Diss. ETH No. 17600

A Workflow Approach to Stream Processing

A dissertation submitted to
ETH ZURICH

for the degree of
Doctor of Technical Sciences

presented by
BIÖRN JOHAN BIÖRNSTAD
Dipl. Informatik-Ing. ETH
born February 3, 1978
citizen of Raperswilen TG
and Wigoltingen TG

accepted on the recommendation of
Prof. Dr. Gustavo Alonso, examiner
Prof. Dr. Heiko Schuldt, co-examiner
Prof. Dr. Cesare Pautasso, co-examiner

2008

Typeset with LaTeX
Electronic version

Acknowledgements

As usual, this dissertation was not the work of one person alone. I would like thank the following people for their support during my PhD studies. First of all, I want to thank Prof. Gustavo Alonso for accepting me as a Ph.D. student and, especially towards the end, for his patience with my never-ending dissertation.

My thanks also go to Prof. Heiko Schuldt and Prof. Cesare Pautasso for accepting to be my co-referents. I am indebted to Cesare who became my co-advisor after completing his own Ph.D. Cesare always encouraged me in what I was doing and supported me in good and bad times, even after he had left ETH. This dissertation owes much to his feedback.

I was fortunate to spend my time in an international research group with nice and interesting people. In particular, I would like to thank my former office mates who contributed to a comfortable working environment. These were, in chronological order, Win Bausch, Cesare Pautasso, Daniel Jönsson, Christian Plattner, Thomas Heinis and Patrick Stüdi. On almost every weekend, Thomas, Patrick and I "held the fort" at IFW for the rest of group.

I appreciate very much the help from my father who fought his way through the dissertation to proofread it in short time.

Last but not least, I want to thank my parents and my brother for their unconditional love and support throughout all the years. Without their help, I would probably not be writing these lines now.

Contents

Abstract

Today, workflow languages are widely used for service composition. Workflow and business process management (BPM) systems are typically based on a step-by-step execution model where a task is started, the result is received, and then the next task is scheduled for execution in a similar fashion. To track the execution of individual service invocations and of the overall workflow process, a state machine based approach is used. The model corresponds to the request-response nature of many service interfaces and maps directly to technologies such as Web services or business process modeling specifications such as WS-BPEL. However, there are services which do not follow this interaction pattern but rather proactively produce new information to be consumed by an application. Examples include RSS feeds listing the latest bids at an auction, result tuples from a data stream management system (DSMS), stock price tickers or a tag stream from an RFID reader.

This dissertation shows how to extend traditional state-based workflow management techniques with the necessary features to integrate streaming data services and combine them with conventional request-response services. First, we study the problem of accessing a stream from within a workflow process. We discuss different alternatives in terms of expressiveness and performance. One approach involves the extension of the traditional request-response task model. We show how to accomplish this on the level of the workflow language and describe the necessary changes in a workflow engine. Our solution provides a notion of stream which abstracts from the mechanism or protocol used to access the content of the stream. This makes the service composition independent of what type of stream is processed, e.g., RSS items, RFID tags or database tuples.

The invocation of a streaming service leads to a stream of result elements independently flowing through the invoking process, thereby creating a pipelining effect. This leads to safety problems in the execution of the process, as state-based workflow execution models are not designed for such parallel processing. E.g., considering that the tasks of a process do not always take the same time to execute, a task might not be ready to process a stream element when the element is offered to the task. This can lead to loss of data in the stream processing pipeline. We discuss different solutions to avoid the safety problems connected with pipelined processing and identify the minimal necessary extension to the semantics of a workflow language. This extension is based on a flow control mechanism which controls the flow of data between tasks in a process and allows the safe use of pipelining in the execution of a process.

Our extended semantics for pipelining will block a task when it is not safe to execute it. However, if the services that are composed into a stream processing pipeline show variations in their response time, this will decrease the throughput of the pipeline, as our measurements show. Therefore, based on the flow control semantics, we show how to use a buffered data transfer between tasks in a pipelined process. The buffers will smooth the irregularities in the task duration and allow a pipeline to run at its maximum possible throughput.

Finally, to evaluate our approach, the stream processing extensions proposed in this thesis have been implemented in an existing workflow system. Apart from describing the implementation in detail, we present several performance measurements and an application built on top of the extended system. The application is a Web mashup which integrates the data from a live Web server log with a geolocation database in order to provide a real-time view of the visitors to a Web site together with their geographic locations on a map.

Kurzfassung

Die Verwendung von Workflow-Sprachen zur Komposition von Diensten ist heute weitverbreitet. Workflow- und Businessprozess-Systeme (BPM) basieren meist auf einer schrittweise Ausführung von Prozessen, bei der eine Aufgabe gestartet, das Resultat erhalten wird und anschliessend die als nächstes auszuführenden Aufgaben ermittelt werden. Um den Fortschritt der einzelnen Aufgaben wie auch des ganzen Prozesses zu verfolgen, verwenden herkömmliche Workflow-Systeme endliche Automaten. Dieses Model entspricht dem "request-response" Verhalten vieler Dienste und entspricht direkt Technologien wie Web services oder Spezifikationen für Prozessmodellierung wie WS-BPEL. Es existieren jedoch auch Dienste, welche nicht nach diesem Muster kommunizieren, sondern aktiv Informationen produzieren, die von Applikationen konsumiert werden können. Einige Beispiele hierfür sind RSS-Feeds mit den neusten Geboten an einer Auktion, Resultat-Tupel eines Data-Stream-Management-Systems (DSMS), Börsenticker oder der Datenstrom eines RFID-Lesegeräts.

Diese Dissertation zeigt auf, wie man herkömmliche zustandsbasierte Workflow-Techniken erweitern kann, um Datenströme integrieren und mit konventionellen "request-response" Diensten kombinieren zu können. Als erstes untersuchen wir das Problem des Zugriffs auf einen Datenstrom aus einem Prozess heraus. Wir diskutieren verschiedene Alternativen, welche sich in der Aussagekraft und der Performanz unterscheiden. Einer der Ansätze beinhaltet die Erweiterung des herkömmlichen request-response Models einer Aufgabe. Wir zeigen, wie die Workflow-Sprache entsprechend erweitert werden muss, und wir beschreiben die nötigen Änderungen im Workflowsystem. Unsere Lösung bietet eine abstrakte Sicht auf einen Datenstrom, welche unabhängig ist vom konkreten Protokoll für den Zugriff auf den zugehörigen Dienst. Dadurch wird eine Dienste-Komposition unabhängig vom Typ des verarbeiteten Datenstromes, z. B. RSS-Elemente, RFID-Kennungen oder Datenbanktupel.

Der Empfang eines Datenstroms von einem Dienst führt dazu, dass mehrere Datenelemente unabhängig durch den Prozess fliessen, welcher den Dienst aufgerufen hat, wodurch ein Pipeliningeffekt ensteht. Dieser Effekt führt zu Sicherheitsproblemen bei der Ausführung des Prozesses, da diese Art der überlappenden Ausführung in Workflow-Sprachen nicht vorgesehen ist. Weil die einzelnen Aufgaben in einem Prozess nicht immer gleich viel Zeit benötigen, kann es zum Beispiel vorkommen, dass eine Aufgabe nicht bereit ist, ein Datenstromelement zu bearbeiten, weil sie noch mit dem vorhergehenden Element beschäftigt ist. Dies kann zum Verlust des betreffenden Datenelements führen. Wir diskutieren verschiedene Alternativen zur Vermeidung der Sicherheitsprobleme und erörtern die minimal benötigte Erweiterung der Semantik einer Workflowsprache. Diese Erweiterung basiert auf der Kontrolle des Datenflusses zwischen den verschiedenen Aufgaben in einem Prozess. Diese

Flusskontrolle erlaubt den gefarhlosen Einsatz von Pipelining bei der Ausführung eines Prozesses.

Unsere erweiterte Semantik für Pipelining blockiert eine Aufgabe, falls sie nicht sicher ausgeführt werden kann. Falls jedoch die Dienste, welche in einem Prozess kombiniert sind, unregelmässige Ausführungszeiten aufweisen, führt dies zu einem verringerten Durchsatz in der Verarbeitung eines Datenstromes, wie unsere Messungen zeigen. Auf der Flusskontrolle aufbauend, zeigen wir, wie ein gepufferter Datenaustausch zwischen Aufgaben verwendet werden kann, um diese Unregelmässigkeiten auszugleichen. Dies erlaubt einem Prozess mit Pipelining, seinen maximalen Durchsatz zu erreichen.

Um unseren Ansatz zu beurteilen, haben wir schliesslich die vorgeschlagenen Erweiterungen für die Bearbeitung von Datenströmen in einem bestehenden Workflowsystem implementiert. Abgesehen von einer detaillierten Beschreibung der Implementation, diskutieren wir verschiedene Messungen und eine Applikation, welche aufbauend auf dem erweiterten System realisiert wurde. Die Applikation ist ein “Web mashup”, welches Daten aus einem Webserver-Log mit einer Geolocation-Datenbank integriert, um in Echtzeit die Besucher einer Website zusammen mit ihren geografischen Standorten auf einer Karte zu visualisieren.

1 Introduction

1.1 Motivation

Workflow management or *business process management* has its roots in office information systems and gained popularity in the context of *business process reengineering* as an instrument to coordinate tasks and resources (e.g., people or software applications) [34]. The Workflow Management Coalition defines workflow management 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. [126]

During the last decade a huge number of related tools have been developed and the literature has been steadily growing, making workflow management a well established technology. Over time, the degree of automation in business processes has increased, i.e., the tasks in a business process are executed less by people and more by software applications. With the convergence of workflow management, enterprise application integration (EAI) and Web technologies [57], this has led to the use of workflow management techniques for fully automated service composition [35, 95].

Workflow systems are mostly based on a step-by-step execution model where a task is invoked, a response received, and then the next task is scheduled for execution in a similar fashion [76]. For this, the state of each task (e.g., idle, running, finished) is tracked by a finite state automaton. The model corresponds to the request-response nature of many service interfaces and maps directly to technologies such as Web services [3] or business process modeling specifications such as the Web Services Business Process Execution Language (WS-BPEL or BPEL) [89].

Successful as this model is, it also has limitations. These become apparent when traditional business processes must deal with services which do not simply react to the exchange of messages, but instead proactively produce new information to be consumed by the application [48]. Web feeds [13, 87] today provide huge amounts of information [39], e.g., about Web logs [110], recent earthquakes [115] or listing the latest bids at an auction [28]. Data stream management systems [48] filter and correlate real-time streams and notify applications with the distilled information. The deployment of RFID technology in supply chains involves a huge amount of events flowing towards central servers while complex decisions must be made based on this data [25]. Efforts are being made to make data streams accessible through Web services [97] as part of the Grid [42], an active research area in workflow management [33, 11, 96]. These examples show the need for *stream processing* capabilities in workflow management tools. A *stream* is defined as a possibly infinite sequence of elements [105, 74]. The data type of the elements depends on the service which generates the stream.

Also services or programs which normally follow the request-response model can be interacted with more efficiently if their output is treated as a stream. This is possible if the service is computing its result incrementally and internally accumulating the individual parts before returning the overall result. If the service can be adapted to directly return the individual elements of the result, this gives the invoking client the possibility to incrementally process the result without having to wait for the service to complete its computation [101].

Examples of services whose output can be turned into a stream are databases, where a result set consists of several tuples, or a program whose standard output can be split into several lines or records. In [109], the authors describe how to speed up the visualization of a galaxy and star formation experiment. The employed simulation generates large data files with snapshots of the evolving system. Both the simulation and the visualization of the snapshots are computationally intensive. In order to reduce the time to compute an animation, the authors take advantage of the independence of the individual snapshots. This allows the visualization of the frames to be parallelized over distributed computing resources. The approach could be taken a step further by having the simulation output each snapshot as soon as it has been computed. The snapshot can then be rendered in parallel to the computation of the next snapshot. This way, the simulation and the visualization are overlapped and the animation can be viewed while the simulation is still running.

The need for supporting stream based processing in service composition tools goes beyond the increasingly important ability to cope with services that produce data streams. As an example, in a customer support system with a high load, streaming can avoid having to create a process instance for each incoming support request. Instead, a single process is started and the requests are processed in a pipelined manner. Such an approach can have significant advantages in terms of scalability and expressiveness of the business process. In standard process description languages such as BPEL, an incoming request message creates a new process instance for the handling of the request. Also, BPEL does not support the pipelined processing of multiple stream elements. This means that the resources available to the system might not be used to full capacity and that an inherent limit on scalability is unnecessarily introduced.

1.2 Contributions

Stream access The very first step towards stream processing is the ability to interact with the source of a stream. As current workflow approaches assume a request-response interaction with external entities, we propose different ways of accessing streams from within a workflow. The approaches have different performance characteristics and differ in terms of expressiveness. One approach involves the extension of the traditional request-response task model.

We show how to accomplish this on the level of the workflow language and describe the necessary changes in a workflow engine.

Stream abstraction Different streaming services must be accessed through different protocols. These protocols employ different interaction patterns for retrieving the elements of a stream. Elements can be pushed by the service, pulled by the invoking client or even have to be periodically polled for. In our approach, we abstract from such lower-level details of accessing a streaming service and provide the user with a uniform view of a streaming service at the language level. This makes it possible to use the same process together with different types of streams, e.g., RFID events, SOAP message streams, RSS feeds or message queues, without having to change the structure of the process.

Safe pipelining By integrating streaming services into a workflow, multiple stream elements concurrently flow through the workflow, thereby creating a pipelining effect. While the pipelined processing of a data stream is beneficial in terms of performance, traditional state-based workflow languages have not been conceived for this type of interleaved processing. We analyze the problems that pipelining causes in such a language and discuss different possible solutions. We identify minimal but necessary *flow control* extensions to the semantics of a workflow modeling language which allows a workflow to be safely pipelined over a data stream.

Buffered information flow In many cases, a minimalistic solution for safe pipelining is not good enough. As we show in Section 6.3, the performance of a stream processing pipeline depends very much on the execution characteristics of the pipeline stages. Variations in a stage's execution time have a negative impact on the pipeline throughput. Such irregularities, however, can be evened out by placing buffers between the pipeline stages. We therefore show how flow control can be complemented by buffered information transfers in a process. We identify the minimal functionality required by such buffers and describe their integration into an existing system.

Transparent flow control An important aspect of our flow control extensions is that it changes the execution semantics of the workflow language in a way which is compatible with the existing semantics. Therefore, the syntax of the language does not have to be changed, as it still correctly reflects the semantics of the language. Thus, flow control is a property of the process execution and does not affect the design of a process. Flow control (and the use of buffered information transfers) can be transparently enabled or disabled when executing a process. An advantage of this approach is that streaming services can be integrated into existing processes with minimal changes.

