general rule, the shorter the required answer time of a process compared to the mean time to repair of the system, the more important become availability issues. Hence, in a traditional workflow environment where manual activities tend to have a latency (time until a task is accepted by an employee) of several hours and comparatively long processing times, availability is not overly important given that the typical recovery time for a server will be in the order of minutes. On the other hand, consider a PSS for processing images sent by an earth observation satellite. The purpose of a process in such an environment can be to coordinate the transformation, storage, and analysis of the images which are sent at a constant rate. The single operations of the process will be rather short, but the arrival rate and the amount of information is very high. In this setting, blocking of the process support system will necessarily result in the loss of information, given that the buffer space available is limited and images continue arriving while the server is recovering.

While the centralized PSS architecture provides a basic level of fault tolerance (by storing the process images persistently in the instance space), this does not solve the availability problems, since it guarantees only that operations can continue after the server has recovered. For continuous availability even during server failures, some form of redundancy has to be introduced into the system, allowing to switch to a copy of the system and process state information in the case that a
Continuous Process Availability

Server fails.

Concepts for high availability in computing systems have been around for a long time, and some of them could serve as the base for implementing PSS availability:

- Perhaps the first general-purpose computing system with high availability was Tandem Corporation's NonStop Kernel, which combined a distributed hardware design with an operating system exploiting redundancy [Bar81]. It is one of the earliest examples of the process pair approach which maintains copies of running processes on a second, backup machine in order to allow a failover to these copies in the case of a failure.

- In database management systems, availability is achieved through backup mechanisms which mirror the contents of a database on a remote, failure-independent site [GR93]. These ideas could be applicable to process support systems since all relevant state information of the PSS is kept in a database.

- Another redundancy technique in database management systems is database replication, which aims at keeping redundant copies of data items in multiple databases [CP84]. The difference to backup mechanisms is that replication applies redundancy not only for availability but also for performance purposes, allowing to access multiple copies of a data item at the same time. Another difference is that replication mechanisms are usually more flexible, allowing to replicate selected data objects instead of the same database, and allowing to maintain an arbitrary number of copies.

A concept for high availability in the special context of a workflow management system has been developed in the EXOTICA project [KGAM96]. This approach combines the three techniques described above by implementing the process pair idea through database backup, where replication mechanisms are used to implement the backup mechanisms to achieve higher flexibility. The mechanism proposed there has a number of unique properties which make its application to a PSS context highly desirable. The description in [KGAM96] is, however, on a theoretical level and lacks information about practical experiences with the algorithms proposed. The main focus of this work in this regard is in an in-depth evaluation of the proposed process backup techniques. To this end, the algorithms have been implemented on top of the OPERA/COPS architecture, and the performance of several implementation options has been evaluated. The interesting questions are whether high availability of processes is implementable at an acceptable overhead, and which implementation technique is the most appropriate.

10.2. Techniques for Establishing Availability

In this section, we give a short overview of availability techniques which are of importance for process support systems.

10.2.1. Process Pairs

The concept of process pairs can be seen as a general approach towards high systems availability [GR93]. As an example, we will shortly describe the approach using an example, the Tandem NonStop kernel, which has been the first practical effort of a general purpose computer system providing high availability [Bar81]. As early as 1974, it combined a distributed hardware architecture with a message-based communication concept and a fault-tolerant operating system.
In the NonStop kernel, all operating system services are implemented as process pairs. In addition, the concept is available at the application level, allowing to build fault-tolerant applications. The idea is that both processes in a pair have the same state at all times, which allows the backup process to take over whenever the primary fails. Fig. 10.1 shows an example containing a process pair that controls a tape device. A client application, also running as process pair, wants to append information to the tape. All information in the kernel is through a request/reply message passing scheme, hence, in a first step, the primary client process sends a message containing the append request to the primary server process. This process sends a checkpoint message containing the request to its backup. Upon receiving the acknowledge, it starts executing the operation (i.e., appends information to the tape). After the successful operation, the state is again checkpointed to the backup, after which an acknowledge is sent to the client. Note that the client has as well to use a checkpointing scheme in order to ensure fault tolerance. The system does not enforce checkpointing at the client side. It is up to the application programmer to ensure correctness by synchronizing the process pair.

The checkpointing scheme allows to tolerate one failure per process pair. If, for instance, the primary server processor or its I/O channel fail, the backup will be notified by the system and can take over since it has up-to-date knowledge of the current request. One of the key goals of the mechanism is to ensure exactly once semantics. In the example, it has to be guaranteed that the information is appended to the tape only once, even if failures occur. It is easy to see that, as far as failures on the server side are concerned, this can be guaranteed. On the client side, there can be the problem of repeated requests. To solve this, request sequence numbers in conjunction with the re-sending of results can be used [Bar81].

The process pair concept shows the principle of achieving high availability in the general case. A key disadvantage of the approach is the high number of messages needed to synchronize primary and backup. The overhead caused by this is the main reason why the approach has found
10. Continuous Process Availability

![Diagram of 1-safe vs. 2-safe backup](image)

Figure 10.2.: 1-safe vs. 2-safe backup

no widespread acceptance. (Although it is still used, for instance in Tandem's NonStop servers, but only for specialized applications with extremely high availability requirements.) The situation is somewhat similar to the problem of providing atomicity, as pointed out in chapter 5. Like atomicity, process-pair approaches were not successful as a general operating system concept, but were adopted by database management systems, where the unit of checkpointing (transactions) is more coarse-grained, hence reducing the overhead. The according mechanisms will be discussed in the next section. It is also noteworthy that, recently, approaches have been taken to introduce high availability into commodity systems. The Microsoft Cluster Server (MSCS), which was already discussed in section 9, offers failover services, but on a very coarse level and without automatic checkpointing between the servers, which has to be done by applications.

10.2.2. Backup in Database Management Systems

Database management systems provide an abstraction, the transaction concept, that helps to reduce the performance problems that occur with the general checkpointing scheme described above. Also, for many types of applications, especially online transaction processing (OLTP), the correctness and continuous availability of the database is the main requirement. Hence, the process pair and checkpointing approaches have been adopted to databases and are frequently used. The main differences are that instead of a process pair, a pair of databases is used, and that checkpointing is performed on a per-transaction basis.

In the simplest form, a highly available database system consists of a primary and a backup database, where both have exactly the same physical structure, but only the primary is accessed by clients. After a transaction has been processed, the results, which are read from the primary's log, are sent to the backup database [GR93]. In practice, there are many possible variations of this basic scheme. The most important differences between implementations concern the degree of correctness to be guaranteed and the required performance of the recovery phase:

- **1-safe vs. 2-safe processing**: The degree of correctness of the backup mechanism can be controlled by choosing either a 1-safe or a 2-safe protocol (Fig. 10.2). With a 1-safe algorithm, the backup is updated asynchronously, while the primary commits locally. Naturally,
this can lead to the loss of information if the primary fails before all changes have been propagated. If the loss of transactions is not acceptable, a 2-safe algorithm has to be used, in which the changes at the primary are not committed before it receives an acknowledgment that they have reached the backup database. This variant ensures that primary and backup are always identical, but has the disadvantage of a longer delay until the transaction finished.

- **Hot standby vs. cold standby**: In the description of the backup algorithm above, we implicitly assumed a hot standby configuration, in which the backup applies changes it receives from the primary to the database instantly. This guarantees short recovery times, since after the failure of the primary, operation can continue with almost no delay. If a longer recovery phase can be tolerated, it is possible to use a cold standby approach, in which the changes sent to the backup are only appended to the log, but not applied to the database. Only in the case of an actual failover, the database is updated. (Note that it is possible to apply changes in the background when there is little load on the backup.) This mode reduces the answer time caused by 2-safe strategies, at the cost of a longer recovery time.

### 10.2.3. Database replication

Database replication concepts are based on storing multiple copies of a data object on different sites. This improves the performance of read operations in a spatially distributed network since applications can access the nearest copy, avoiding the communication overhead of reading a remote centralized database. It also improves availability because the failure of multiple databases can be tolerated as long as one copy remains available.