We have implemented a prototype to validate our approach to stream processing as well as to measure its performance. We also describe in detail an application which has been implemented on top of the prototype system. The application is a *Web mashup* which integrates two data sources, a Web server access log and a geolocation[1] database. The mashup delivers a real-time view of the location of the visitors of the Web site. Thanks to the stream processing and pipelining capabilities of our system information flows from the Web server log, through a data integration pipeline and to the user interface with a minimal delay.

1.3 Structure

Chapter 2 discusses related work, notably in the areas of workflow management, stream processing and the intersection thereof, stream processing with workflow systems.

Chapter 3 introduces the process modeling language used in this thesis. We define how process models are executed and describe the workflow engine which we have extended with stream processing capabilities.

Chapter 4 discusses how streaming services can be accessed from a workflow process. It studies the problems which arise when a data stream flows through a process and presents a flow control semantics, a minimal but necessary solution to allow safe pipelined workflow processes. Based on flow control, the chapter also shows how to use buffers to deal with varying task execution times.

Chapter 5 describes how we have extended an existing workflow system to support stream processing. The use of flow control inside processes requires to also use flow control in the rest of the runtime systems. We also discuss how to implement buffers to be used for the data transfer inside processes.

Chapter 6 evaluates the performance of the extended workflow engine. We quantify the overhead introduced by using flow control in the execution of processes. We investigate the use of pipelined processes as a means to provide a Web service.

Chapter 7 describes a real-time Web mashup application of our system where several transformation steps have to be executed over the stream from a Web server access log. We identify the challenges of this application and show how they can solved using our system.

Chapter 8 summarizes the contributions of the thesis and discusses possible directions for future work.

[1]IP address to coordinate translation

2 Related Work

This thesis extends the research area of workflow management in the direction of stream processing. In this chapter, we discuss work in the areas of stream processing and workflow management as well as in their intersection, stream processing with workflow systems. First, however, we briefly cover the more fundamental topics of pipelining and flow control.

2.1 Pipelines and pipelining

When pipelining is being used to process a stream, multiple elements of the stream are processed in parallel, each element in a different *stage* of the pipeline. The benefit of this is that the throughput of the pipeline in terms of stream elements per time unit increases. The performance increase depends on the depth of the pipeline which determines the degree of parallelism the pipeline can achieve.

A classical example of pipelining can be found in the area of microprocessors [18]. Here the execution of an instruction is split into different stages, e.g., *fetch*, *decode*, *execute*, *memory*, *write back*, each having separate execution logic in the processor. In order to be executed, an instruction passes through all the stages. As soon as an instruction has left one stage this stage can accept the next instruction, resulting in the pipelining effect. A pipeline in a microprocessor is always linear, whereas in our approach we allow for parallel branches[1]. Also, in typical microprocessors, the stages are synchronized by a global clock signal. This is in contrast to our workflows where each task executes asynchronously and flow control must be used for the coordination with other tasks. As discussed in Section 4.8 the problems which can arise when pipelining the processing of a stream using our approach are the same as when using microprocessors.

UNIX pipelines [9] have been used for a long time to compose standard UNIX programs into (simple or complex) data processing pipelines by connecting their standard input and output streams. Since all programs involved in a pipeline run in parallel, continuously reading and writing from inter-process *pipes*, the pipelining effect helps to execute the pipeline in minimal time. While UNIX pipelines are constrained to linear topologies, CMS Pipelines [56] extend the concept to allow multiple data paths inside a pipeline. The syntax for specifying such data flow graphs is, however, still a one-dimensional piece of text, making it difficult to manage complex compositions.

Automator [7] is a tool for creating simple “workflows” for automating repetitive end-user tasks in MacOS X. It allows the user to build linear dataflow pipelines

[1]We do not regard superscalar stages as parallel branches.

out of *actions* which can, e.g., remote control desktop applications or accomplish non-UI tasks such as converting files between different formats. Thus, this approach is similar to UNIX pipelines. However, Automator does not support a pipelined execution of its workflows, presumably to avoid any confusion on the side of the user due to an unfamiliar programming concept.

A similar tool in the context of Web mashups is Yahoo! Pipes [128]. It provides a visual AJAX editor for interactively building pipelines for processing RSS [13] feeds and similar data sources. Users can choose from a palette of predefined processing *modules*. The result of a pipe is published as another RSS feed or as a JSON data structure. Although mashups are programmed using data flow pipelines, the tool does not support infinite streams, let alone the pipelined processing of such. The primary data type is a list of items which corresponds to the data model of RSS. Many of the modules, such as *Filter*, *Sort* or *Count*, operate on an item list. The feature which comes closest to pipelining is the *Loop* module which applies another module to every item in a list.

According to [4], Damia [62] is a similar but more powerful tool than Yahoo! Pipes. It supports, amongst others, a more general data model and a larger set of data sources. The system supports streaming RSS and Atom sources through "notifications to its applications" when changes in the sources are detected, however, no details are given on this mechanism.

IBM's BatchPipes [61] is an application of pipelining in the area of mainframe computing. Traditionally, batch jobs work on data from persistent storage and are executed sequentially. Using BatchPipes, jobs can be run in parallel while the results are piped directly from one job ("writer") to another ("reader"). This results in a reduced total processing time of the batch jobs.

In its latest version, HTTP [40] has introduced the notion of pipelining. Although HTTP involves no pipeline of components as in a workflow, the network path between the client and the server can be regarded as a long "bit stream pipeline". Instead of waiting for a request to be answered by the server before submitting the next request, an HTTP client can use the pipelining feature to submit new requests over a persistent connection while other requests are still pending. Thus, the transmission of requests, the processing of requests at the server and the transmission of responses are parallelized, yielding a higher throughput.

2.2 Flow control

Flow control is a well known concept in digital communication [107]. It is used to control the rate at which a sender transmits data to a receiver, such that the receiver always has enough capacity left to accept and process arriving data. For example, when using a serial data interface, separate wires can be used for flow control signalling. In this regard, the RTS/CTS mechanism used by RS-232/ITU-T V.24 [63] has similarities with the marker based flow control mechanism developed in this thesis (Section 4.4.2).

Looking further up in the communication stack, transport protocols often provide a reliable communication channel [107]. Although such protocols support the retransmission of lost data segments due to transmission errors, using flow control can avoid additional data loss due to the receiver being overwhelmed by data or due to congestion caused in the network.

2.3 Stream processing

There exists a large body of research in stream processing, an overview of which is given in [105]. The article discusses several stream processing languages and argues, e.g., that every functional programming language can be used for general purpose stream processing. An example of such a functional language, not mentioned in [105], is Haskell [59]. A Haskell stream library [106] provides functions for manipulating streams. Such infinite data structures are supported by the language's *lazy evaluation* approach.

[74] discusses *dataflow process networks*[2] which are shown to be a special case of Kahn networks [69]. In dataflow process networks the processing of streams is modeled with *actors* which are connected by unbounded FIFO queues. Appending to a queue is a non-blocking operation, while reading from a queue blocks the caller if the queue is empty. An actor maps input tokens to output tokens when it *fires*. A set of *firing rules* can be used to specify the conditions under which the actor will fire. These firing rules are similar to the task activator in our meta-model (Section 3.1). Each firing rule specifies what tokens are expected at the individual inputs of the actor. This corresponds to a task activator requiring other tasks to have reached a certain state. This is specially the case when using flow control, as this also involves FIFO channels between tasks (Section 4.4.4). However, an actor must read from queues to determine which firing rule matches the input tokens. If a sequence of blocking read operations can be used for this, the set of firing rules is called *sequential* and can be implemented in a Kahn process network. In contrast, in our approach reading from the state storage (including FIFO buffers) is always non-blocking. Therefore, there is no restriction on the structure of the activator expression.

Although not focusing on stream processing, [101] presents a model for *arbitrary result extraction* from services. The paper defines *progressive extraction* where the same output parameter of a service delivers different values over time. This sequence of values can be regarded as a stream. However, the interaction is client-centric, i.e., the client needs to query the service for a new result. This behaviour corresponds to our pull-push approach (Section 4.2.2). Another formalization of the interaction with a streaming service can be found in the *Multi-responses* pattern described by [10]. The interaction corresponds to our multiple-output task model (Section 4.2.3), although our model abstracts from the concrete interaction with

[2]A *process* here denotes an entity which consumes input and produces output and should not be confused with a workflow process.

the service and can be used on top of other interactions, such as as the pull-based approach of [101].

2.3.1 Data stream management systems

In recent years the interest has grown in stream processing systems handling large amounts of real-time data. Several so-called *data stream management systems* (DSMS) have been proposed [48]. The focus of these projects has been on the efficient execution of *continuous queries* over data streams. In TelegraphCQ [23] these queries are specified in an SQL-like language. Aurora [22] employs a graphical representation of queries where operators of different queries form a loop-free directed graph (query plan) which reflects the flow of the data. Similarly, PIPES [71] also uses directed graphs to define the processing of data streams.

Similar to DSM systems is StreamIt [112], a language and compiler for high-speed stream processing. The language defined by StreamIt does not allow arbitrary graphs, but is strictly block-structured. Stream processing pipelines are defined by composing operators written in a C-like language. Such pipelines are then compiled in an optimized way for special purpose stream processing hardware.

Because the mentioned stream processing systems focus on performance, they restrict the language in which they can be programmed to graphs (or queries) of operators with well-defined semantics. This allows the systems to automatically rewrite or compile the specified stream pipelines to a more efficient version. In our approach, however, we focus on the composition of heterogeneous black box services which do not have a pre-defined semantics. This thesis extends the types of services which can be composed to also cover streaming services, thereby increasing the number of possible applications of the system.

2.4 Workflow modeling approaches with pipelining

There are a number of workflow engines and workflow languages that support streaming to some degree. OSIRIS-SE [15] aims at reliably handling large amounts of continuous data but uses a different programming model than most systems for this purpose. Here, a network of stream operators is setup by running a workflow process. The tasks of the process instantiate the operators which are then interconnected by FIFO-queues [16]. The engine has then little control over these operators that follow the multiple input/output model described in Section 4.3.1. It is possible for a stream operator to invoke a Web service with each received stream element. Compared to our approach such a service invocation is not part of the workflow process, but is rather part of the operator implementation.

An example of a tool supporting pipelined execution over non-streaming Web and Grid services is Triana [79]. Although lacking support for control flow branches and exception handling, its language based on dataflow networks fits with the queue semantics described in Section 4.3.1. Furthermore, Triana supports the continuous re-execution of a process such that tasks are invoked over each stream element. This is similar to our pull-push approach of consuming streams (Section 4.2.2).

Like Triana, SCIRun [92] is based on a dataflow network. Tasks communicate through FIFO-queues and can additionally produce multiple results during one execution similar to the push mechanism described in Section 4.2.3.

Similarly, Kepler [78] also uses data flow networks to model scientific workflows. More specifically, Kepler supports different execution semantics for a workflow graph. This is the same approach as we take to add safe pipelining to workflows. With the flow control extensions, we provide an alternative execution semantics without changing the composition language. Mostly, the *process network* semantics [17] is used, where tasks can read and write multiple stream elements during one execution. Thus, the system supports the pipelined processing of a stream. Instead of using flow control, these semantics employ conceptually unbounded buffers between individual tasks. An interesting feature is the support for pipelining over collections contained in the stream [81]. Thereby, a collection of elements can be expanded into the stream together with special delimiter elements to allow downstream tasks to recognize the collection boundaries.

YAWL [118, 117] is a workflow language and system based on the analysis of workflow patterns [119, 100] and inspired by Petri nets. The notions of place and token in YAWL seem suitable for stream processing. However, the system uses global variables for the data transfer between tasks. Since neither the language nor the system supports flow control, the data in global variables may get overwritten if tasks are executed repeatedly in order to process a stream.

The marker mechanism presented in Section 4.4.2 can be compared to a finite capacity Petri net [86] were the markers are places with a capacity of one token and tasks are transitions. It has been studied how Petri nets can be applied to workflow modeling [116]. However, it has also been shown that Petri nets have limitations when used to model complex workflows [118].

The newest version of UML [90] recognizes the need for streaming data transfer between actions in activity models. It allows inputs and outputs of actions to be declared as streaming. During one execution, an action may consume multiple tokens from a streaming input and produce multiple tokens on a streaming output.

[52] discusses how to extend a traditional state-based workflow model to support *anticipation*. With this extension, a task is allowed to enter an *anticipating* state when its predecessor has started to execute, creating a pipelining effect in the workflow. The intermediate output of the predecessor can then be made available to the execution of the anticipating task. However, the way data is exchanged between anticipating tasks is not strictly controlled. A task can update its output parameters while it is executing and an anticipating successor task can read its input parameters to receive updated output from its predecessor. There is no mechanism to synchronize this data exchange. The authors assume that tasks are mostly interactive and are executed by humans. Humans can make intelligent decisions regarding anticipation and data communication. They can decide when to anticipate, if at all, and how to deal with updated output from a predecessor task. Also, when humans

are involved, the data flow can be external to the workflow system and be controlled entirely by the humans.

[96] gives an overview of different ways of applying parallelism to the processing of data in the context of Grid workflows. Amongst others, the paper describes several patterns of pipelined execution which correspond to the different approaches of dealing with pipeline collisions presented in Section 4.3.1: *best effort* (discard data, Section 4.3.1), *blocking* (block, Section 4.3.1), *buffered* (queue, Section 4.3.1), *superscalar* (multiple instances, Section 4.3.1), *streaming* (multiple input, Section 4.3.1).

2.5 Workflow management systems

This thesis shows how to add stream processing capabilities to an existing workflow system, JOpera [94]. Therefore, in this section we review different ways of process modeling and different process execution models and compare them to the approach taken in JOpera.

2.5.1 Process modeling

Current process modeling techniques can be divided into two types, activity-based and state-based ones. In both approaches, processes are modeled as annotated graphs, the difference lies in the annotation possibilities and in the interpretation of the resulting graph.

Activity-based approaches

Using an *activity-based methodology* [47], the focus lies on describing *what* should be done and the *order* in which it should be done. A process model is thus described in terms of activities (tasks) and their execution dependencies, also called transitions. A transition usually has an associated condition which determines when the transition can be made. Activity-based languages typically support complex activities, i.e., the nesting of processes.

The meta-model we use in this thesis (Section 3.1) is activity-based. A major difference to most other activity-based approaches is that in the control flow we use entry conditions for activities (data conditions) instead of transition conditions. However, one can easily be implemented by the other. Also, most approaches focus on the control flow and do not model the data flow between activities as a graph as we do. Instead, data is passed around using process variables as temporary storage.

The following briefly describes examples of other activity-based process modeling approaches. The Web Services Business Process Modeling Language (WS-BPEL) [89] is an industry standard targeted at the composition of Web services. It emerged out of the fusion of two modeling languages, IBM's Web Services Flow Language (WSFL) [75] and Microsoft's XLANG [111]. WS-BPEL supports not only graph-based process models but also block-constructs (e.g., while-loops) which originate from XLANG. Although our approach does not support block-based control constructs, these can be implemented with the help of subprocesses (e.g., a control flow loop inside a subprocess).