Of this basic principle, many variations exist which differ mainly in the way they maintain the mutual consistency of a given set of copies, the degree of correctness they can guarantee, the types of failures which can be tolerated, and optimizations which are introduced to speed up either read or write accesses. A detailed discussion of these algorithms is beyond the scope of this dissertation. An interested reader may refer to the large body of research in this area and consult [BHG87, CHKS92, CP92, DGMS85] for basic information and [PL91, LKR94, GHOS96, KA98, KPAS99] for recent research work in this area. What is interesting for the purpose of this work is that, like database backup functionality, a replication mechanism could be used as the foundation for highly available processes. This is especially interesting since the PSS architecture relies on databases for persistent storage anyway, and hence an utilization of the replication mechanisms provided by these databases is a straightforward approach. Actually, implementing a backup strategy for processes based on database replication is one of the options we have evaluated, and hence it is important to understand those aspects of replication algorithms that are necessary for the understanding of the evaluation presented later in the chapter.

There are two basic categories for classifying replication algorithms:

- **Symmetric vs. asymmetric replication**: This category distinguishes between schemes allowing to update every copy of a data item and approaches using a primary copy approach in which only one copy can be changed and the changes are subsequently propagated to the other replicas (Fig. Fig. 10.3). From a practical perspective, it is easier to implement a primary copy strategy, since a symmetric scheme requires complicated distributed locking mechanisms to ensure global correctness. A symmetric scheme, on the other hand, can be more efficient because the primary database of asymmetric schemes is likely to become a bottleneck if the load increases.
10. Continuous Process Availability

- **Asynchronous vs. synchronous replication**: This category distinguishes the level of correctness that can be guaranteed by the replication facility. With synchronous replication, modifications to a copy are applied to the other copies as part of the modifying transaction. A 2-phase-commit is necessary to ensure that all copies are updated atomically, and hence, this is a very time-consuming algorithm. This is the reason why many commercial DBMS resort to asynchronous replication mechanisms which resemble 1-safe backup strategies in the sense that the copies are updated separately from the changing transaction. The increased performance is bought, however, at the price of possible inconsistencies if transactions access out-of-date copies in an inappropriate way.

10.3. Backup Algorithms for Process Support Systems

The description of the different approaches above shows that availability can generally be achieved by keeping (one or more) spare copies of the critical items. In a process support system, these critical items are processes. The obvious solution is thus to adapt the process pair concept approach and introduce suitable checkpointing protocols that keep the multiple copies synchronized.

In implementing the checkpointing protocol, it is possible to exploit the fact that the process state resides in the instance space. With other words, all necessary information about the process is stored in a database, and the problem of synchronizing the process pair can be reduced to keeping copies of the process state mutually consistent. (It is important to note here that this implies that suitable cluster mechanisms are in effect which hide the actual location of processes from external applications. We assume that appropriate functionality, like it is present in the COPS architecture, is existing.)

In this section, we describe possible implementation strategies for a process backup mechanism. A general requirement in doing so is flexibility. Hence, the trivial solution of using a dedicated backup is often not feasible because of the high investment cost for doing so. Instead, we concentrate our discussion on more flexible schemes that allow to use the same database as the primary site for some processes and as the backup for others. This allows to optimally utilize the available resources.

We will present two principal variants, which have also been implemented and will be subject to a performance evaluation at the end of this chapter. The first variants relies on the replication facilities of the underlying database management system. The second variant implements the
backup mechanism as part of the application layer.

10.3.1. Database replication as the basis for backup mechanisms

We have already argued that backup and replication techniques are fairly similar. In principle, it is possible to use replication techniques to implement a backup strategy. In practice, the state of the art in database replication does not really allow it. The reason is that most DBMS vendors implement only asynchronous and asymmetric replication mechanisms [GHOS96]. This approach can only be used to implement 1-safe strategies, but these are not acceptable for a PSS context, because they can lead to the loss of information.

From a practical standpoint, and to our knowledge, only Oracle’s Advanced (Symmetric) Replication mechanisms [Ora95, Ora97] provide enough flexibility to use replication as the basis for a backup. Oracle8 Server offers various replication techniques, of which only one is really suited to the purposes of backup: Synchronous multimaster replication, where all copies of a data item are fully updatable and changes are propagated to all sites within the same transaction.

The unit of replication in this scheme is a master group, a collection of base tables that share the same replication schema. The database administrator can create master groups using a graphical user interface or a set of predefined procedures, which also allow to control where a given group is replicated. The propagation of changes is implemented using stored procedures which are installed on all database servers participating in the replication. These packages are generated for each master group when the replication schema is defined. The replication algorithms themselves are tuple-based. Every time a row of a replicated table is changed, a trigger invokes the appropriate procedure in the table’s propagation package. In contrast to Oracle7, Oracle8 uses internal triggers that are not part of the DML and are optimized for performance.

Process backup can be implemented using Oracle’s multimaster replication in the following way:

- A master group is defined that contains the tables containing the process data.
- The master group is replicated between two server’s databases. Because of the symmetric nature of the replication mechanism, both servers have full access to all processes.
- The synchronous replication mechanism ensures that the processes are mutually consistent at all times. For the servers, the replication is transparent, and no modifications to their code are necessary.

Obviously, this approach has potential disadvantages. First, it is very inflexible. Once one of the replication sites becomes unavailable, the remaining DBMS allow no updates to the replicated tables until the failure is repaired or the replication scheme changed. This, however, requires removing the failed database from the set of replication sites and adding a new node that acts as new backup. The process of restructuring the replication scheme is complex since the stored procedures have to be modified and the new site has to be brought up-to-date by sending all data. This procedure consumes a considerable amount of time, as we will later see in the presentation of the actual measurements.

A second disadvantage of this approach is that it does not scale well. If a cluster consists of more than two servers, the nature of the replication mechanisms requires that all process are replicated between all the nodes, increasing the delay until all copies are synchronized.
10. Continuous Process Availability

10.3.2. Semantic backup

Semantic backup refers to the fact that the entities that are subject to the backup algorithms are objects of the application level rather than database tuples or tables. In the case of a process support system, this abstract objects are the entities constituting a process, like the state of a process and its tasks, the data value passed to and returned from task invocations, and additional configuration information, like the resources allocated for a task or the site that was chosen to execute a program.

Using semantic backup in process support contexts was proposed by the EXOTICA project [KGAM96], where concepts for high availability of workflow management systems were discussed. The solutions developed there apply to the general case of process support systems, too. They are based on the principle that at the end of each navigational step at a primary server, information about changed entities is wrapped in a so-called change buffer and sent to the backup server for storage. At the backup server, the change buffer is decoded and its contents are converted into the local database format. Hence, the advantage of this approach is database independence: Replication can take place between databases with different conceptual schemas or even different data models.

Using semantic backup provides a degree of flexibility that allows to implement a number of optimizations which would not be possible with the more inflexible approach described before:

- It is possible to select the replication schema (i.e., the location of primary and backup server) on a per-process basis. This allows to distribute the system load evenly over all servers and is an important contribution towards optimal resource utilization.

- It is possible to provide different degrees of availability by deciding for each process if it should be subject to a hot standby or cold standby algorithm or if it should not be duplicated at all. In [KGAM96], the concept of availability levels was introduced for this purpose (Fig. 10.4): Each process can be categorized either as normal (no backup), important (cold-standby backup), or critical (hot-standby backup). This allows to trade performance for availability. A critical process will be available almost instantaneously after a server failure, but the hot standby requires an increased runtime overhead. An important process, on the other hand, needs to be installed in the backup database at recovery time before it becomes accessible, but it has shorter answer times at runtime. The different levels give users the possibility to deliberately choose the degree of availability needed, while considering system throughput and runtime behavior of process instances.