The Business Process Modeling Notation (BPMN) [20] is a standardized graphical language. As a graphical notation, it is similar to UML but is focused on modeling workflow processes. XPDL [127] is a serialization format for BPMN diagrams, defined by the Workflow Management Coalition. Many modeling tools and execution engines are based on BPMN and/or XPDL [125]. In terms of features, BPMN/XPDL is a superset of our meta-model, however, the main control flow constructs are very similar.

UML [90, 103] supports the modeling of workflows through the Activities framework. In contrast to many other languages, UML Activity Diagrams support the graph-based modeling of the data flow. Another speciality is the notion of token, similar to Petri Nets. Tokens follow the control flow and data flow in the process.

Scufl is the language used by the Taverna [91] and Freefluo [114] projects. It is activity-based and supports explicit *data flow links* as in our model.

The Web Services Choreography Description Language (WS-CDL) [121] supports only block-based control constructs and no arbitrary graphs. Activities can have pre- and post-conditions associated which influences the way the process is executed. Another choreography-like language is defined by the ebXML Business Process Specification Schema (BPSS) [88].

Event-based process chains (EPC) [70], a semi-formal approach to process documentation [82], are based on the notions of *functions* (corresponding to activities) and *events*. Events are prerequisites for starting a function and at the same time represent the result of executing a function. Events and functions are connected in chains which form a graph structure. This approach is similar to our model insofar as the activator and data condition also denote the prerequisite for starting a task.

The con:cern project's approach [27] is also based on activities with pre- and post-conditions. Con:cern's use of named conditions [26] is similar to the events in EPCs and the starting conditions in our approach.

State-based approaches

State-based methodologies have their focus on the states of a system and its reaction to events. The possible states of a system are connected by transitions which indicate under which circumstances the system may change to another state. Usually, a transition is annotated with an event, a condition and an action (ECA). If the event occurs and the condition holds, then the action is executed and the transition is made. A state might also have an action associated which is executed whenever the system enters that state.

A first example of a state-based approach are state diagrams used to model finite state machines. UML defines a notation for state diagrams (*behavioral* and *protocol state machines*). The UML diagrams are based on Harel's statecharts [55] which have been applied to workflow modeling [124].

Similar to finite state machines are Petri Nets, a graphical and mathematical modeling tool which is applicable to many systems [86]. It has been shown that it is possible to apply Petri Nets to workflow modeling [116]. Because of their limitations when modeling complex workflows, the YAWL project has developed a workflow modeling language [118] which is inspired by Petri Nets and the analysis of common workflow patterns [100].

The meta-model of jPDL [67], the process modeling language used by JBoss jBPM, lies between UML state machines and Petri Nets. The underlying *graph oriented programming* approach, on top of which jPDL is implemented, is more expressive and very flexible. However, the semantics of the approach cannot be easily captured since the behaviour of a process model can be specified partly in terms of custom Java code.

Event-based process chains have been analyzed regarding the support of workflow patterns [82, 118]. The EPC approach was also extended in order to support all basic workflow patterns [82]. This resulted in *yEPCs* where the *empty connector* introduces a state-based behaviour.

2.5.2 Process execution models

The way navigation is implemented in JOpera is through compilation of the process model to an executable module and subsequent execution of this code (Section 3.3.2). Process instances can also be executed differently. A process model can be compiled to ECA-rules which are then executed by a rule-based workflow engine (e.g., [72, 24]). This is similar to our approach where the compiled code of a process consists of rules for the individual tasks expressed as Java statements. The OSIRIS distributed workflow engine [102] has the concept of *execution units* which is also similar to ECA-rules. An execution unit describes the execution of a particular task and contains information on where to transfer control after the task has been successfully executed or in case of a failure.

I many systems, navigation is based directly on a graph structure which has been derived from the process model. If the state of process instances is stored as graph structures in a database, it is possible to execute the navigation algorithm inside the database by issuing the appropriate SQL statements [76]. However, most systems use a graph representation in memory (e.g., [51, 54, 76, 92, 11, 67, 31, 19, 2]). In a workflow system which is implemented in an object-oriented language, this graph can be built of objects in the same language. These objects contain the process instance state as well as the navigation logic. The objects are kept in persistent storage and are loaded when the corresponding process instance needs to be navigated. It is also possible to represent only the structure of the process as object graph in main memory. Then, dedicated threads are used to execute process instances over the structure, thereby keeping the state of a process instance's active execution paths in execution *contexts* [53]. In order to separate the structure of a process model from its semantics, instead, the visitor pattern [45] is being used in some workflow systems [46, 29].

3 Background

As part of this thesis, we have extended the JOpera workflow system [94] so that it can support data streams. This chapter first provides an introduction to its workflow meta-model (language). We introduce the constructs to define a workflow model together with a visual notation and specify how to execute such a model. Then, we describe the architecture and give details of the functionality of JOpera.

3.1 Process modeling

The main concept in a workflow model is the *process*. A process is defined by a directed graph where nodes represent tasks and edges represent dependencies between tasks [76]. A *task* is a step to be executed as part of a process. Each task specifies how it is executed, more precisely, which service to invoke. Services can be external, like Web services [3] or databases, or internal, such as XSL Transformations [120]. A task may have several input and output parameters. Input parameters receive data to be used for the task execution and output parameters hold the result of the execution.

A task may also invoke another process. Such a nested process is then called a *subprocess*. A subprocess behaves like a service which is invoked by the task. The task reflects the current state of the subprocess and finishes its execution when the subprocess terminates. Thus, subprocesses allow the modularization of complex processes as procedures do for procedural programming languages. To be able to invoke a process in a function like manner, processes also have input and output parameters as tasks do.

In the visual notation, tasks are represented by rectangular boxes (Figure 3.1). Atomic tasks which directly invoke services are drawn with a single border and complex tasks which invoke a process have a double border. Rounded boxes are used to indicate input and output parameters and are attached to the box of the task they belong to. The orientation of the arrow connecting a parameter to its task determines whether the parameter is for input or output.

The order in which the tasks are executed (control flow) and the means to provide input to a task (data flow) is explained in the following.

3.1.1 Control flow

The control flow of a process describes the partial order in which tasks are executed. A *control flow dependency* from task A to task B specifies that B may not execute before A been executed. According to this ordering, every task has a (possibly

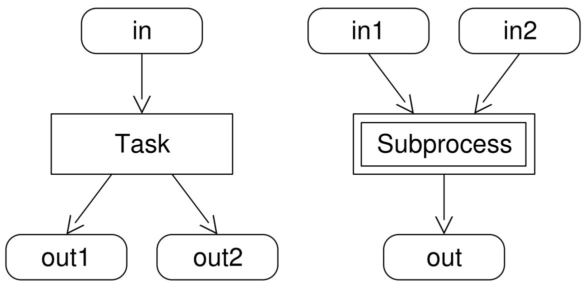


Figure 3.1: Visual notation for atomic and complex tasks with input and output parameters. Parameters are connected to their task with hollow arrows.

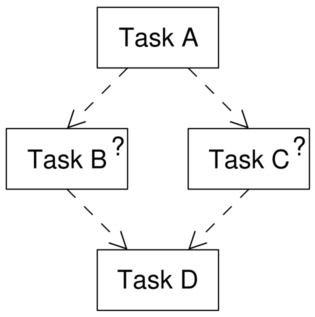


Figure 3.2: Control flow notation. The tasks are partially ordered by control flow dependencies (dashed arrows). Task B and Task C may execute in parallel and have a non-trivial data-condition which is indicated with a question mark. Parameters are not shown in the control flow notation.

empty) set of immediate *predecessors* and *successors*. Correspondingly, a dependency has a predecessor task and a successor task. Figure 3.2 shows the control flow notation. A control flow dependency is represented by a dashed arrow from the predecessor to the successor task.

When a task is executed it can either terminate successfully or it can fail. In order to handle both situations, every control flow dependency is annotated with a predicate. The predicate is satisfied if the predecessor task has reached a certain execution state. Examples of such states are: *Finished* for successful termination and *Failed* if an error occurred[1]. Only if the predicate is satisfied, the control flow dependency becomes *enabled* and control is passed along the dependency to the successor task. In most cases, a *Finished*-predicate will be used which requires the successful termination of a task. If the failure of a task should be explicitly dealt with, an *exception handler* task can be attached using a *Failed*-dependency.

Every task has a *starting condition* which consists of an *activator* and a *data-condition*. The activator is a Boolean expression referencing the incoming control flow dependencies. The expression describes how control arriving from multiple

[1]The complete list of execution states is introduced in Section 3.2

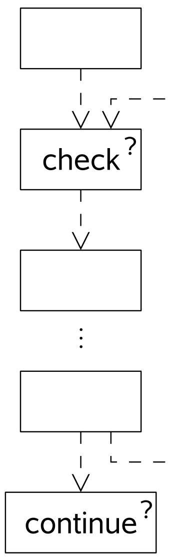


Figure 3.3: Simple loop defined by a control flow cycle.

predecessor tasks is synchronized. Normally, the expression is a conjunction (AND) which requires all predecessors to have terminated. Other synchronization patterns can be employed by specifying the appropriate expression. If at least one predecessor must have terminated, then the control flow dependencies can be combined with OR. For instance, the activator for Task D in Figure 3.2 would be `Finished(Task B) OR Finished(Task C)`.

Conditional branches

The data-condition of a task is a Boolean expression referencing arbitrary parameters of tasks in the process, usually the input parameters of the task. A task is only executed if its data-condition is satisfied. Otherwise, the task is skipped. Conditional branching can easily be implemented using appropriately specified data-conditions. A task which has a non-trivial data-condition, i.e., other than "true", carries a question mark as a visual indicator (Figure 3.2).

Loops

A loop is defined by introducing a cycle in the control flow graph: the first task in the loop depends on the last task in the loop in order to possibly re-execute after each loop iteration. The activator of the first task must be a disjunction (OR) of the incoming dependencies so that the loop can be entered from the predecessor of the first task or from the last task in the loop. The loop condition is encoded in the data-condition of the first task and the task following the loop so either of these tasks gets executed after a loop iteration. Figure 3.3 shows an example of a loop.

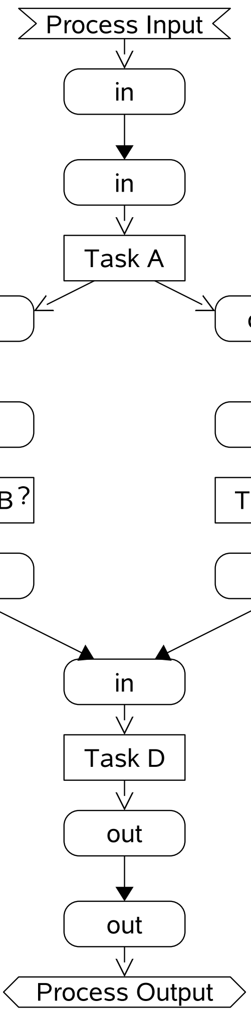


Figure 3.4: Data flow notation. Parameters (rounded boxes) are connected by data flow bindings (filled arrows).

3.1.2 Data flow

The data flow of a process specifies how data is passed between tasks, i.e., how results from one task execution are passed as input to another execution. A *data flow binding* connects an output parameter with an input parameter of two respective tasks. After a task has been executed, data is copied from its output parameters to input parameters of other tasks according to the data flow graph. In Figure 3.4 the data flow notation is shown.

Before a task can execute, its input data must be available in its input parameters. Therefore, a task must not execute until all its predecessors have terminated and the corresponding data has been copied to its input parameters. Consequently, a data flow binding implies a control flow dependency between the corresponding tasks to ensure the correct execution order and the availability of the input data.

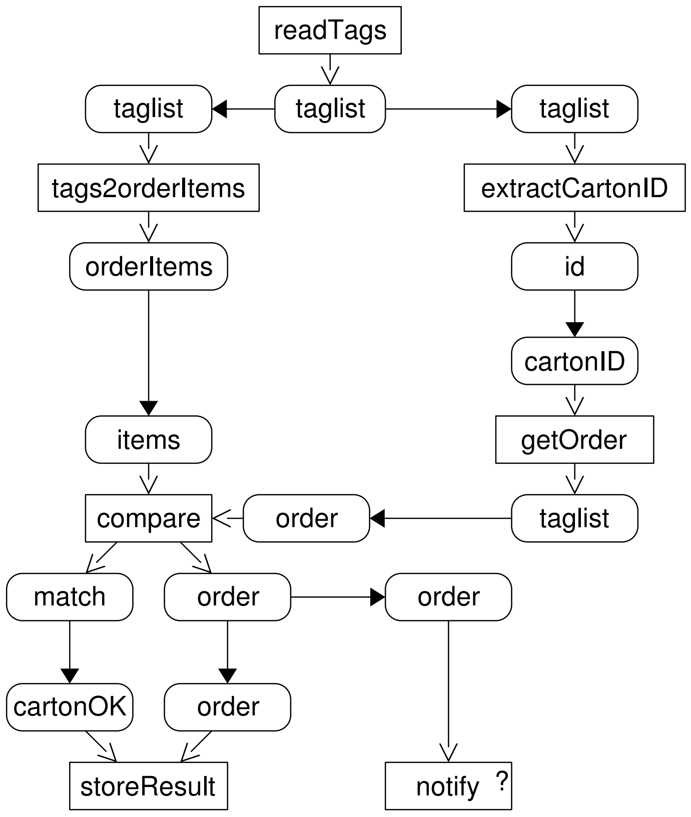


Figure 3.5: Data flow of the order verification process.

Example 3.1: Order verification process

This example demonstrates how to model a process using the constructs introduced in this section. The same process will later be used to show how a process is translated to executable code. The context of the process is as follows. We assume a warehouse where orders from customers are commissioned and shipped. The various items for an order are picked manually and placed into a carton. The carton is then labeled and shipped to the customer. The challenge is to make sure that the packages which leave the warehouse contain exactly the items which were ordered.

This quality control is done by the process discussed in this example. Before a carton may leave the warehouse its contents are identified and compared with the corresponding order which is stored in a database. To do this in a fully automated way, all items and cartons are labeled with RFID tags (radio-frequency identification, [73]). Figures 3.5 and 3.6 show the individual steps of the verification process. For every carton to be checked a process instance is created.

The first task in the process, readTags, retrieves an RFID reading of the carton. The result is a list of unique identities of the RFID tags on and in the carton. This tag list is made available to the rest of the process in an output parameter.

Since RFID tags carry serial numbers without direct relation to labeled item, the tag identifiers need to be mapped to article numbers. The tags2orderItems queries

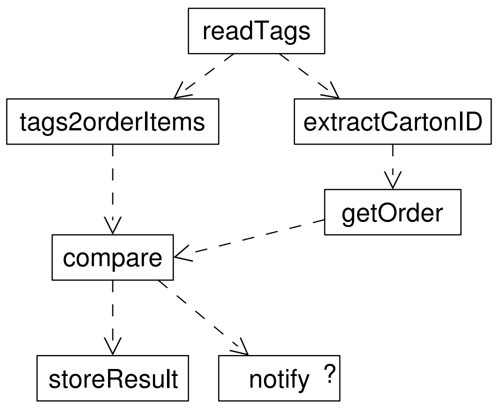


Figure 3.6: Control flow of the order verification process.

a database to accomplish the translation. This results in a list of items and the corresponding quantities.

In order to compare the obtained list of items with the actual order, the current order needs to be identified. The extractCartonID task locates the RFID tag of the carton amongst the list of tags. This can be done based on certain bits in the tag number which identifies the category of a tag (pallet, case, item, etc.). Using the tag of the case the order can be retrieved from a database (getOrder). The order contains a list of items to be shipped which can be compared with the actual contents of the carton. The identification of the case contents and the retrieval of the order from the database can be done in parallel, as there is only a dependency between the extractCartonID and the getOrder tasks.

The control flow of the two branches is merged by the compare task. This task takes the case contents and the order as input and determines whether these match. Thus, the compare must wait for both its predecessors to have completed. The result of the carton verification is recorded in a database (storeResult). If the actual contents of the case deviate from what is specified in the order, then the notify task sends a message (to a human or to another part of the system) in order to prevent the case from being shipped. This conditional execution is achieved through specifying an appropriate data-condition over the match output parameter of compare, e.g., `compare.match = 'ok'`.

3.2 Process execution

To be executed, a process is instantiated. A *process instance* contains the state necessary for the execution of the process [47]. During its lifetime, an instance goes through a set of *states* (Figure 3.7). These states are used to track the progress of individual tasks and of the entire process. Upon process instantiation a task is *Initial*. After being started, a task stays in the *Running* state until its execution terminates. After successful execution, the task is marked as *Finished*. If the ex-

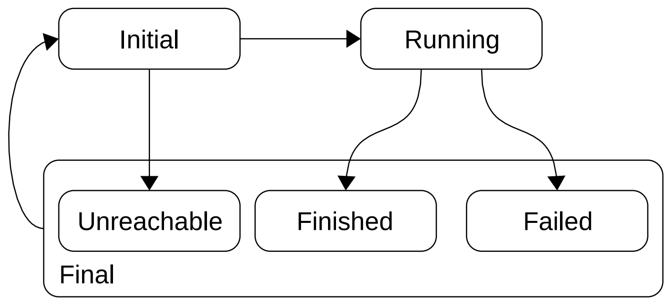


Figure 3.7: Finite state machine for a process or task instance.

ecution of a task fails for some reason, this is indicated with the *Failed* state. A task which, instead of being executed, has been skipped is marked as *Unreachable*. The *Finished*, *Failed*, and *Unreachable* states are referred to as *final states*. The finite state machine of a task is resettable, i.e., after reaching a final state, a task can be reset to the *Initial* state. This is necessary in order to support the repeated execution of tasks found within loops.

When a process instance is created, all of its *start tasks* (tasks without predecessors) are started and the process instance goes to the *Running* state. The task dependencies specify how the process execution continues from there on. Once all tasks haven been executed or skipped, the process instance terminates too.

3.2.1 Navigation

When a task terminates, the other tasks in the process are checked to determine whether they should be executed. This operation is called *navigation*. We assume there is no concurrency involved in the navigation of a process instance. During each navigation step, the state of the instance is only modified by the navigation algorithm. Any events occurring during navigation are buffered and dealt with in a later navigation step.

As described, the prerequisite for the execution of a task is encoded in its starting condition which is specified as an activator and a data-condition. When a task is in the *Initial* state this means that it has not yet been executed (Figure 3.7). For every task which is *Initial*, the navigation algorithm evaluates the corresponding starting condition against the current state of the process instance. First, the activator is checked. If it is satisfied, then control has reached the task and the data-condition determines if the task should be started. If this is the case, an execution request is submitted (Section 3.3.1) and the task is marked as *Running*. Before starting the task, data is copied to its input parameters as specified by the data flow bindings[2].

[2]See Section 3.3.2 for details.

3.2.2 Process termination

The execution state of the overall process instance is derived from the contained tasks. A process instance stays *Running* as long as at least one task is *Running*. When all tasks have terminated, the process instance terminates too. The termination state of the process instance is determined by the tasks. Mostly, all tasks reach the *Finished* state or are skipped and marked as *Unreachable*, in which case the process instance is considered *Finished* to indicate a successful execution. However, if a task fails and there is no other task to handle the exception (through a *Failed*-dependency), the failure is propagated to the process instance by setting its state to *Failed* as well. Thus, a process instance terminates *implicitly*. The alternative approach would be to use a special task to stop the process *explicitly* as is done in WS-BPEL [89] or UML activity diagrams [90].

3.2.3 Dead path elimination

Dead path elimination is the process of identifying and skipping parts of a process which will never be executed [76]. Tasks which will never be executed are marked as *Unreachable*. Detecting unreachable tasks is necessary in order to support the implicit termination of process instances.

The navigation algorithm detects two situations in which a task is skipped. The first case is when the control flow in the process never reaches the task. Consider, e.g., a task T which has a *Finished* dependency on another task P (the predecessor). If P fails, it will change to the *Failed* state and never become *Finished*. Thus, the activator of T will never be satisfied. In order to detect such situations a *deactivator* is used and evaluated if the activator is not satisfied. The deactivator expression can be automatically derived from the activator, using De Morgan's theorem, and is satisfied whenever a predecessor task reaches a state which prevents the activator from becoming satisfied in the future.

The second situation is encountered when the activator is satisfied (control has reached the task) but the data-condition is not satisfied. In this case the data-condition directly specifies that the task should be skipped. Thus, it is also marked as *Unreachable*.

3.2.4 Loops

The execution semantics described so far do not support loops. The reason is that the starting condition of a task is only evaluated when the task is in the *Initial* state. However, after the first iteration of a loop, the contained tasks have been executed or skipped and are in a final state. Before these tasks can be re-executed, their state-machines have to be reset to the *Initial* state. This is achieved as follows. When a task terminates, all its immediate successors are *reset* by putting them in the *Initial* state. Thus, the successors' starting conditions are re-evaluated by the navigation algorithm and the successors can potentially be re-executed as part of a

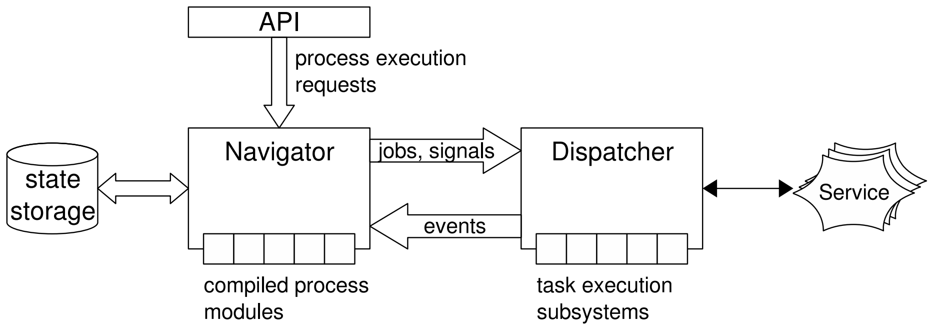


Figure 3.8: Architecture of the JOpera runtime system.

new loop iteration. In order to not interfere with active task executions a task is not reset if it is still *Running*.

With regard to dead path elimination, it should be noted that a “dead path” inside a loop is only temporarily “dead”. The tasks on the path might be executed in the next loop iteration when their starting conditions are re-evaluated.

3.3 JOpera

This section describes JOpera [94], the process engine which was extended as part of this thesis. First, we describe the architecture of the runtime *kernel* which supports the execution of processes. Based on the system architecture, we then give details on how process models are compiled into executable code. Such a module is linked to the kernel to execute instances of a certain process. Finally we present JOpera’s process development environment for Eclipse.

3.3.1 Runtime system architecture

The runtime system of JOpera (Figure 3.8) supports the storage of process instances (*state storage*), the execution of navigation code on process instances (*navigator*) and the execution of tasks through the invocation of services (*dispatcher*).

The stateful system components (state storage and queues) are transactional. The navigator uses this to bracket the operations of every navigation step with a transaction. In case of a system failure, this allows the system to recover to a consistent state.

Navigator

The navigator component is responsible for running the navigation algorithm on active process instances. Whenever a task of a process instance finishes, this event is handled by the navigator. To do so, the module containing the compiled navigation algorithm for this particular process is instructed to evaluate the current state of the process instance. If necessary, the process module is loaded first. When the module determines that some task is ready to be executed, it sets up a corresponding *task execution job* and submits it to the dispatcher. Since executing a navigation algorithm on the state of a process instance and submitting task execution jobs are very fast operations, the navigator executes in a single thread. An advantage of the single-threaded approach is that no concurrency is involved in the navigation of process instances which relieves the navigator of properly synchronizing multiple threads.

In the remainder of the thesis, when we speak of the navigator we always also refer to the compiled navigation algorithm for the process currently being navigated.

State storage

The state storage component stores the runtime information pertaining to all process instances. The data in the storage is accessed exclusively by the navigator. Every process and task instance in the storage is assigned a unique *task instance identifier* (TID). A TID is a tuple (*process*, *instance*, *task*) where *process* is the unique name of a process[3], *instance* is an identifier unique among all instances of the process, and *task* is a name unique among the tasks of the process. The organization of the state storage reflects the process model. Data is addressed with a tuple (*TID*, *category*, *parameter*) where *TID* is a task instance identifier, *category* is the type of parameter (input, output, system), and *parameter* is the name of a task parameter. Thus, the data in the state storage is addressed at the granularity of a single parameter. Data such as the state of a task is mapped to parameters of a special "system" category. With this approach arbitrary runtime state information about a process can be stored in the described structure.

Dispatcher

The dispatcher component handles task execution jobs which have been submitted by the navigator. Such a job specifies the type of service to invoke and parameters for controlling the invocation (e.g., service location, arguments, timeout,...). The dispatcher itself does not execute the job, it rather has a set of *task execution subsystems* which are able to invoke a particular type of service (e.g., Web services, databases). Based on the type of service, the dispatcher delegates the job execution to the appropriate subsystem. The dispatcher uses a pool of threads to invoke the subsystems. This allows the runtime system to parallelize service invocations which yields a higher throughput.

Task execution subsystems

The subsystem takes care of communicating with the service using the correct protocol to accomplish the service invocation. It sets up a service invocation request

[3]The name of a process can be further divided into a package name, a local name and a version identifier.

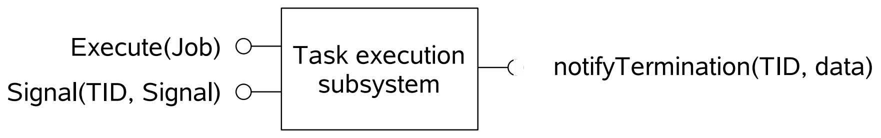


Figure 3.9: Task execution subsystem interface and API.

using the information provided in the job. Input parameters are marshaled in a service type specific manner. After submitting the request to the service location specified in the job description, the subsystem waits for the reply from the service. From the reply it extracts the output parameters and makes them available to the process. The navigator is then notified about the job termination.

Standard functions provided by the subsystem allow the runtime system to control the execution of a job (Figure 3.9). Through the `Execute` method the subsystem receives a job to be executed. This method is kept very general in that it passes just a single job description to the subsystem. The job contains a description of the service invocation to be undertaken, based on parameters which are specific to the subsystem.

To control a job after it has been started, signals can be sent to the subsystem in charge using the `Signal` method. This method takes as parameters the identifier of the job to be manipulated and the signal type. A signal is always targeted at a TID and is routed by the runtime system to the subsystem executing the corresponding job. The supported signal types are *Suspend* (suspend the job), *Resume* (resume a suspended job) and *Kill* (abort the job). Signals are usually initiated by the user to influence the execution of process instances or individual tasks. A subsystem may choose not to support certain signal types which do not fit with the service type (e.g., remote Web services cannot be suspended in general). New signal types can easily be added to the system. The prerequisite for successfully signalling a job is that the receiving subsystem knows how to handle the signal type. The rest of the system is agnostic with respect to the semantics of a signal.

After a job has been executed, a notification is sent to the navigator together with the result of the execution. The subsystem API provides the `notifyTermination` method for this.

Queues

The communication between the navigator and dispatcher components takes place through queues. The navigator sends task execution jobs and signals to the dispatcher and the dispatcher notifies the navigator of the progress of job executions. This allows the two components to execute asynchronously. The navigator can issue multiple jobs independently of how long it takes to execute such jobs. The dispatcher sends a notification whenever a job and thus the corresponding task has finished to allow the navigator to continue with the execution of the process by possibly starting other tasks.

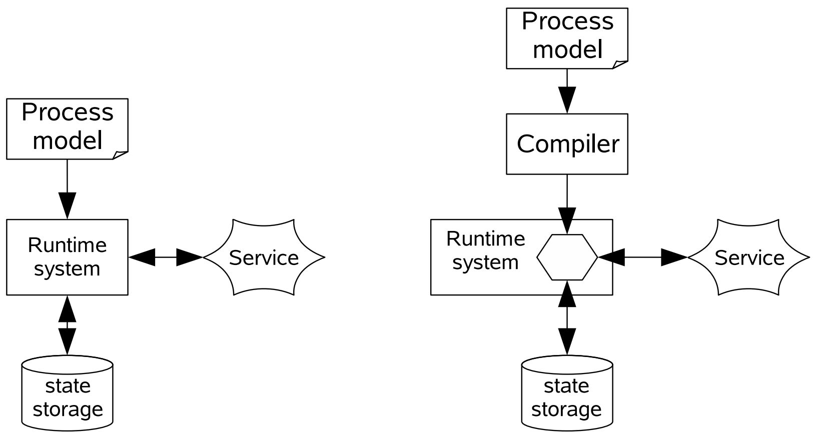


Figure 3.10: Left: process models are interpreted by the runtime system. Right: process models are compiled into executable modules which are loaded into the system at runtime for navigating process instances.

Flexible deployment

The state storage and the navigator-dispatcher queues can be implemented with different technologies. As an example, the state storage can be implemented in main memory or by using a relational database together with the appropriate mapping from the state storage model to the relational model. The flexibility of the system component implementation allows the system to be deployed in a variety of ways. In a monolithic deployment, all components run on the same machine. In the other extreme case, several navigators and dispatchers are each deployed on a different machine. These distributed components are connected by the state storage and queues which themselves can be centralized or distributed.