- The perhaps most important advantage is the aforementioned database independence. Not
10.4. An overview of the implemented mechanisms

In order to compare the performance of various replication strategies, we have implemented them on top of the OPERA/COPS architecture and evaluated them under various settings. The performance results are presented in the next section. In this section, we give an overview of the implementation.

10.4.1. Semantic Backup

The backup mechanisms implemented on top of the OPERA/COPS architecture use the Opera Canonical Representation, OCR, as the basis for replication mechanisms that ensure the consistency of multiple copies of a processes. On the server, every update to a process is encapsulated in a transaction, which executes what is usually referred to as a navigation step in the process. The transaction is controlled by the transaction manager, which initiates the replication algorithms at the end of a transaction. It was already explained that semantic backup is performed by exchanging between primary and secondary information casted in terms of the internal representation (instead of pages or database tuples). In our implementation, at the end of each navigation step and as part of the same transaction, all changes to the process state are saved in a change buffer at the primary and sent to the backup server for further treatment.

In order to improve the performance of the replication protocols in practice, a number of improvements have been applied to the algorithms:

- Since 2-safe is used, the backup mechanisms we propose could suffer from the high overhead of 2PC. We can alleviate this cost by taking advantage of the fact that only two sites are participating in each distributed transaction. We use a “degenerated” form of linear two-phase commit [BHG87] in which the primary sends the change buffer to the backup, the backup commits locally and then sends an acknowledgment back to the primary, which commits. The decision for the commit of the distributed transaction is taken at the moment the backup transaction commits. Correctness is guaranteed since if the primary fails after the backup’s decision, but before its own commit, the backup will take over navigation anyway and continue with the valid process state. If the backup does not take over (this can be the case when the down-time of the primary is too short), the primary has to inquire about the process state during recovery. To this end, each server uses a reconfiguration protocol at startup during which it determines the actual state of all processes that are registered in its database by querying the backup servers. If a backup server has taken over a process, it can be removed from the old primary’s database since the new primary has elected a new backup during fail-over. If the backup has not yet taken over, it simply sends the actual state of the process to the primary which updates the process image and continues navigation.

- Obviously, the change buffers play a crucial role in the performance of the backup system since they are exchanged at the end of each navigation step and stored at both the primary and backup. In the algorithms of [KGAM96], the change buffers are stored in persistent hash tables on both the primary and the backup. This adds significant, unnecessary overhead in practice. In our implementation, the change buffers are kept directly in the database, thereby simplifying the backup mechanism. Our experiments revealed that
storing the change buffers in the databases requires only between 40 and 80 ms per buffer, depending on the database size. In order to speed up the time needed to store and retrieve the buffers, a separate tablespace with a very large extent size (1 GB) was used, thereby reducing the overhead of frequently allocating new extents. The change buffers were stored as character arrays with a maximum size of 4 KB. The average size of the change buffers in our experiments was 300 Byte (significantly smaller than the information exchanged by database replication strategies).

- To increase the performance of the backup system even further, it is possible to execute the navigation steps partially in parallel. We do so by using multiple threads to speed up the processing of navigation steps. Fig. 10.5 shows the structure of a transaction as it occurs when the hot standby mechanism is used. First, the process image is fetched from the database. It is cached in main memory for the duration of the transaction. Navigation takes then place over the cached information, which will also act as the change buffer. After navigation is completed, the transaction forks into two threads. The first thread applies the changes to the process image at the primary, while the second sends the change buffer to the backup. At the backup a transaction is started that applies the change buffer to the process image. After its commit an acknowledgment is sent to the primary, and then the primary commits.

10.4.2. Backup based on database replication

For comparison purposes as an alternative to the semantic backup, we have implemented a solution based on Oracle's multimaster replication. It is important to note that this solution, although it provides a hot-standby algorithm from a technical perspective (since all changes are directly applied to the backup database), is in practice only comparable to cold-standby backup because of the complexity of the recovery phase. In the current Oracle implementation, changing the replication schema is mostly a manual procedure and takes a considerable amount of time. This aspect has to be taken into account when the performance results are discussed.
10.5. Experimental results

10.5.1. System Configuration

The configurations used, a cluster of workstations of different types connected by a standard 10 Mbit Ethernet, resemble many real environments. The concrete machines involved in the experiment are listed in Table 10.7, and an overview of the configurations used is given in Fig. 10.6. In most experiments we used configuration A, in which the Sun4 workstations were running the workflow engines and the PCs were used as database servers running Oracle8 DBMS. Servers and clients were located on Sun4 workstations in the standard experiments. In order to determine the impact of communication overhead, in some experiments we used configuration B, where database and server are located on the same workstation. Here we used the Sun10 workstations for databases (Oracle8) and servers. Finally, for the evaluation of mixed environments consisting of relational and object-oriented databases, we used configuration C with an ObjectStore database located on a Sparc10 server, and Oracle8 on a PC.

In all experiments, the same process was used, which consisted of 10 activities and had a process image size of about 50 KB. The structure of the workflow is shown in Fig. 10.8. The tree-like form was chosen in order to study the impact of process structure on the performance. We expected the time for navigation to be the longer the more successors a task has. Our measurements revealed, however, that there are no significant differences. Because of this, in the results below, only average times over all tasks types are given.

For performance reasons, the clustering facilities of Oracle8 were used in order to group together related tuples. All significant tables were grouped in one hashed cluster with the instance ID as cluster key, ensuring that the components of each process image are stored next to each other. The benefit is that, as long as all objects for one cluster key fit on a page, a database object can be retrieved from the database server with only one page access.

10.5.2. Behavior without backup

As base line, we use the performance measurements for normal processes using configuration A. To study the impact of the database size on navigation performance, the measurements were made with a varying number of concurrent process instances which resulted in different database sizes. The process images had an average size of 50 KB, which lead to database sizes between 500 KB and 50 MB.
<table>
<thead>
<tr>
<th>Machine type</th>
<th>Processor</th>
<th>Main memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARCstation 10</td>
<td>Model 30 SuperSparc</td>
<td>96 MB</td>
</tr>
<tr>
<td>SPARCstation 4</td>
<td>Sparc 60 Mhz</td>
<td>64 MB</td>
</tr>
<tr>
<td>Dell GXM 166</td>
<td>Intel Pentium 166 Mhz</td>
<td>32 MB</td>
</tr>
</tbody>
</table>

Figure 10.7.: The machine types used in the experiments

Fig. 10.9 gives the average execution time for various types of transactions occurring during navigation. We distinguish between three transaction types used in the servers:

- **Process Instantiation (INST)** involves creating a new copy of a process template and identifying those tasks that are immediately executable. In practice, this is the least frequently executed operation.

- **Activity start (START)** is called whenever the server is notified that an activity has started. It is the simplest transaction type, since it requires only the change of the activity’s state in the process image.

- **Activity termination (TERM)** comprises updating the terminating activity’s state and performing navigation (evaluating the metaprogram in order to identify the activities that have become executable), and updating the states of the executable activities.

The results show that, beyond a certain size of the database, the time per transaction stays almost constant. An explanation for this is that databases with sizes below this threshold fit in main memory, which reduces the access time on the database server.

Comparing the transaction types, we find that the most expensive operation is process instantiation, which is not surprising given that it involves both reading a process specification from the database and creating a new process instance. Compared with the cost for this operation, the differences between task termination and task start are not very large, although far more objects are modified by the task termination operation.

Fig. 10.10 shows a more detailed analysis of the cost per transaction. Most time is spent during the initial reading of the process image from the database. The time for updating the process image is much shorter because, on the one hand, only the changed components are updated and, on the other hand, the process image is most likely to be still in the database buffer because of the recent read operation that started the transaction. The impact of the read operation becomes even higher when the number of concurrent processes grows, as is shown in the left part of the figure, where we give the distribution of cost for a database with 1000 processes. Again, the cost of writing back the
10.5. Experimental results

Changes does not increase in the same proportion because of the buffered image. Because of the dominant impact of read operations, an obvious strategy for further optimization is the deployment of a larger cache for process images in the engine.

10.5.3. Backup based on DBMS replication

The backup mechanism implemented on top of Oracle's replication, as explained above, lead to an overhead, with respect to the base line, of approximately 200 % for process instantiation (because new database objects have to be inserted in the remote database) and an encouraging 30 % for the much more frequent navigation transactions (figure Fig. 10.11). These response times are acceptable in practice, especially if we take into account the absolute numbers, which stay below 4 seconds for process instantiation and take less than 0.2 seconds for navigation.

A strange behavior was observed, however, in the case of process termination, where the process image has to be removed from the database. The cost of removing one instance grew constantly with the database size, with an average duration of 15 minutes when 1000 concurrent processes existed. The reason for this can partly be explained by Oracle's row-based change
propagation, where each changed tuple is propagated separately to the remote sites. This does not, however, explain the reason for the significant overhead. We believe that this is an implementation bug that will be solved in future versions of the DBMS (we used the very early version 8.0.3). This particular point aside, we believe the other results to be representative. The results show, however, that Oracle's synchronous replication mechanisms need to be improved.

From a practical standpoint, the current behavior creates significant problems in real scenarios. Process support systems may execute a large number of processes. Although current systems tend to keep the data for completed processes in the database, the trend is to remove these processes from the on-line database and store them in a data warehouse for analysis. This is one of the reasons why the OPERA architecture uses separate storage spaces. In particular, the history space plays the role of data warehouse for completed processes so that this data can be manipulated and pre-processed without interfering with the on-line operations of processes being executed. In our system, we automatically remove a terminated process from the instance space into the history space by copying it (if change buffers are available, it will use them to avoid having to access the primary) and then deleting it from the instance space. Obviously, this operation can become quite expensive if the problem persists.

10.5.4. Semantic replication, cold standby

To mirror actual working conditions, all experiments were conducted with all servers acting both as primary and backup, with processes evenly distributed between the servers. In cold standby configurations, the biggest source of overhead is sending the change buffer to the backup and wait until it is persistently stored in the database. Fig. 10.12 shows the performance of process instance replication for cold standby mode. The results show that the cold standby mechanism leads to an overhead, with respect to the base line, that is around 65% for small databases and falls to 40% if the database size grows. Although this overhead may seem comparatively large, in practice it is almost negligible since it represents an overhead of between 0.5 and 0.3 seconds for starting a process and less than 0.3 seconds for the other two operations. In systems where there is human interaction involved, this delay will be easily masked by artifacts such as the terminal response time. For scientific applications this overhead is more significant but still within acceptable bounds, especially when considering the advantages of having a backup strategy for both fault-tolerance and process migration.

The differences between small and large databases arise from the fact that locating the process
images at the primary gets increasingly expensive as the database grows while the cost for storing the change buffers stays constant. To analyze in more detail which operations are performed and their contribution to the overall overhead, Fig. 10.13.a shows the relative cost of the operations involved in the transaction for a large database with 1000 processes. In the case of cold standby, reading the object takes as much time as in the non-replicated case. The overhead is created by the operations related to the change buffer. This implies that with a more efficient mechanism for change buffer storage the cost per transaction can be further reduced. It is questionable, however, whether the performance gain warrants the added complexity. In any case, given the results shown, cold standby as currently implemented seems to be a viable backup solution.

10.5.5. Semantic replication, hot standby

The performance measures for the hot standby mechanism are shown in Fig. 10.14. Due to the fact that both sites need to be synchronized and the change buffer needs to be applied at the backup, the overhead is larger than in the cold standby approach. It is not, however, significantly larger: for starting and terminating activities it is around 90% for the large database. Again, a comparatively large figure, but still acceptable since the actual time required is about 0.6 seconds. As before, this lapse of time is easily masked in many applications because there are additional operations like the communication with the clients, actualization of graphical user interfaces, transport of data between machines, and start of applications, that are quite expensive and are likely to be the real source of delays in practice.

The contribution of individual operations to the overall overhead is shown in Fig. 10.13.b. Note that even with the optimization of applying the changes at the backup in parallel, there is still the overhead of storing change buffers twice. In the case of starting a process, the overhead is significant and it does not seem to be possible to reduce it further. Since in a real application it is the user who determines which processes are critical and which important, the overhead can be assumed as part of the cost of having a hot standby configuration, but a large number of critical processes is likely to result in performance problems. Also note that we are still within the few seconds bound, which is not a dramatic increase in delay.
10. Continuous Process Availability

![Cost analysis for a Terminate Activity step and 1000 concurrent processes: (a) cold standby, (b) hot standby](image)

10.5.6. Time to recover from failures

The time to recover from server failure is shown in Fig. 10.15. The average time for recovering one process was between 1.2 and 1.3 seconds. This means that, for a large database with 1000 processes, the overall recovery time is about 19 minutes, which leads to an average unavailability of 9.5 minutes since processes are available as soon as they are recovered. The main cost factor is the installation of the process image in the database, which requires re-executing all original transactions by interpreting the change buffers. While the time for recovery may seem very long, it is within the bounds that are tolerable in business process environments where activities are very long. Moreover, it is similar to the recovery cost in database systems. The performance can be further improved by recovering multiple processes in parallel. Since no data conflicts between process images exist, the degree of parallelism for process recovery is bound only by processor performance and disk bandwidth.

The recovery cost for critical processes is dependent on the degree of fault tolerance that a process has to provide after a fail-over. If the process has to stay critical, the recovery time is as long as for important processes, since a new backup has to be elected and the process has to be installed in its database. Because of this, normally process degradation is applied, i.e., the process' availability is reduced every time a failure occurs. If it is reduced to important, during failover only change buffers have to be stored the backup, which leads to a considerable improvement of the recovery time while preserving the process's availability. It is possible to promote a process to critical level again dynamically at a later point in time. If the probability of server crashes is moderate, critical processes can be downgraded to the normal availability level, which means that they will block if their new primary should fail. By using this scheme, the recovery cost is further reduced. We give the recovery cost for both variants of degradation in Fig. 10.15. Note that, again, we did not apply parallel recovery of multiple processes. We plan to integrate mechanisms for parallel application of change buffers as a future optimization. The performance of the recovery using degradation to normal availability is especially promising for process migration facilities. Migrating a process involves creating a backup copy at the new location and then making this copy the working copy. With the performance obtained in our experiments, this seems like a feasible approach.
10.5. Experimental results

Average time per transaction in s

Figure 10.14.: The performance of hot standby using semantic replication

10.5.7. Impact of server location

In the experiments described so far, the database servers were located on remote machines and each database access had the additional cost of communication between the PSS server and the database. Since the database accesses are by far the dominant factor in the cost of a transaction, it could be assumed that by reducing communication cost the overall overhead of replication would be reduced too. To investigate the impact of communication on the overall performance, experiments were made with a modified configuration that placed the process engines and their databases on the same machine (Fig. 10.6B). With this configuration, PSS and database communicate over the much faster shared memory.

The results of these experiments (Fig. 10.16) show, however, that the gain in performance is very small, mainly because DBMS and PSS have to share the same processor. (Note that the absolute numbers shown cannot be compared to the results of the other experiments because the SUN10 servers used here are slower than the PCs employed in the previous measurements.) This small impact of communication overhead, even when using comparatively slow Ethernet connections, implies that the general deployment of a three-tier architecture with database and server residing on different nodes is feasible from a performance point of view.

This allows to extend the PSS architecture, without performance penalties, to more general schemes where a server uses multiple, distributed databases in order to achieve a higher processing capacity and to further increase the resilience to failures. In such an environment, a process engine is operational even if some of its databases fail. While in the case of a database failure, some processes will have to be taken over by the backup, the other processes can continue navigation and the server stays accessible for the start of new process instances which are stored in the remaining databases. Another promising option is the shared use of databases by multiple servers. This allows, if a PSS server fails, for quick fail-over by starting a new server process on a running machine and connecting it to the still accessible database of the failed server.
10. Continuous Process Availability

![Figure 10.15.: Time to recover from server failures depending on process type](image)

10.5.8. Heterogeneous environments

An important goal of the OPERA approach is the ability to integrate all pre-existing systems. Today, most specialized PSS require a specific DBMS that has to be bought and installed in addition to the process environment itself. One of our goals in the overall design was to avoid such restrictions, and database-independence is one of the main advantages of the semantic replication concept.