3.3.2 Compiler

There are two approaches to executing process instances: by interpretation of the process model or by using a compiled version of the process. The JOpera system takes the latter approach and contains a compiler to convert process models into executable modules. Before it can be executed, a process model is compiled into an executable module which is then loaded into the runtime system for execution (Figure 3.10). This module contains a compiled version of the navigation algorithm tailored to the process and is used to manage all the instances of the process. Thus, the compiler contains the semantics of the workflow language.

The compiler is not part of the runtime system. However, the compiler could be included in order to allow for just-in-time compilation of process models. The compiler consists of three parts: a control flow compiler, a data flow compiler and a setup compiler. These parts of the compiler are described in the following. They produce source code for the Java programming language [50] which is compiled into the byte code of a single class for dynamically loading into the runtime system.

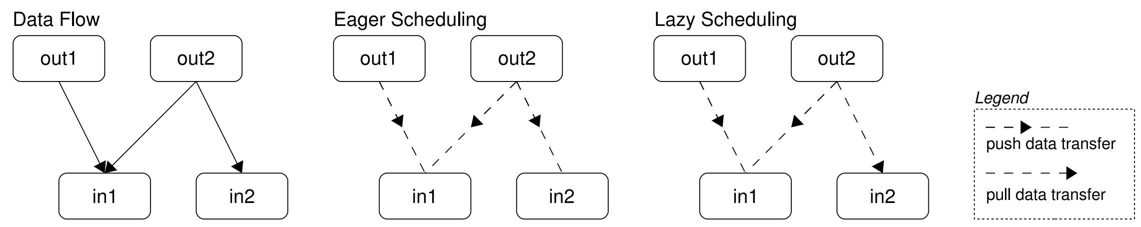


Figure 3.11: A set of data flow bindings (left) can be scheduled eagerly (center) and lazily (right). In the case of the lazy approach, the transfers involved in the data flow merge are eager to ensure consistent semantics.

Setup compiler

The setup compiler generates the part of the process module which initializes a process instance before it is executed. It sets the state machine of the process and its tasks to the *Initial* state and stores the default value for parameters where such has been defined. The runtime system creates process instances by calling this initialization code.

Data flow compiler

The data flow compiler analyses the data flow of the process and schedules data transfers between parameters. For each data flow binding in the process, a corresponding data transfer copies the data from the output parameter to the input parameter. We define the data flow bindings which connect to the input parameters of a task as *incoming bindings* and the data flow bindings which connect to the output parameters of a task as *outgoing bindings*.

Data transfers can take place before or after a task has been executed. Figure 3.11 compares the two approaches on which we give more details in the following. In the simplest case, data transfers are scheduled eagerly, i.e., immediately *after* the corresponding task has executed. Thus, when a task terminates, the data transfers for its outgoing data flow bindings are carried out. This way, the output of a task is *pushed* to other tasks as soon as possible. The advantage of this approach compared to handling data transfers right before the task execution is the clear semantics for input parameters which have multiple bindings. In this case, the semantics are "last writer wins", which means that the data in the input parameter is determined by the predecessor task which terminates last. The reason is that the data in the parameter is overwritten every time a predecessor task terminates and the corresponding data transfers are executed.

The JOpera data flow compiler schedules data transfers lazily in order to minimize the number of data transfers which have to be performed. This approach is the opposite of the previous one which executes data transfers after the corresponding task's termination. All data transfers are handled at the latest possible point in time when data must be available in a task's input parameters. So, data transfers

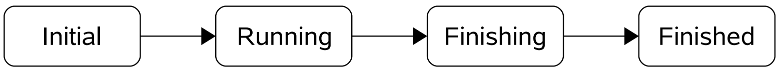


Figure 3.12: Excerpt of the the internal finite state machine used by the compiler.

are scheduled to *pull* data immediately *before* the task execution. This way, if a task is not executed as part of a process instance – because its starting condition is never satisfied –, the associated data transfers can be saved. However, for input parameters with multiple bindings the data transfers cannot be executed lazily because it would not be clear which of the bindings should be used to determine the data for the input parameter. In order to have clear semantics with such parameters, the corresponding data transfers are performed eagerly, i.e., immediately after the corresponding predecessor task has executed. This ensures “last writer wins” semantics for such input parameters as with the above approach.

Control flow compiler

The control flow compiler generates the core part of the navigation algorithm which schedules tasks in a process for execution in an appropriate order. As described in Section 3.2.1, this scheduling can be accomplished by evaluating the starting conditions of all tasks over the state of a process instance. The compiler translates the starting conditions of the tasks in a process into executable code. The generated algorithm is stateless, so it can be used with any instance of the corresponding process. After loading the necessary parts of the process instance state (task states and parameter values) into local variables, the starting condition is evaluated by compiled expressions.

Whenever the starting condition of a task is satisfied, a task execution job is put together and submitted to the task execution component (Section 3.3.1). The job consists of a specification of how the task should be executed (type of service, address of service) as well as the data contained in all input parameters of the task.

Copying data between task parameters is part of the overall navigation algorithm. Therefore, the control flow compiler includes the data transfers which have been scheduled by the data flow compiler in the appropriate locations. Before a task execution job is assembled, the data transfers are executed which have been scheduled to run prior to the task execution. This makes sure that the input parameters contain the data which is needed to execute the task. When a task terminates in the *Finished* state, data transfers are carried out which have been scheduled to run after the task execution. The compiler uses a special intermediate state for this in the finite state machine of a task. Before reaching the *Finished* state a task changes to the *Finishing* state (Figure 3.12). When a task is in the *Finishing* state data transfers scheduled for the task termination are executed. Immediately after the data transfers have been carried out the task leaves the *Finishing* state and becomes *Finished.*

Example 3.2: Process compilation

In this example we show how a process model is compiled to executable Java code. To do so, we use the order verification process from Example 3.1. The generated code interacts directly with the various kernel components (Section 3.3.1).

Before a process can be executed an instance must be created and initialized. This process-specific initialization code is generated by the setup compiler and for the readTags task looks as follows.

```
TID Context_TASK_storeResult = TASK(Context, "storeResult");
SetupSystemBox(Context_TASK_storeResult);
SetupParam(Context_TASK_storeResult, Box.System, Box.Name,
  "storeResult");
SetupParam(Context_TASK_storeResult, Box.System, Box.Type,
Box.Activity);
SetupParam(Context_TASK_storeResult, Box.System, Box.Prog,
  "{cartoncheck}storeResult[1.0]");
```

The `SetupSystemBox` method basically sets the execution state of the task to *Initial*. `SetupParam` is used to set other properties of the task such as its name. Input and output parameters are initialized with default values, in this example no default value has been specified for the `taglist` output parameter. All this information is written to the state storage within the scope of the process instance (specified by the `Context` parameter).

The navigation algorithm is contained in a method which is invoked every time the state of a process instance changes, e.g., when a task execution terminates. This method evaluates the current state of the instance and decides which tasks are ready to start and computes the overall state of the process instance. The method is invoked with the identifier (TID) of the current process instance as a parameter.

```
public void Evaluate(TID Context) throws MemoryException {
```

Since the navigation algorithm is stateless, the state of the process instance must be loaded into local variables initially. This allows the compiled code to be executed directly on the state. The most important part of the state is the execution states of the tasks and of the process (only one task is shown as example).

```
State State_PROC = Memory.getState(Context_PROC);

State State_compare = Memory.getState(Context_TASK_compare);
```

When the navigation method is invoked for the first time, the process instance is still in the *Initial* state. This state is handled specially by the method – in the *Initial* state the actual execution of the process instance is started by activating the start tasks of the process.

```
if (State_PROC == State.INITIAL) {
```

In our example process, there is one task without predecessors, readTags, which needs to be started.

```
 Exec.Start(Context_TASK_readTags, InputParams, SystemInputParams);
```

After starting the start tasks, the process instance can be marked as *Running*.

```
Memory.setState(Context_PROC, State.RUNNING);
```

When the process is in the *Running* state, "normal" navigation takes place, i.e., starting conditions are evaluated, tasks are started and data is transferred between tasks. As an example we look at the life cycle of the compare task. This task represents a control flow merge. It synchronizes control coming from tags2orderItems and getOrder by requiring both tasks to have successfully terminated. Since starting conditions are evaluated only in the *Initial* state, this is checked first.

```
if (State_compare == State.INITIAL) {
 if ((State_getOrder == State.FINISHED)
   && (State_tags2orderItems == State.FINISHED)) {
```

As compare has no data-condition specified, the satisfied activator is sufficient to start the task. Before it can be started, however, its input parameters must receive data according to the data flow. Therefore, data transfers are executed to copy the contents of the output parameters of tags2orderItems and getOrder.

```
  Memory.Copy(MakeAddress(Context_TASK_getOrder,
    Box.Output, "order"), MakeAddress(
    Context_TASK_compare, Box.Input, "order"));
  Memory.Copy(MakeAddress(Context_TASK_tags2orderItems,
    Box.Output, "orderItems"), MakeAddress(
    Context_TASK_compare, Box.Input, "items"));
```

Then, a task execution request can be submitted using the current values of the input parameters.

```
  InputParams.put("items", Memory.Load(MakeAddress(
    Context_TASK_compare, Box.Input, "items")));
  InputParams.put("order", Memory.Load(MakeAddress(
    Context_TASK_compare, Box.Input, "order")));

  Exec.Start(Context_TASK_compare, InputParams,
    SystemInputParams);
```

On successful termination a task changes to the *Finishing* state where the result of the execution is persisted in the output parameters. Then, the task is immediately marked as *Finished* which allows other tasks to be started in response.

```
if (State_compare == State.FINISHING) {
 Memory.Store(MakeAddress(Context_TASK_compare, Box.Output,
   "match"), (Serializable) Results.get("match"));
 Memory.Store(MakeAddress(Context_TASK_compare, Box.Output,
   "order"), (Serializable) Results.get("order"));

   Memory.setState(Context_TASK_compare,
     State.FINISHED);
   State_compare = State.FINISHED;
```

The successful termination of compare allows control to reach the notify task. This task, however, has a data-condition which lets it start only if a mismatch was detected by compare.

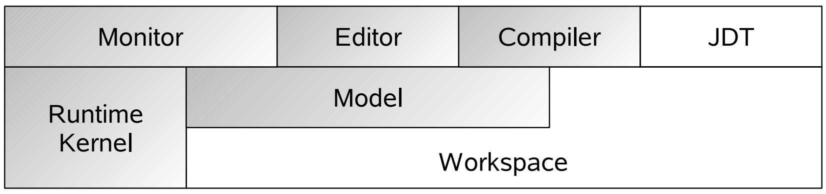


Figure 3.13: Architecture of the JOpera IDE.

```
String p_compare_match = Memory
  .LoadStr(MakeAddress(Context_TASK_compare,
    Box.Output, "match"));
if (p_compare_match.equals("ok")) {
```

If the data-condition is not satisfied, the task is marked as skipped.

```
Memory.setState(Context_TASK_notify,
  State.UNREACHABLE);
State_notify = State.UNREACHABLE;
```

Finally, in every call to the `Evaluate` method, the execution state of the entire process instance is derived from the task states. As an example we show the detection of failures which are not handled by any exception handler task.

```
if ((State_readTags == State.FAILED)
  || (State_extractCartonID == State.FAILED)
  || (State_compare == State.FAILED)
  || (State_tags2orderItems == State.FAILED)
  || (State_getOrder == State.FAILED)
  || (State_storeResult == State.FAILED)
  || (State_notify == State.FAILED)) {
 if (State_PROC != State.FAILED)
  Memory.setState(Context_PROC, State.FAILED);
}
```

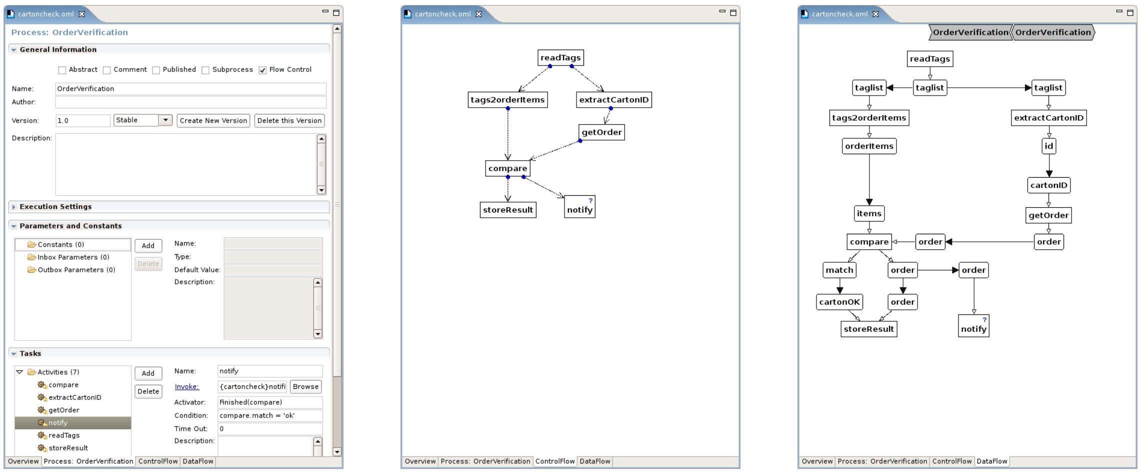
3.3.3 JOpera IDE

This section describes *JOpera for Eclipse* [93], JOpera's integrated development environment (IDE) for rapid prototyping and development of workflow processes. The IDE is implemented on the Eclipse platform [38] as a set of plug-ins.

Architecture

Figure 3.13 shows the architecture of the IDE. It integrates with the Eclipse workspace (Eclipse's virtual file system) and the Eclipse Java Development Tools (JDT) [36]. The IDE features an embedded runtime kernel which allows to test processes during development without having to deploy them on a remote engine.

Following the Eclipse approach, editing, compilation and execution of processes is loosely coupled. The development evolves around process models which are stored in files in the workspace. The *model* component manages the memory representation of



cartoncheck.oml

Process: OrderVerification

General Information

[ ] Abstract [ ] Comment [ ] Published [ ] Subprocess [✓ Flow Control]

Name: OrderVerification

Author:

Version: 1.0 Stable

Description:

Execution Settings

Parameters and Constants

- Constants (0)
- Inbox Parameters (0)
- Outbox Parameters (0)

Name:

Type:

Default Value:

Description:

Tasks

- Activities (7)
  - compare
  - extractCartonID
  - getOrder
  - notify
  - readTags
  - storeResult

Name: notify

Invoke: {cartoncheck}notifi

Activator: Finished(compare)

Condition: compare.match = 'ok'

Time Out: 0

Description:

Process Details Control flow Data flow

Figure 3.14: Different views of a process model in the editor.

these models and provides it to other components. The editor displays and modifies the model and may commit changes to the workspace. The compiler incrementally converts new or updated process models to Java bytecode. The monitor finally uses the process model to visualize the runtime state of process instances in a graphical manner similar to the editor.

Model

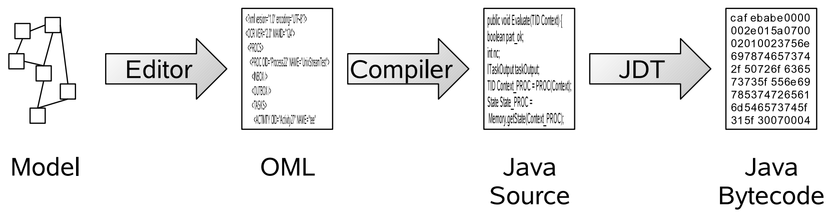
Process models are persisted using the Opera Modeling Language (OML) [94], an XML-based language. An OML file can hold multiple such models. The models are organized in projects and are visible across files in the same project. Namespaces are used to avoid name clashes between different OML files.

Editor

The editor is used to visually design process models through manipulation of the control flow and data flow graphs (Figure 3.14). The editing of the control flow and the data flow is separated into different views, allowing the user to focus on either part of the process model. A third structured non-graphical view gives an alternative access to most of the process details, in particular all the non-graphical properties. While a process model is being edited, the editor keeps the different views of the model consistent. Specifically, for every data flow binding a corresponding control flow dependency is created and the control flow graph is synchronized with the activators of tasks.

Compiler

The JOpera compiler translates process models into Java code as described in Section 3.3.2. It is triggered by changes to OML files and generates or updates the



```
<?xml version="1.0" encoding="UTF-8"?>
<OCR VER="2.0" MAXID="124">
 <PROCS>
  <PROC OID="Process22" NAME="UnixStreamTest">
   <INBOX />
   <OUTBOX />
   <TASKS>
    <ACTIVITY OID="Activity27" NAME="tee"
```

```
public void Evaluate(TID Context) {
boolean part_ok;
int nc;
ITaskOutput taskOutput;
TID Context_PROC = PROC(Context);
State State_PROC =
 Memory.getState(Context_PROC);
```

```
caf ebabe0000
002e015a0700
02010023756e
697874657374
2f 50726f 6365
73735f 556e69
785374726561
6d546573745f
315f 30070004
```

Figure 3.15: Transformation chain for compiling a process model to executable code.

corresponding Java source files. Similarly, these files are then picked up by the JDT compiler to generate Java bytecode (Figure 3.15). After a process model has been fully compiled, the corresponding bytecode is automatically deployed in the embedded kernel. Thus, the newest version of a process model is always available for testing.

Monitor

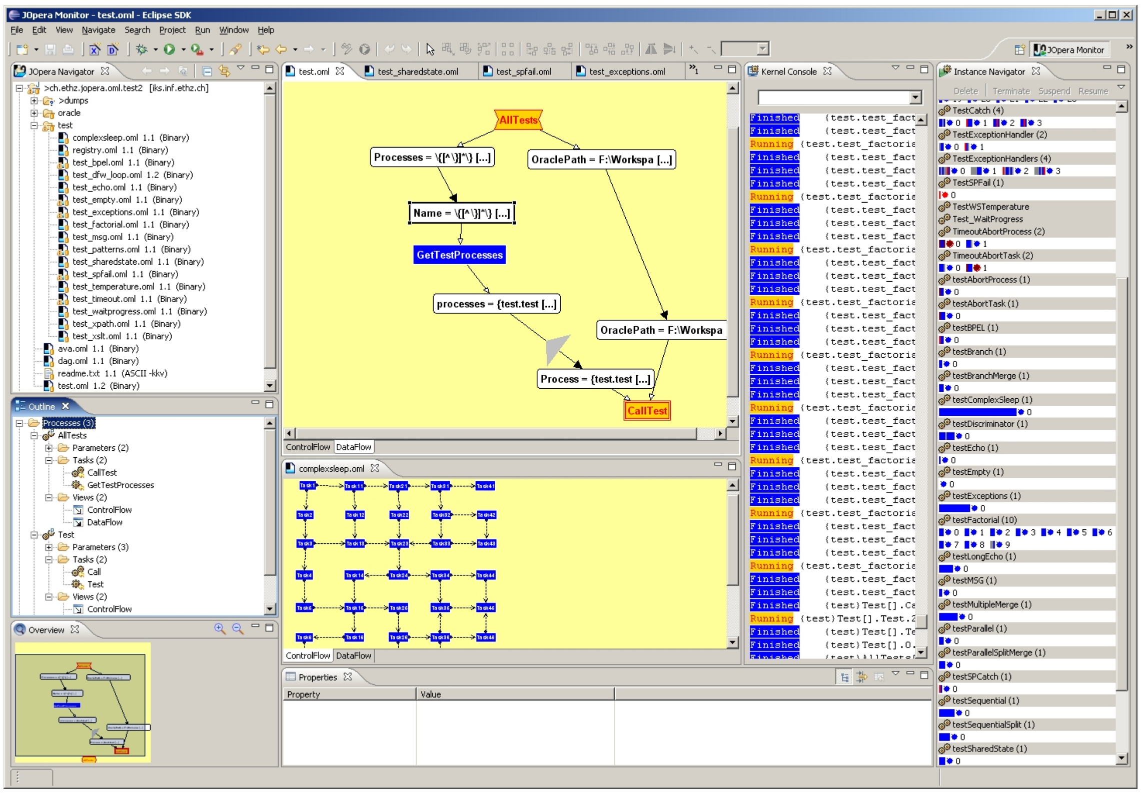
The monitor tool is the main graphical interface for the embedded kernel. After starting processes, the state of process and task instances (including important parameters) can be inspected in real-time. The monitor uses the same graphical representation for a process instance as the editor uses for the process model, augmented with runtime information from the state storage (Figure 3.16). The execution of process and task instances can be controlled by sending signals to them.

3.4 Discussion

This chapter has introduced the modeling and execution of workflow processes. At the core of our approach to model processes is the concept of tasks with starting conditions. This is the essence of the activity-based paradigm which is widely used by workflow systems (Section 2.5.1). Therefore, we believe that our process meta-model is sufficiently generic so that the results of this thesis are universally applicable to activity-based systems.

A different way of looking at the problem of process modeling is to put the focus on the states of a system. Conceptually, there are two main differences between state-based and activity-based approaches. First, a state-based workflow process remains in a state until a certain event occurs. While in this state, the process might be idle, i.e., no action is being executed. In an activity-based process, at any time at least one activity is running. Second, activity-based approaches usually do not support the notion of explicit events which can be used to model events external to the system.

At first sight, this might seem like a big discrepancy. However, this is not the case: an activity-based process can easily be converted into a state-based representation. An activity can be decomposed into an action (starting the activity), a state (while



```
>ch.ethz.jopera.oml.test2  [iks.inf.ethz.ch]
  >dumps
  oracle
  test
    complexsleep.oml  1.1  (Binary)
    registry.oml  1.1  (Binary)
    test_bpel.oml  1.1  (Binary)
    test_dfw_loop.oml  1.2  (Binary)
    test_echo.oml  1.1  (Binary)
    test_empty.oml  1.1  (Binary)
    test_exceptions.oml  1.1  (Binary)
    test_factorial.oml  1.1  (Binary)
    test_msg.oml  1.1  (Binary)
    test_patterns.oml  1.1  (Binary)
    test_sharedstate.oml  1.1  (Binary)
    test_spfail.oml  1.1  (Binary)
    test_temperature.oml  1.1  (Binary)
    test_timeout.oml  1.1  (Binary)
    test_waitprogress.oml  1.1  (Binary)
    test_xpath.oml  1.1  (Binary)
    test_xslt.oml  1.1  (Binary)
  ava.oml  1.1  (Binary)
  dag.oml  1.1  (Binary)
  readme.txt  1.1  (ASCII -kkv)
  test.oml  1.2  (Binary)
```

```
Processes (3)
  AllTests
    Parameters (2)
    Tasks (2)
      CallTest
      GetTestProcesses
    Views (2)
      ControlFlow
      DataFlow
  Test
    Parameters (3)
    Tasks (2)
      Call
      Test
    Views (2)
      ControlFlow
```

```
Finished    {test.test_fact
Finished    {test.test_fact
Running  {test.test_factoria
Finished    {test.test_fact
Finished    {test.test_fact
Finished    {test.test_fact
Running  {test.test_factoria
Finished    {test.test_fact
Finished    {test.test_fact
Finished    {test.test_fact
Running  {test.test_factoria
Finished    {test.test_fact
Finished    {test.test_fact
Finished    {test.test_fact
Running  {test.test_factoria
Finished    {test.test_fact
Finished    {test.test_fact
Finished    {test.test_fact
Running  {test.test_factoria
Finished    {test.test_fact
Finished    {test.test_fact
Finished    {test.test_fact
Running  {test.test_factoria
Finished    {test.test_fact
Finished    {test.test_fact
Finished    {test.test_fact
Running  {test.test_factoria
Finished    {test.test_fact
Finished    {test.test_fact
Finished    {test.test_fact
Running  {test.test_factoria
Finished    {test.test_fact
Finished    {test.test_fact
Finished    {test.test_fact
Running  {test.test_factoria
Finished    {test.test_fact
Finished    {test.test_fact
Finished    {test}Test[].Ca
Running  {test}Test[].Test.2
Finished    {test}Test[].Te
Finished    {test}Test[].O.
```

```
TestCatch (4)
  0  1  2  3
TestExceptionHandler (2)
  0  1
TestExceptionHandlers (4)
  0  1  2  3
TestSPFail (1)
  0
TestWSTemperature
Test_WaitProgress
TimeoutAbortProcess (2)
  0  1
TimeoutAbortTask (2)
  0  1
testAbortProcess (1)
  0
testAbortTask (1)
  0
testBPEL (1)
  0
testBranch (1)
  0
testBranchMerge (1)
  0
testComplexSleep (1)
  0
testDiscriminator (1)
  0
testEcho (1)
  0
testEmpty (1)
  0
testExceptions (1)
  0
testFactorial (10)
  0  1  2  3  4  5  6
  7  8  9
testLongEcho (1)
  0
testMSG (1)
  0
testMultipleMerge (1)
  0
testParallel (1)
  0
testParallelSplitMerge (1)
  0
testSPCatch (1)
  0
testSequential (1)
  0
testSequentialSplit (1)
  0
testSharedState (1)
  0
```

Figure 3.16: Monitoring process instances.

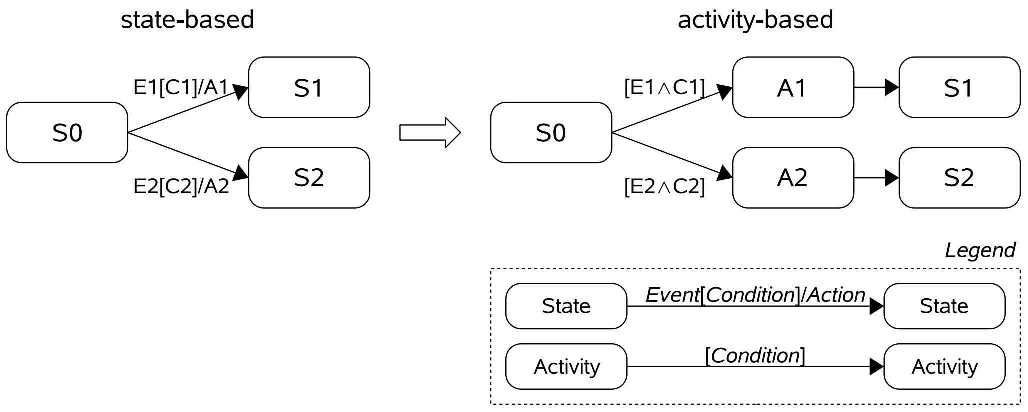


Figure 3.17: Example conversion of a state-based model to the corresponding activity-based model with the same behaviour.

the activity is executing) and an event (when the activity terminates)[4]. Activities thus have an implicit state (*Running* in our case) and implicitly generate an event (termination of the task). Since there is only one event type, it is used by all transitions. The modeling of the transition condition is the same in both approaches.

Vice versa, a state-based model can be implemented with activities. Thereby, every state is modeled by a special wait-activity. When started, this activity waits for the occurrence of one of the events corresponding to the outgoing transitions. The outgoing transitions of the wait-activity correspond to the transitions of the original state. The transition conditions are augmented with a filter which allows the transition to be followed only if the wait-activity terminated due to the event specified for the transition in the state-based model. In case the transition has an associated action, this can be modeled by an additional activity to be executed before reaching the next wait-activity. Figure 3.17 illustrates this conversion using a simple example.

Because of these similarities between activity-based and state-based process modeling, we believe that our results also could be applied – with minor modifications – to state-based workflow systems, which is, however, left as future work.

[4]In fact, this is what a workflow execution system typically does internally to execute an activity (Section 3.3.1)

4 Integration of streams in workflow processes

In this chapter we extend the state-based process model (Chapter 3) with features to integrate streaming services. First, we discuss how to model a streaming data source so it can be included in a process (Section 4.2). We introduce the pipelined execution of tasks over stream elements. The problems arising thereby are discussed in detail (Section 4.3). We then present a minimal but necessary extension to the process execution semantics to support safe pipelining (Section 4.4). Based on the support for safe pipelining, we discuss how to include buffering in a process to improve the performance of a pipeline (Section 4.5). Finally, we discuss the impact of the new semantics (Sections 4.6 and 4.7).

4.1 Motivation

The process meta-model from Chapter 3 assumes that a service which is invoked by a task has a request-response interface. However, there are services which do not follow this interaction pattern but rather proactively produce new information to be consumed by the business process. Examples include RSS [13] feeds listing the latest bids at an auction, result tuples from a data stream management system (DSMS) [21, 23], stock price tickers, tags from an RFID reader etc.

The need for supporting stream based processing goes beyond the increasingly important ability to cope with services that produce data streams. As an example, in a customer support system with a high load, streaming can avoid having to create a process instance for each incoming support request. Instead, a single process is started and the requests are processed in a pipelined manner. Such an approach can have significant advantages in terms of scalability and expressiveness of the business process. In standard process description languages such as WS-BPEL [89], an incoming request message creates a new process instance for the handling of the request. Also, WS-BPEL does not support the pipelined processing of multiple stream elements. This means that the resources available to the system might not be used to full capacity and that an inherent limit on scalability is unnecessarily introduced.

In this chapter we discuss how to extend the state-based business process model from Chapter 3 with the necessary features to integrate streaming data services and combine them with conventional request-response services. In order to support both types of services, there is the constraint that the task model must remain compatible with the request-response interaction style.