In order to study the behavior of the system when heterogeneous databases are used, we configured a system (Fig. 10.6.C) with servers using both an object-oriented database engine (ObjectStore) and a relational engine (Oracle8). The servers worked according to the usual scheme, with the object-oriented databases acting as backup for the relational ones and vice-versa. The results for the cold-standby backup algorithms (Fig. 10.17) show that the performance of the ObjectStore-based server (a) was slightly better than that of the relational one (b), due to the more efficient caching mechanisms it provides. ObjectStore has a page-server architecture where a cache manager process at each client machine maintains a pool of cached database pages. This improves performance especially for applications where no concurrent access from multiple sites takes place. In contrast to this, in the Oracle-based implementation, each process has to be fetched from the server prior to navigation. We expect that with the implementation of a client-side cache the performance of the systems will become similar. Note that the performance of the Oracle-based server was better here than in the homogeneous environment because of the faster application of the change buffers at the (ObjectStore-based) backup.

Our experiments with the heterogeneous environment are encouraging and show that database independence can be achieved with acceptable overheads. Another important point is that semantic replication of the type used in our backup strategies provides an efficient way of introducing replication mechanisms into systems that have no built-in replication support (like ObjectStore) or only offer asynchronous replication (like most commercial relational DBMS).
10.6. Discussion

The implementation of process backup mechanisms and their subsequent evaluation has revealed that providing continuous availability in a process support system can be achieved with an acceptable overhead. Hence, availability must be seen as an important "execution guarantee" to be provided in PSS applications that control mission-critical applications. An important result of our measurements is that, independent of the chosen backup protocol, the cost for synchronizing the backup copy has an upper bound, even for the most costly operation and large database, of 0.7 seconds. This indicates that for those PSS applications where the processing time of the single activities exceeds this margin (and this is the case in most potential application areas of process support), the overhead is negligible.

Moreover, the selection of the appropriate backup protocol allows to control the degree of overhead incurring. To discuss this aspect, Figures 10.18.A, 10.18.B, and 10.18.C show the results for the three different operations in the four approaches evaluated: no backup, database

---

**Figure 10.16.:** Impact of database location on performance

**Figure 10.17.:** Heterogeneous environment: (a) Oracle-based server and (b) ObjectStore-based server
replication, cold standby, and hot standby. The first conclusion to draw from the comparison is that the differences between semantic backup and backup strategies implemented over database replication are not very large. Given the added flexibility, this allows to conclude that semantic backup a good choice in a PSS whenever properties like evenly load distribution and fine-grained control over replication schemata is an important criterion. Furthermore, as discussed above, the database approach requires manual intervention in case of failures in order to switch from the primary to the backup. This, in many cases, renders the approach unfeasible in practice. In addition, database strategies only work if the database platform is homogeneous, that is, the same database is used as primary and backup. In some cases, it is even required that both primary and backup run on the same operating system.

If failures are unlikely to occur, runtime performance is the key requirement, and the necessary infrastructure (homogeneous database platform) is available, the implementation on top of a database replication mechanism could be chosen. This has the mentioned disadvantages, especially of the high recovery cost, but it limits the runtime overhead to only 30 percent. An important aspect to note is that using DBMS replication mechanisms requires no modifications to the PSS servers. Hence, since current commercial PSS do not, in general, incorporate backup mechanisms, it is possible to use the techniques described in this paper to extend their functionality with a backup service in a transparent manner and without degrading their performance.

The principles presented can also be seen in a wider context. For instance, as a way to provide backup or replication services in environments where the database engine does not provide such services. Also, the access pattern observed in a PSS is quite different from that considered in tra-
ditional database applications. Neither OLTP (short transactions, small set of changes) nor OLAP (long transactions, mainly read operations) characterize the access patterns of a process support system, which could be characterized as "online object manipulation", given that the transactions observed manipulate a large number of objects. The work in this paper can thus be seen as a first step towards gaining a better understanding of such applications and, ultimately, establishing a new class of benchmarks.

Finally, it has to be noted that the semantic backup mechanisms with their high flexibility can be used as the basis for process migration facilities which allow to change the location of processes, either for load balancing purposes or if a server has to be shut down for maintenance purposes. This is not possible with the DBMS-based replication approach, but it can be achieved with relatively little overhead on top of semantic replication. This aspect is discussed in the next chapter.
10. Continuous Process Availability
11. From Process Replication to Process Migration

The cluster architecture together with the backup facility described in the last chapters can be used as the basis for process migration mechanisms allowing to move processes between servers while they are executing. Such mechanisms are important prerequisites for effective load balancing in distributed PSS and can thus be seen as indispensable building blocks for scalable process support environments. The ability to migrate processes is also an important contribution towards system maintainability, since it allows to redistribute the workload of a running server to other machines in the case of planned outages for maintenance reasons. Without this ability, either all processes on the server would be blocked (a contradiction to the availability requirement) during the maintenance period. In this chapter, we will discuss process migration mechanisms for a PSS system. What is interesting from a research point of view is how the process replication mechanism developed for the OPERA system can be extended towards a process migration mechanism. Hence, the focus of this chapter is on the discussion of the respective extensions and on a general description of the components necessary in the kernel if load-balancing mechanisms have to be added.

The chapter is organized as follows. Section 11.1 contains a motivation for the need of migration mechanisms in scalable process support systems. Section 11.2 describes a first, naive, algorithm. Section 11.3 presents a more sophisticated algorithm, whose properties are discussed in section 11.6. In section 11.5, we discuss the problem of load balancing policies in a PSS, and section 11.6 contains a concluding discussion.

11.1. Motivation: Load balancing in a PSS context

Load balancing mechanisms in distributed systems aim at optimally using the available resources in order to minimize answer times and maximize resource utilization. The usual taxonomy distinguishes between static and dynamic load balancing mechanisms [CK88], where both strategies aim at optimally placing the processes to be executed on a set of available processors:

- **Static mechanisms** make process placement decisions only at the time of instantiation, i.e., before the process is actually started. To enable proper resource utilization, they require an a-priori knowledge of the load that the new process will generate, as well as general information about the current total mix of all processes in the system. Based on this data an execution site for the process is selected, and the better the predictions have been, the better is the distribution of load in the system.

- **Dynamic mechanisms**, in contrast, do not require a priori knowledge, but rely on the ability to dynamically change process assignments. This requires process migration facilities, i.e., mechanisms allowing to move running processes to a new site when the load on a particular processor reaches certain thresholds. Implementing process migration facilities in the general case is very complicated because the complete context of a process has to be
migrated, which includes including virtual memory contents, open files, message buffers, register contents, and environment information [Ras95].

In a process support system, only dynamic load balancing methods make sense because it is hard to predict the load characteristics of a process. In principle, with a logging and analysis component in place, it would be possible to estimate the load of a newly started process based on statistical information about previously executed instances of the same type. In practice, this is likely to be complicated. The first problem is to find a suitable measure for the load caused by a process. To understand this, it is important to consider the way in which process are executed. Most of the time, a process does not cause load on the process server, but on the remote systems on which the activities are executed. It is only during a navigation phase that a server and its resources are used. Hence, since there is no constant server utilization, it is nearly impossible to give a meaningful number for the load caused by a process. A dynamic mechanism that can react if an overloading is actually detected is will be much more suitable.

A second problem is the high amount of human interaction which occurs, for example, in workflow or software process environments. It makes the estimation of process runtimes very complicated, given that the execution times for human activities may vary widely. Third, due to the long duration of workflow processes, it is not likely that the system configuration will be stable for the whole lifetime of a process. Hence, it may be necessary to move a process during its execution time just because its server has to be stopped. Given that, in certain applications, processes will run for days or weeks, this is not unlikely to happen.