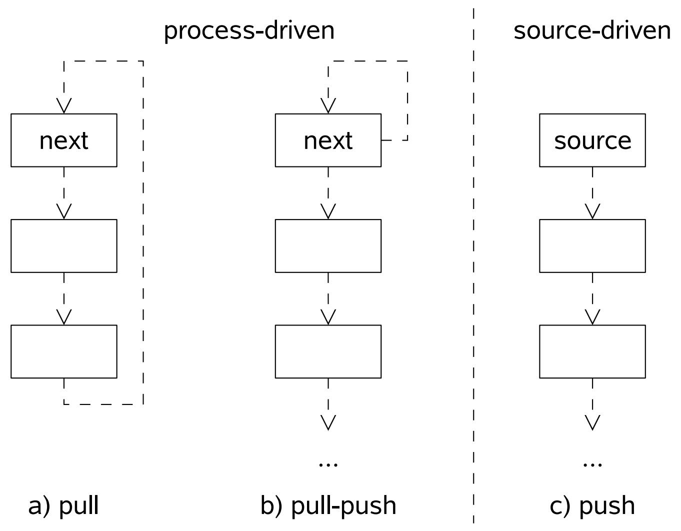


Figure 4.1: Continuously consuming data from a stream source.

4.2 Accessing streams

Before we can process data from a stream source we need to make the data available to the workflow. For discrete data sources (e.g., a database), a process connects to the source with a dedicated task which returns the retrieved data as its output. Thus, the task represents the data source inside the process. In the case of a stream we also represent a source of streaming data as a task in the process. The difficulty in doing so is that the stream source produces data continuously. To read the stream from a process, there are two alternatives: process-driven and source-driven. In the first case the workflow pulls the data from the source whenever it is ready for it. Thus, the workflow cannot be overrun with too much data since it decides when it is ready to receive more. In the second case the source pushes the data to the workflow. The workflow then needs to be ready to receive the data.

Considering the possibility to pull or push the data into the workflow, there are three patterns for using a task to read from the data stream source (Figure 4.1). In all three cases the first task provides the data from the stream source and the other tasks are invoked in sequence to process this data.

4.2.1 Pull

The first case (Figure 4.1a) is a conventional loop in which the last task has a control flow dependency back to the loop's first task. In every iteration the source task is invoked to pull the next item from the stream source. The advantage is that the source task will not output data and start the next task before the entire loop iteration has finished. This guarantees that no task in the loop will be started while it is already running. However, this is also a disadvantage: while one of the tasks in the loop is active all the others are idle. This means that the throughput of the workflow is limited by the length of the entire loop. For some applications this is inefficient as more than one task could be active concurrently.

4.2.2 Pull-Push

The second option (Figure 4.1b) is to make the control flow cycle contain only the source task. Therefore, when the task has been executed it is restarted immediately. At every iteration the tasks following the loop are also restarted in order to process the data. This way, the source task pulls the data from the stream source and pushes it into its successor task.

The advantage of this approach is the possibility of pipelining. We use the term pipelining the same way it is used with processor-architectures [60]. In a processor pipeline all stages are active concurrently, each processing a different instruction. Similarly, in a sequence of tasks, every task is working on a different stream element, allowing the tasks to be active concurrently. When a task has finished processing an element, it receives a new one from its predecessor task. The source task at the head of the sequence constantly provides new stream elements to the task pipeline.

Any path in the control flow graph of a process constitutes a pipeline if multiple tasks on the path are active at the same time. Because the pipeline handles multiple elements of a stream in parallel, the overall throughput in terms of stream elements is higher than in the pull approach. The use of pipelining in workflows does, however, pose some problems as is discussed in Section 4.3.

4.2.3 Push

The third approach (Figure 4.1c) goes even further and makes the loop smaller than one task. The actual loop executes “inside” the source task because the task runs forever (or at least as long as there is data coming from the stream source). This makes it necessary to extend the semantics of a task: in addition to providing output when finishing, a task can also output data while it is running. We call this feature *multiple output*.

To implement the multiple output feature the state machine of a task instance needs to be extended with a new *Outputting* state (Figure 4.2). With this state machine, the task is started as usual by making the transition from the *Initial* state to *Running*. When there is partial output available from the stream source, the task temporarily goes to *Outputting* and then goes back to *Running*. Successor tasks which want to consume this output have an *Outputting* control flow dependency on the source task. If the data stream is not infinite, the source task will eventually become *Finished*.

This approach is very similar to pull-push approach. Both approaches enable pipelining by having the source task pushing data into the successor task. Compared to the pull-push, the push approach models an end-to-end push delivery of the stream data. The advantage is that there is no need to define an explicit control flow loop because the state of the task reflects the state of the stream source. Furthermore, the workflow can deal with the end of the stream explicitly through a

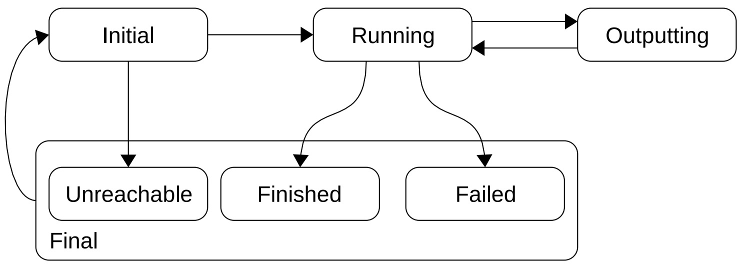


Figure 4.2: Finite state machine for a task instance with multiple output.

task with a *Finished* dependency on the source task. The disadvantage consists of, as with the pull-push approach, the complexity introduced by pipelining.

Because of its advantages in terms of performance and ease of modeling, we choose to pursue the push approach. While this approach requires to extend the model of a task, if this should not be feasible, it is always possible to use the pull-push approach instead without giving up the performance advantage. The increase in performance comes at a certain price. The remainder of the chapter addresses the implications of using pipelines in a process and presents solutions.

Example 4.1: High-volume account creation

This example shows how to apply the multiple-output feature to consume a stream of messages in a process. By using this feature a pipeline is created in the process and the throughput of the process is increased.

At some universities several thousand freshmen enroll every year. When a new student registers, among other things an account needs to be created for accessing the university's computers.

We assume that proper management interfaces are available so that a workflow system can be used to set up new accounts. In order to sustain a high volume of account creation requests, a single process instance is used and requests are streamed through it. This avoids the overhead of creating a process instance for each request and leverages pipelining inside the process. The process picks up requests from a persistent queue at maximum speed and places its replies into a different queue. The throughput of the process is limited by the slowest task in the pipeline. Therefore, the process should consist of a high number of tasks with similar execution times.

Figure 4.3 shows the account creation process. Requests sent to the system contain information on the account to be created and the credentials of the requester. The process picks up a request from the queue and validates it (syntax, authorization). Then, the status of the student is checked against a database. After the checks, the account is created, a home directory installed and permissions on the home directory are set to give the new account access. Finally, a reply indicating the successful account creation is sent to a queue. If any of the steps in the account

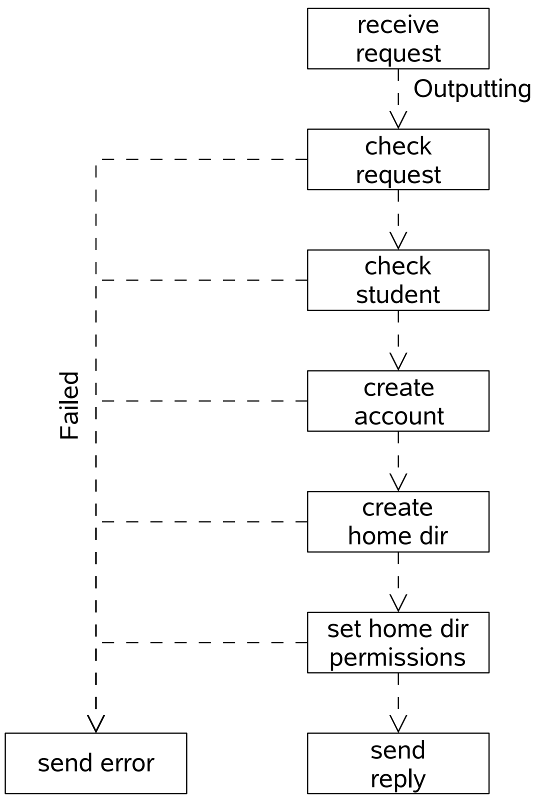


Figure 4.3: Account creation process.

setup fails, an error is sent back to the originator of the request using the send error task.

The second task, check request, has an *Outputting* dependency on the first task which receives incoming requests. This allows the first task to run continuously and push requests into the process. The other tasks have normal *Finished* dependencies as they invoke traditional request-response services.

With a case-driven approach, a process instance would be created for every incoming request. When the number of requests is as large as in this example, it becomes a serious challenge to manage and execute all the process instances in parallel. With the stream-based workflow this scalability problem is solved without having to modify the existing services. These services are still invoked in a request-response manner and do not need to be aware of the pipelined execution.

4.2.4 Navigation with multiple-output tasks

As far as the process navigation is concerned, the *Outputting* state is very similar to the other task states. Tasks which need the partial or intermediate results of a multiple-output task have a corresponding *Outputting* control flow dependency on the task. When the task provides output in the *Outputting* state, the navigation algorithm is run on the process instance so other tasks in the process may be started

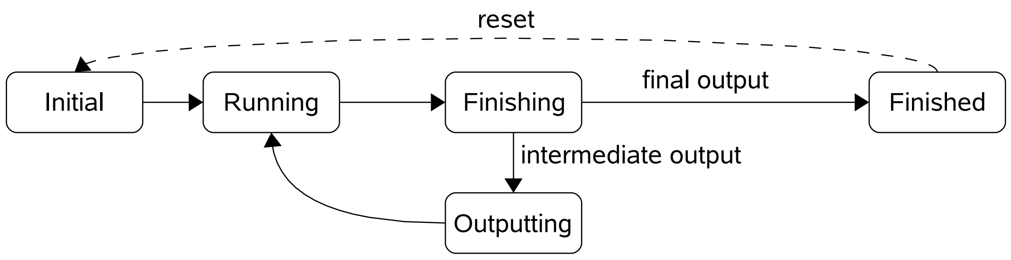


Figure 4.4: Partial state machine for a multiple-output task showing the differences between the *Finished* and *Outputting* states.

with the new data. Then the task changes back to *Running*. This allows the task to provide further results. Once the task has reached a final state, its execution terminates and stays in the final state until its state machine is reset.

The *Outputting* state is very similar to the *Finished* state in that the task in both states provides output and may trigger the execution of other tasks. Therefore, the navigator reuses the *Finishing* state for multiple-output tasks (Figure 4.4). In this temporary state all data transfers are executed which are scheduled to be handled when the task finishes (Section 3.3.2). This ensures that the intermediate results of a multiple-output task are properly forwarded to other tasks by execution of the necessary data transfers.

4.3 Pipelined execution

In Section 4.2, we introduced task pipelines which are started either by an open loop or by a task which produces multiple outputs. It turns out that, the model presented in Chapter 3 has some safety problems when it is used to model pipelined workflow execution. In the following, we analyze the problems which occur in task sequences and in control flow merges.

4.3.1 Pipeline collisions

Already for a simple sequence of tasks as in Figure 4.5, the introduction of pipelining can cause problems. Considering that different tasks may take different amounts of time to execute, it might happen that while a task T is processing a stream element, its predecessor finishes processing the next element. Since T is *Running*, it will not be reset by the predecessor (Section 3.2.4). Therefore, the starting conditions of T will not be re-evaluated and the new input for T will not be processed. We call such a situation in a pipeline a *collision*.

To avoid collisions, we need to define the behaviour of the navigator in cases when task input becomes available to a task which is still running. The following briefly compares five ways to deal with this situation discussing their impact on the semantics of tasks.

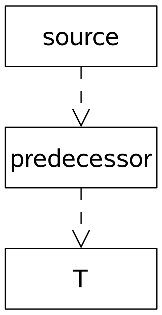


Figure 4.5: Simple pipeline with three tasks. The first task (source) drives the pipeline by pushing out data.

Discard data

In the simplest case, the pipeline collision is not avoided and the input is discarded. Discarding data which arrives too fast has been studied in the context of stream processing engines. Here, *load shedding* [108] is used to reduce the load on the system and thus to reduce the latency of the processing of data. Where and when data is dropped is determined optimally for each overload situation. In the context of business processes, discarding data is rarely a desirable solution and is therefore mentioned here only for the sake of completeness. The approach is only useful if, e.g., the stream data is being aggregated in some way by the workflow process and thus missing data does not completely invalidate the result (e.g., when calculating an average value). Also, the techniques of intelligent load shedding cannot be directly applied to a business process as the tasks invoke black box services with unspecified semantics. If data is to be discarded, this should be done as early as possible to avoid unnecessary processing of the data before it is discarded.

Multiple input

The second alternative is for the new input to be provided to the running task execution while the task is in the *Running* state. This kind of data handling allows a task to process multiple elements of a stream during one execution. This is symmetric to the multiple output feature of source tasks (Section 4.2.3). An important limitation of this approach is that the multiple input feature is not compatible with non-streaming tasks since these assume discrete input and output.

If a task supports both multiple input and multiple output, it becomes a “stream operator”. Such an operator can be used to filter elements of a stream or to calculate aggregate functions as the state of a task execution persists over multiple stream elements. The model of a stream operator task also includes the *Outputting* state in order to produce partial results during an execution (Figure 4.2). In [15] the authors take a similar approach and employ stream operators as tasks to process streaming data.

This approach exhibits two problems. One is that it forces the composition engine to be aware of the nature of the tasks and distinguish between streaming tasks and non-streaming tasks – not to mention the difficulty to combine them. The other is that it moves all the problems of dealing with streaming data and collisions to the stream operator implementation, with the engine acting solely as a configuration tool of a data stream processing pipeline.

Multiple instances

Another option is to create a new instance of the task when data is available in the pipeline and execute the new instance concurrently with existing ones. This behavior is similar to the "Multiple instances" workflow patterns [100]. The approach, however, leads to several non-trivial problems. From the engine point of view, the state pertaining to every instance that is created to process each stream element needs to be stored somewhere, thereby adding significant overhead. From the programming point of view, the synchronization of instance termination and starting of successor tasks needs to be addressed. This is not easy as there might be constraints on the order in which stream elements should be processed.

Queue

Given that task instances should not be required to accept multiple inputs once they are started and considering the problems of starting multiple instances of the same task in parallel, the fourth alternative we present is to buffer the task input in a queue so that only one task instance is running at all times.

With this approach the results produced by predecessor tasks are copied into the successor task's input parameters as usual. However, if the task is running, the data is buffered into a queue of execution requests instead of restarting the task over the new input data. Once the task completes its execution, it fetches its next input from this queue when it is about to be started again.

Compared to the multiple input and multiple instances strategies, the queueing approach makes no additional assumptions about the task execution capabilities. All stream elements are processed in order, and for every element the task is invoked without having to be aware of the stream.

Although, as a first approximation, queues may have unlimited capacity and thus are able to handle an infinite number of input data, in order to implement this semantics the engine can only provide queues of limited capacity. When a queue gets full, executions have to be skipped because the input data can no longer be queued. Queues can only even out temporary variations in speed between a producer and a consumer of a stream. For permanently different rates, an additional blocking control mechanism between producer and consumer is needed as is discussed below.