11.2. A naive algorithm

Fortunately, implementing process migration in a PSS is simplified by the fact that all process state information is represented in an abstract form (the process image) in the PSS database. Hence, there is no additional context information that has to be migrated. Moving the process image is sufficient. (This is a bit of an oversimplification. It has, of course, to be ensured that external system components, like agents currently executing programs on behalf of a process or users monitoring its state, are not affected by a process migration. We assume here, like in the description of the replication mechanisms, that appropriate cluster mechanisms exist which hide the cluster structure from the external components.)

A naive algorithm for process migration can be built on top of the process state propagation mechanisms which used for the backup service. These mechanisms allow to copy the state of a running process to a remote server and can be utilized for process migration using the following simple mechanism:

1. Suspend the operation of the process which has to be migrated.
2. Use the copy mechanism to transfer the process state to the new site.
3. Install the process at the new server.
4. Remove the process from the old server and update the configuration information, which includes informing the concentrators of the changed location.
5. Resume the operation of the process.

This simple algorithm provides the necessary functionality for process migration and requires very little additional components in the server, since it is based on the propagation service. A closer analysis, however, reveals that it has a number of disadvantages:
11.3. A smarter algorithm

- It cannot ensure the permanent availability of a process during its migration, which is unsatisfactory given the effort taken to make important and critical processes continuously available. A migration mechanism for the COPS environment should integrate soundly with the availability postulate to allow the migration of mission-critical processes without a loss of accessibility.

- While the time a process is suspended may not be very long if only a single process is migrated, the suspension can become a severe problem in the case of bulk migrations, i.e., if all processes on a server have to be transferred because the server has to be shut down. Transferring the processes in serial order is very time-consuming; first suspending all processes and then migrating them increases the time the single processes are unavailable.

An alternative approach, which provides a higher degree of availability, can be based on the replication service itself. Exploiting the fact that a highly available process is already existing on two servers, process migration can be simply achieved by switching primary and backup. In the remainder of this chapter, we will present such an algorithm and discuss its properties.

11.3. A smarter algorithm

11.3.1. Objectives

The High Availability Process Migration algorithm (HAPM) is built on top of the replication service which is part of the cluster configuration adapter. Remember that the replication service allows to maintain multiple synchronized copies of a process, which is utilized by the HAPM to guarantee the availability of a process while it is moved from a source to a target host: Instead of halting the process while its image is copied to the target, the backup of the process is shortly activated by upgrading it to the critical availability mode. This implies that, via the replication service, the process image is automatically copied from the current server to the target server. It is then possible to switch primary and backup and afterwards degrade the process to its previous availability. The result is that the process is now running on the target server. Applying this algorithm to all processes running on a given server allows to shut it down without affecting any ongoing activities. Note that it is possible to select the new server on a per-process basis, which allows to evenly distribute the load of a terminating server to the remaining ones.

The advantage of this scheme is that it allows the safe migration of processes without the need for additional hardware or software components. The crucial point of server migration is to ensure the atomicity of the migration process, so that neither a process gets lost nor ends up being stored on two servers. Achieving this in an environment without backup mechanisms would require the deployment of transactional middleware such as persistent queues or TP-Monitors [BN97]. Thus, the advantages of this smart algorithm are threefold: First, we preserve platform independence since we do not have to rely on DBMS providing middleware interfaces (e.g., the XA interface needed for interaction with most TP monitors) — many object-oriented DBMS, for example, do not provide these mechanisms. Second, system complexity is kept bounded (we do not need yet another middleware component that has to be bought, installed, and maintained), which fosters the portability of a PSS and simplifies new installations. Finally, and perhaps most important, our solution has good performance, leading to the fact that the backup mechanisms provide a suitable base for fast process migration which allows to use it even for the transport of large process sets.
11. From Process Replication to Process Migration

11.3.2. Extending the replication service

To implement the migration mechanism, we introduce a new availability level called symmetric critical. This level is derived from the critical availability level, i.e., a hot standby algorithm is used that applies changes synchronously to both copies of a process. The difference of the symmetric critical level is that changes can be initiated at both nodes. (Recall that normally, only the primary can make changes which are then propagated to the backup node.) This is needed for the migration mechanisms since while a process is moved to a new location, messages can arrive at the old as well as at the new node, depending on how fast the information about the new location propagates in the system.

To avoid inconsistencies due to concurrent changes on both copies, a locking protocol is used: The primary node maintains a lock for each replicated process. It is used to synchronize changes to the process state and has to be acquired by any node wishing to update the process. From the primary node, the lock is accessible without overhead since it is stored locally. The backup node has to request the lock by sending a message to the primary. This scheme has the advantage of reducing message overhead as far as possible since only the backup has to request the lock over the net. In the more likely case that the primary performs the changes, no messages are needed.

Note that in principle, the process migration could be implemented using important availability (i.e., cold standby semantics). From the correctness point of view, this poses no problems since for important as well as critical processes, a 2-safe algorithm is used that ensures that no changes are lost. The cold standby mechanism, however, has the disadvantage that changes are not applied directly to the backup copy, but stored in a persistent hashtable and applied only in the case of failover. While this makes sense for a backup mechanism (where in the normal, error-free case the backup copy will never be used), in the case of process migration it is known that the backup copy will be needed (since it becomes the new primary copy). Using a cold-standby scheme in this case would thus only cause unnecessary additional overhead since each change would have to be stored twice, first in the hashtable and then in the process copy.

11.3.3. Migrating processes

The migration algorithms vary depending on the availability level of the process to be migrated:

- Normal availability (Fig. 11.1): The process is promoted to synchronous critical availability, which causes a change buffer containing its actual state being sent to the target node (Steps 1
11.4. Properties of the algorithm

11.4.1. Correctness

The aim of the HAPM algorithm is to provide availability while preserving consistency during process migrations. We have to show that consistency is really maintained, i.e., that no inconsistent state can be reached regardless of node or communication failures. We can drill the notion of consistency down to the property "no change buffers are lost", i.e., once a change to a process has been committed on one server, it is and stays visible system-wide. To prove that our system has these properties, we use the same arguments as for the backup mechanisms [KGAM96]. In all cases, a process is migrated by going through the synchronous critical stage. This implies that a hot-standby algorithm is used to replicate the process. A hot standby algorithm uses 2PC techniques [BHG87] to guarantee consistency among the distributed copies, ensuring that all replicas are updated atomically.

Our notion of consistency does only cover mutual consistency among the servers in the system. It is still possible that a message from a client is lost due to a server crash. Consider the case where a message (e.g., containing the results of a program that just finished) is received by a server, and the server crashes before it can handle the message. In this situation, the program results are lost. This problem does also apply to our migration algorithm if, for instance, the primary server of a
11. From Process Replication to Process Migration

process in *synchronous critical* mode fails. If these kinds of failures are not tolerable, there is the possibility to use reliable communication mechanisms like persistent queues [GR93, BN97] for the communication between clients and servers, or to make clients fail-safe, i.e., require them to re-transmit messages if no acknowledge about the successful processing of a message is received within a certain timeframe.

### 11.4.2. Scalability

The COPS architecture presented in chapter 9 allows to reconfigure a cluster if the load increases, since new servers can be added whenever necessary. Because of the transparency achieved through the concentrators, these configuration changes are not noticed by clients. The load-balancing service (see below) ensures that part of the other node's work is transferred to the new server. It is also possible to start an arbitrary number of concentrators, so that an increasing number of clients provides no problems. In spite of this ability to scale a cluster at will, the question might arise if the cluster maintenance algorithms are scalable as well. To discuss this problem, we divide the algorithms into three categories:

- The complexity of the cluster maintenance algorithms implemented by the cluster configuration manager is dependent on the cluster size since changes have to be synchronously propagated to all nodes. Cluster re-configurations occur, however, only if servers fail or new nodes are added, which is rather seldom.

- The complexity of the process replication mechanisms does not depend on the cluster size since, regardless of the number of servers in the system, a process is always replicated on two nodes only. Given that the network bandwidth is sufficient, adding servers will thus not impact the overhead caused by the replication.

- The main part of the process migration algorithm does also affect only two nodes and is thus not affected by the cluster size. Only the final propagation of the new process location affects all nodes in the cluster, but it requires only the sending of a rather short piece of information and can be implemented using efficient broadcast mechanisms if the underlying network supports this.

### 11.4.3. Overhead

The overhead of the migration facility is mainly determined by the replication mechanisms that are used to transfer processes between servers. From a performance perspective, the overhead of the *symmetric critical* replication mode can be compared to the *critical* mode used for the backup algorithms. Recall that both modes use the same replication techniques, with the difference that the symmetric mode allows updates on both servers while a process is migrated. Hence, the performance observations described in Fig. 10.14 apply to the migration facility as well. The overhead determined, 90% for a large database, is acceptable given that processes are running in this mode only while they are migrated. It has especially to be taken into consideration that a "standard" implementation of process migration would lead to a blocking of the process while it is migrated, while with the HAPM algorithm it remains functional. Hence, we can conclude that the HAPM migration facility is an important contribution towards more efficient PSS cluster management and scalable systems.
11.5. Load balancing policies

The COPS architecture makes a clear distinction between the load balancing mechanism (implemented through the process migration facility) and load balancing policies that can be implemented on top of the mechanism. This allows to easily adapt or even exchange the policy while the underlying mechanism stays the same. A local process migration manager provides two services to the local load balancing manager: a singleton migration mechanism that allows to move one process, and a bulk migration mechanism that permits to move a whole group of processes at once. The bulk migration mechanism can apply a number of optimizations that are possible when multiple processes are moved, like a reduced number of messages and optimizations in transaction processing at the servers. It is up to the load balancing managers to implement suitable policies by using the migration services and load information collected in the cluster. The load balancing policy can then easily be adapted by changing the load balancing managers. A load balancing policy usually consists of three "sub-policies" [SHK95]:

- The information policy, specifying which information about system load is accessible to the load balancing managers
- The transfer policy, determining when a process has to be moved to another server.
- The placement policy, determining the new node to which a process is to be moved.

In the sequel, we will discuss suitable policies for process support clusters that make use of the special requirements in these kinds of systems:

- **Information Policy**: As basis for the load balancing decisions, the load balancing managers have to have knowledge of the current load distribution in the cluster. Because of the availability needed in mission-critical applications (and because of the potentially large and physically distributed architecture of a cluster), a centralized solution is not feasible. Instead, knowledge of the system load has to be stored on each server. A suitable cooperative protocol for the exchange of load information in a distributed system is the Gradient Model Method [LK87], which reduces message overhead while ensuring that all nodes have a view of the current system load that is as close to reality as possible. The asynchronous character of this protocol guarantees the necessary scalability even in very large clusters.

Another aspect of the information policy is the measure that is used for determining the load of a particular server. The load of a PSS server is determined by three values: The processor load of the machine the server is running on, the processor load of the database server, and the disk utilization of the database server. A saturation of any of these values signals a high server load and indicates that processes should be migrated. Averages of the values are determined by each server in regular intervals using appropriate operating system calls (for instance, determining the average length of the processor job queue), and normalized to a value in the range [0..1], where 0 means no load and 1 indicates high load. The server load index is then computed by taking the maximum of the three values. Measuring processor loads instead of the load of the server itself is the most portable solution, since it makes no assumptions about the implementation of the server. If a server is, for instance, implemented using multiple processes, determining the overall load is complicated and implies additional overhead. In contrast, the processor load is always easy to determine and provides a good measure.
• **Transfer policy:** The load balancing manager starts considering nodes for migration when the load on the local machine has been high for a certain time interval, i.e., when the server load index is at or near 1 for several minutes. This means that, on average, more processes are running on the node than can be handled. Determining a process to migrate is difficult because a process that is “running” on a particular server may cause no activity on that server at all. For example, long-running external programs or activities may be active at the time and the process is waiting for them to finish. Because of this, it is not possible to choose an arbitrary process for migration. Instead, a process that is likely to cause load on the server has to be selected. To this end, each server maintains a history of the activities over the last hour. It lists each process that had a state change over the time interval, together with the total number of state changes that occurred. If a process has to be chosen, the system selects the one with the highest number of state changes over the last hour. This strategy can be seen as a LRU-algorithm that assumes that a process that caused system load in the past is likely to be active in the future.

Given that the overhead for migrating a **important or critical** process is low compared to a non-replicated process, highly-available processes should be chosen over normal ones whenever their loads are similar. This reduces the additional load imposed by the migration mechanisms.

• **Placement policy:** The placement policy is different for the different availability levels, since it is important to reduce the effort for migration as far as possible by exploiting special characteristics. If a normal process has to be migrated, the load balancing manager chooses the currently least loaded node based on its gradient information. For important and critical processes, it is first evaluated if the backup server has a sufficiently low load. In this case, primary and backup change their roles, keeping the overhead at a minimum. Otherwise, the node with the lowest load index is selected like in the case of normal processes. Note that this policy is suitable for LAN environments only. If the nodes are distributed over a large area, the placement policy has to take into account the communication cost between the replication partners, ensuring that replicas are in relative neighborhood.

The load balancing policies presented in this section can be seen as a first step towards the development of optimal policies. We have, for example, not yet evaluated the usage of heuristic methods for choosing both the processes to migrate and the nodes where they are moved to. Fine-tuning the policies will have to be subject to future research.

### 11.6. Discussion

The solution we have described implements process migration services by relying on process replication, which has the advantage that a process is available to clients even while it is being migrated, and, as we have described, fosters correctness. Process migration in general is an important contribution towards scalable systems. Implementing process migration using the HAPM algorithm complements the availability mechanisms described in the previous chapter and is an important prerequisite for mission-critical processes.

The performance results of chapter 10, which apply also to the HAPM migration mechanisms, allow to conclude that retaining the availability of processes while they are migrated is possible with a comparatively small overhead. It can be expected that the answer time caused will be much shorter than the delay that would be caused if a process was deactivated during its migration. In certain bounds, the performance can be improved by selecting an appropriate underlying
infrastructure. Relying on homogeneous databases and their built-in replication mechanisms, for instance, can improve the performance of the migration service. Note, however, that the severe delays observed when changing the replication schema in a Oracle-based system have to be resolved in order to make this option feasible.

Overall, the concepts presented in this chapter show how the COPS architecture and its replication service can be developed into a highly scalable cluster that provides advanced load-balancing guarantees. In the scope of this dissertation, there has not been the time to fully investigate this issue. Future work in this area will have to investigate optimization possibilities for the load balancing policies. It has to be evaluated if heuristics can be used to better plan the placement of processes and how far statistical information, as it is provided by most workflow engines, can be exploited to control the load distribution. Another interesting question is how much performance can be gained if "real" clusters, i.e., distributed platforms with dedicated high-performance links, are used as a platform. Together with the replication and migration algorithms of COPS, they could allow to use process support services techniques in areas (like scientific computing) that today cannot benefit from it because of the overhead.
11. From Process Replication to Process Migration
12. Conclusions

12.1. Summary

This work has developed the concept of process-based management of distributed, heterogeneous environments. A process is a collection of program executions and data exchanges over a heterogeneous network. A process support system (PSS) can be seen as a metaprogramming environment for these kinds of applications, providing modeling as well as runtime support and offering to users a set of additional services that help to improve distributed applications. The concept of a process support system is novel insofar as it tries to integrate a variety of services which are currently offered by different middleware tools (like TP monitors, workflow tools, and distributed computing environments), applying them to the special case of process-based computing.

The benefits of process support systems can be compared to those of database management systems in the sense that both try to isolate certain aspects from applications — data management in the case of DBMS, process management in the case of process support systems. The advantage of this approach is that now it is possible to both improve the quality of process management and develop new services. Examples for quality improvements in the context of process support systems are enhanced execution guarantees like atomicity and synchronization. Added services incorporate issues such as on-line monitoring services, logging facilities, and analysis tools. In addition to the development and implementation of a basic PSS infrastructure, the thesis has concentrated on improvements of service quality, focusing on fault tolerance and scalability.

From a practical perspective, the aim of the thesis has been the development of a process support kernel, which was also implemented in form of the OPERA system. The motivation for the kernel concept is the observation that the potential application areas for process support have a set of common requirements, but each area has unique demands which cannot be found in other applications. Hence the idea of a kernel, which provides generic functionality needed in all applications and can be extended by specific elements to adapt it to a specific application.

The modeling language developed for the OPERA system, OCR, meets a number of unique requirements, the most salient of which are flexibility and extensibility. Flexibility is important because the language is used as an internal representation, very much in the sense that assembler languages are used for programming. Flexibility is ensured in OCR by using a rule-like mechanism for the specification of control and data flow. This mechanism allows to map a large variety of modeling languages which are to be used by workflow modelers. Extensibility is needed because of the kernel characteristic of the OPERA system. Given that specific applications may need the support of special constructs in the modeling language, it is a necessary requirement that the modeling language is extensible. In OPERA, extensibility is guaranteed through the use of an object-oriented structure which makes it easy to extend it by using class inheritance mechanisms.

Of the value added services which can be provided by a process support system, fault tolerance is the perhaps most important. To a large degree, the thesis has focussed on these aspects, investigating the introduction of fault tolerance mechanisms into a process support context. In the chap-
12. Conclusions

In Chapters 5, 6, and 7, a concept was introduced that allows to aid process designers in the construction of fault-tolerant processes. Two main abstractions were identified as especially useful in a process support context: Failure atomicity and exception handling. Failure atomicity allows to abstract from failure handling by introducing the concept of rollback, i.e., the automatic establishment of a consistent state after a failure has occurred. We have shown that in a PSS context, advanced transaction concepts are needed. In contrast to failure atomicity, exception handling mechanisms like they are used in programming languages have the potential of specifying forward recovery, i.e., the specification of repair and contingency tasks that can make up for an error that occurred. The combination of atomicity and failure handling delivers a very powerful fault tolerance concept for process support systems which offers a wide variety of failure handling strategies. We have described this integrated concept in detail and have presented a correctness criterion that allows to assess the correctness of process specifications. This criterion can be used as the basis for a validation service automatically checking the correctness of process specifications and giving hints to programmers concerning optimizations. This service has also been implemented as part of the OPERA prototype.

In Chapter 8, a mechanism for inter-process communication was introduced that is based on event handling. Inter-process communication allows to overcome the usual back-box-mechanism used in process-based environments. The benefits are a much richer process model which allows, for instance, to increase the performance of processes since intermediate results of a process can be made accessible to other processes, which can then timely react to them. Inter-process communication is also a key requirement if process support systems have to be combined with certain types of groupware systems. The problem in developing a IPC mechanism for OPERA is the need for a sound integration with the atomicity and exception handling mechanisms. An abort of processes can invalidate intermediate results, a situation in which it is not clear whether other processes depend on these results have to be aborted as well or should remain running. The only solution to this problem is to give the process designers the possibility to pragmatically chose the suitable strategy. The mechanism that we have proposed allows to deliberately set an appropriate termination mode for each imported event. The benefit of the mechanism proposed is that it is possible to use inter-process communication facilities while preserving the transactional semantics of the process model.

The challenge in developing an architecture for a PS kernel was to provide a platform for process management that is extensible enough. To achieve this, a modular structure based on an object-oriented design was chosen. It allows to easily add new modules if this should be necessary. The basic architecture was extended towards a cluster of process servers in the COPS approach. The main challenge has been here to establish structure transparency. External components, like clients or invoked applications, should have the impression of a single server. To achieve this, concentrators were added to the design that hide the actual structure of the cluster from the clients. The resulting COPS architecture is an important prerequisite for available and scalable systems.

One of the improvements implemented on top of the cluster architecture has been a backup mechanism ensuring continuous availability. From a research point of view, the main contribution of the thesis in this regard has been the experimental evaluation of a number of implementation options for a backup service. The main result was that implementing backup as part of the application layer (instead of relying on services provided by the underlying databases) is a solution that is both acceptable from the performance point of view and beneficial because of a high degree of flexibility. Only the application-level, semantic backup allows to control the replication mode on a per-process basis, and it is possible only with this version to use different databases for cooperating servers.

An interesting extension of the replication mechanism was presented in Chapter 11. Here,
12.2. Outlook

While the thesis has discussed the problem of fault tolerance and scalability in process support systems in detail, there are many other aspects of process support systems that wait to be evaluated. In chapter 2, we have presented an overview of the necessary functionality of such a system. Only part of the problems occurring in such a system could be discussed in this thesis. Of the areas presented, the following are especially interesting:

- Scheduling of applications and load balancing between execution sites. In order to make optimal use of the resources in a distributed environment, the PSS should be able to balance the load between the nodes and invoke applications in a way that maximizes throughput and minimizes answer times. This problem is different from the problem of process placement, which was discussed in chapter 11. The main challenge here is here to find appropriate concepts for describing resource requirements and load characteristics in a way generic enough to meet the needs of all the types of process-based applications.

- The problem of appropriate analysis tools for process support environments has not yet been considered widely. Analyzing the histories of previously executed processes can deliver important information which can be used to optimize processes. In this context, the application of data mining tools has been proposed [AGL98], but much more work is necessary in order to find suitable algorithms and models that allow to optimize the various aspects of process execution.

- Load balancing between process servers could only be discussed shortly in this thesis, and we have viewed it mainly from the perspective of extending the process replication service. For a complete solution, much more work is necessary, which will, among others, include finding appropriate load measures and suitable heuristics for the placement of processes.

Another important field for future work is the application of process support concepts to other application areas. The OPERA kernel has already been used in projects tackling specific problem fields. An example is the GEO Opera extension, which aims at solving scientific problems like the one described in chapter 2. Here, the kernel was extended with important functionality, including meta-data management and the automatic re-computation of data sets if source objects are modified [AH97].

Another project in which the kernel was successfully used is the WISE project (Workflow based Internet Services) [AFH+99], which aims at providing an environment for Internet-based inter-enterprise workflows. The application of process support concepts to this field requires a number of interesting extensions, from suitable security concepts which can overcome the problems of a generally insecure environment to appropriate modeling tools which can exploit the immense potential for collaborative interaction inherent in Internet-based applications.

Further work here is clearly possible and necessary. Of particular interest could it be to study the application of PSS concepts to high performance computing. The paradigms used in such
12. Conclusions

Applications are today fundamentally different to the metaprogramming approach which is at the core of the PSS concept. It has, however, been claimed that suitable programming support for highly-distributed high-performance computing is still missing [GFLH98]. Hence, using process support systems for such tasks can be beneficial. It requires, however, to improve the performance of a PSS significantly and, more important, to provide suitable high-performance data transport mechanisms. This concerns the hardware platform as well as suitable software concepts. In terms of hardware, the usage of high-performance cluster concepts has to be considered [Pfi95]. In addition, suitable software support is necessary which can code and decode the data objects and distributed them with at little latency as possible. The coupling of OPERA with high-performance computing platforms [ASLV94] could be very interesting in this regard.
Bibliography


Bibliography


Bibliography


Curriculum Vitae

25.11.1966 Born in Bensberg, Germany
1986 Abitur
Paul-Klee-Gymnasium, Overath, Germany
1986–1988 Civil Service
1988–1995 Study of Computer Science
University Erlangen-Nürnberg, Germany
Degree: Diplom-Informatiker
1995-1999 Doctorate
Department Informatik of ETH Zürich