Block

As the last option, we propose to block the predecessor task whenever a collision is about to occur. Whenever input is available for a task T which is already running,

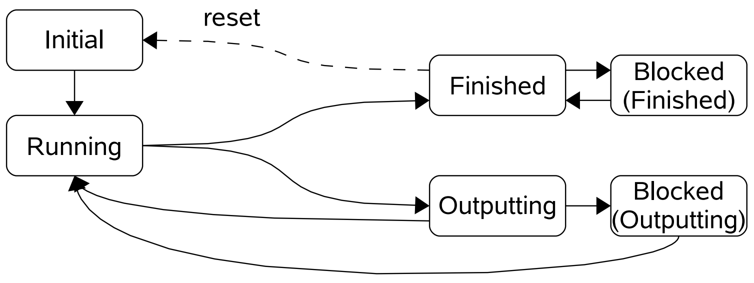


Figure 4.6: Finite state machine for a task including the *Blocked* states.

the collision is avoided through not evaluating the starting conditions and therewith not restarting the running task. Also, the predecessor is put into a new *Blocked* state (Figure 4.6). In this state a task is not allowed to be restarted. This prevents it from producing even more output. If necessary, the blocking of tasks is propagated upstream from the predecessor of T in order to prevent any collisions further up in the pipeline which might occur while T is running. After task T finishes its execution, it checks whether its predecessor is in the *Blocked* state. If this is the case, its starting condition is reevaluated immediately, so it may be started over the new input. The predecessor is released from the *Blocked* state so it can be restarted whenever input becomes available for it.

Like the queueing strategy, this approach makes no special assumptions about the task execution and processes the stream elements in order. In addition, there is no possibility that data is lost as in the above approach because tasks are prevented from being restarted whenever their output has not been processed by the successor task. Therefore, we choose this strategy out of the presented approaches to control the pipelines in processes.

The blocking strategy makes pipelining safe for a sequence of tasks. It could also be used to make the queueing approach safe when queues with a limited capacity are used by combining the two approaches. However, the block approach needs to be refined to work in other constructs like control flow merges. The problem of merges and an improved blocking approach to control pipelining in arbitrary processes are discussed in the following sections. We will also show how to combine the approach with queues, making it safe to use the queueing approach.

4.3.2 Problems of merges in pipelines

Most workflow processes employ a control flow merge in some way. A *control flow merge* is formed by a task with multiple predecessors. Figure 4.7 shows a simple example thereof. Whenever one of tasks A and B terminates, control is passed to C. Thus, control arriving from these two tasks is *merged* at C. In the example, C waits for both predecessors to finish before starting, which means that the merge is *synchronizing*.

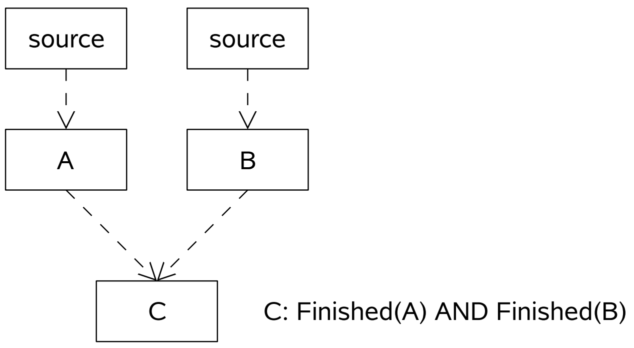


Figure 4.7: A simple process with a synchronous control flow merge.

Independent of which strategy is chosen for avoiding pipeline collisions, when merges are used with pipelining, our basic process model exhibits two problems. In the following examples we consider the simple process from Figure 4.7. A and B can execute multiple times because they are fed with data from the source tasks.

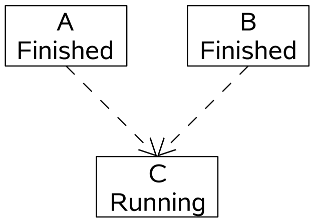
State ambiguity

We assume both tasks A and B are executed and become *Finished* whereupon C may be started (Figure 4.8a). Then, after C has finished (Figure 4.8b), task A is executed again (Figure 4.8c). The finishing of A resets C (Figure 4.8d), triggering the evaluation of the starting conditions of C. Since A is *again Finished* and B is *still Finished*, C will be started another time, although B has not provided any new data (Figure 4.8e). Thus, the synchronizing merge has become non-synchronizing after the first two stream elements have been processed. The problem here is that the navigation algorithm cannot distinguish between different executions of a task because navigation is state-based and there is only one *Finished* state. When the starting conditions are evaluated, it is indistinguishable whether the predecessors have been re-executed or not since this was checked last time.

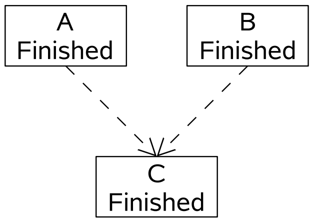
Output overwriting

Assume A is executed and finishes. Then, the starting conditions of C are evaluated but are not satisfied because B has not yet been executed. Before B is executed, A is executed once more. A now has overwritten its previous output with the new output. However, C could not consume the old output before it was overwritten, so the corresponding stream element is lost.

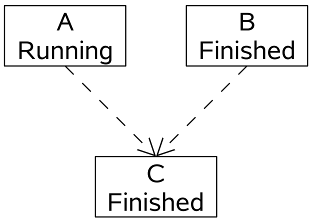
It is important to note that both of the above problems cannot be solved with any of the approaches for avoiding pipeline collisions (e.g., multiple input, multiple instances). The reason is that in the illustrated cases, task C never becomes ready to execute and therefore a collision never occurs. Instead, the information exchange between the tasks must be taken into account.



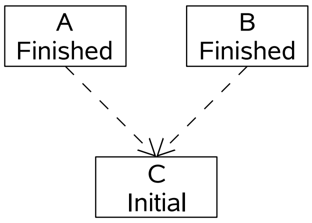
(a) A and B have executed, C is started.



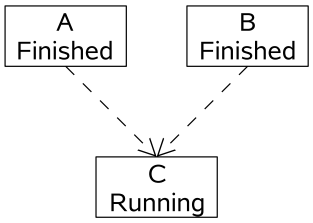
(b) C has finished.



(c) A is restarted.

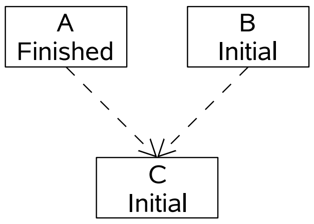


(d) A finishes and resets C.

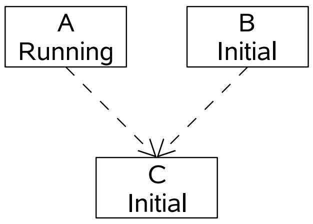


(e) C is restarted.

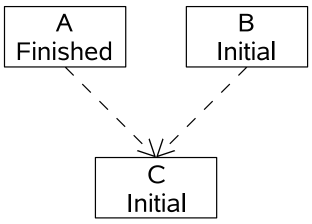
Figure 4.8: State ambiguity occurring in the merge from Figure 4.7. The source tasks are not shown.



(a) A has executed, C cannot start.



(b) A is restarted.



(c) A has been executed again, C can still not start

Figure 4.9: Output overwriting occurring in the merge from Figure 4.7. The source tasks are not shown.

4.3.3 Controlling the flow of information

The previous sections have shown that the flow of information between tasks must be taken into account when pipelining is used in a process. Information in this case includes not only the data which is transferred between parameters. Also the activation of a task as a reaction to the termination of another task constitutes a transfer of *control information* between these two tasks. This information contains the execution state of the terminating task and is is also subject to the overwriting and ambiguity problems. We define the *task outcome* as the overall result of a task execution. The task outcome consists of the output parameter values and the execution state in which this output was generated (e.g., *Finished*, *Failed* or *Outputting*).

We have presented solutions to control pipelining in sequences of tasks. However, merges exhibit problems which cannot be solved with those approaches. In the following we propose an extension to the process execution semantics which solves the pipelining problems also for merges. After that, we give details on how to implement these extensions.

Semantics extension

The output overwriting problem shows that there is a need for controlling the flow of information between tasks. To ensure that a consumer of a stream is not overloaded, *flow control* is a mechanism used to slow down the stream producer. In general, when the consumer receives too much data it informs the producer which will then lower its output rate. To prevent a task from overwriting its previous output we extend the basic model with the following restriction: a task cannot be restarted before its output has been consumed by all successors. In addition to evaluating the activator and data-condition of a task, the navigation algorithm enforces this restriction before executing a task. The restriction guarantees that the output of a task execution can always be stored in the output parameters and does not need an additional buffer. This is similar to the Block approach in Section 4.3.1.

To overcome the state ambiguity problem, a task needs to know when a predecessor has finished but the task has not yet learned about this fact. Therefore, for each predecessor, the task keeps track of when the predecessor changes its state (or reaches the same state again). If it has not changed, the state is regarded as undefined and thus the starting condition cannot be fully evaluated before the state changes again. This prevents that the information is used more than once to start the task.

Pairwise information exchange

The flow of information between tasks is a problem which must be solved pairwise, i.e., for every pair of tasks which exchanges information. This can be illustrated with the overwriting problem. A solution which works for a sequence of tasks is to introduce a new state *OutputConsumed*. When a task finishes, it waits until its successor has been started with the new data. Then it changes to the *OutputConsumed* state. When it has reached this state, it is ready to be restarted because its output has been read and may now be overwritten. However, for a task with multiple successors this approach cannot be applied directly. Such a task cannot enter the *OutputConsumed* state before all of its successors have read the output. To keep track of which successors have read the data, a flag for every successor is needed. Using such flags means that the flow of data to each successor is managed separately. Thus, the data exchange is taking place between pairs of tasks.

Any two tasks which exchange information need to have this communication controlled. As stated above, this information consists not only of data transferred along data flow bindings, but also control information. Thus, any two tasks connected by a control flow dependency or data flow binding constitute such a task pair.

It is important to note that the information exchange is controlled between pairs of tasks and not between pairs of parameters. According to the process model, a task execution always provides data for all the task's output parameters, i.e., the parameters change their value simultaneously. Therefore, even if a task has several data flow bindings to another task, there is no need to control the information flow between single pairs of output and input parameters connected by a binding. Rather, the information contained in the output parameters of a task can be regarded as one set of values. This set can be transferred in one go to the parameters of the other task. This decreases the control overhead for the information exchange between tasks.

Buffers

The use of flow control in a task pipeline prevents the loss of data. However, this may lead to part of the pipeline becoming stalled if there exists a bottleneck. To alleviate the situation, buffers can be employed between the tasks in addition to flow control. Buffers compensate for temporary differences in execution speed of tasks (Section 4.3.1). This reduces the probability that tasks are blocked due to flow control if their output cannot immediately be consumed by other tasks.

When using buffers, information is not copied directly from output parameters to input parameters, but temporarily stored in a buffer when being transferred from one task to another or between tasks and the process. This is done in a first-in-first-out (FIFO) manner, i.e., buffer elements are retrieved from the buffer in the same order as they were appended. In this section we briefly discuss where such buffers should be added in a process.

The information exchange between dependent tasks is controlled pairwise. However, if a task T has more than one predecessor and/or more than one successor, there are three different options for placing buffers between tasks (Figure 4.10). In the first case, a buffer is attached to T to store the output produced by T (Figure 4.10 left). The successors of T retrieve their input from this buffer. Since the buffer is shared by all successors, the first element in the buffer is removed only after all successors have read it. In the second case, a buffer is put in front of T (Figure 4.10 center). Whenever the predecessors have provided sufficient data as

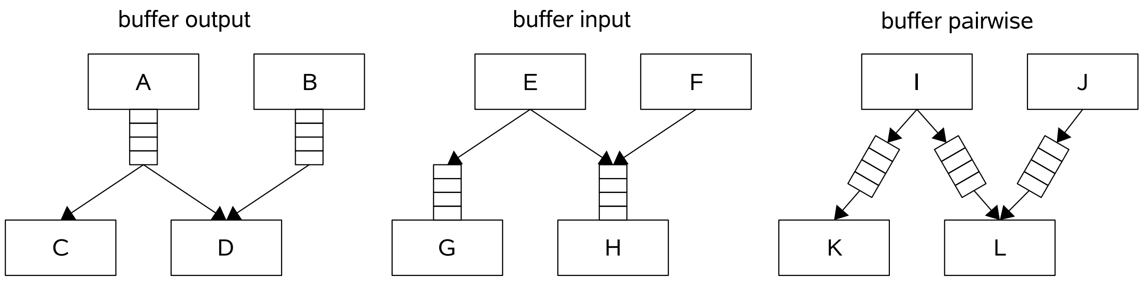


Figure 4.10: Different options for buffer placement between tasks.

input for T, the data is appended to the buffer. Whenever T is ready to execute it fetches the next input from the buffer. This is the same approach as proposed for sequential pipelines (Section 4.3.1). The third approach uses a buffer between each pair of tasks (Figure 4.10 right). Before executing, T collects its input from all the buffers with output from predecessors. The output of the execution is then appended to all the buffers towards successor tasks.

The effect of the first two approaches is that tasks with dependencies get decoupled from each other because the output from a task does not have to be processed immediately by a successor task. Instead, the data is buffered, either at the output of the predecessor or at the input of the successor. However, both approaches have the drawback that *sibling tasks* are not decoupled from each other (C and D, and E and F in the figure). In the case of a shared output buffer, all successors must have read the head element of the buffer before the next element can be processed by any of the siblings. With a buffer at the input, if the control flow is synchronized at T, every predecessor must wait until all other predecessors have terminated before it can be restarted. These synchronizations of sibling tasks form an unnecessary restriction on the execution order of tasks.

The third approach solves these problems by not sharing any buffers between tasks. A buffer is used solely for the communication between a pair of tasks. This means that a task is not only decoupled from its predecessors and successors but also from its sibling tasks. Therefore, we choose this approach to place buffers between tasks. It fits with the approach used for flow control between tasks where the information flow is also managed pairwise. This is of advantage when combining buffers and flow control as will be shown later.

We place exactly one buffer between a pair of tasks even when the tasks are connected by multiple data flow bindings and typically also a by control flow dependency. As stated above, there is no need to control the information flow along every data flow binding. Similarly, we do not need to place a buffer on each binding but the data for all the bindings can be transferred through a single buffer. Further details on using buffers to transfer information between tasks are given in Section 4.5.2.

4.4 Flow control semantics

We have presented the problems of using pipelining in a workflow process and proposed solutions. In this and the following section we show how to implement these solutions by changing the navigation algorithm.

4.4.1 Readiness and freshness markers

To keep track of whether the outcome of a task execution has been consumed, we use a Boolean *readiness marker* for every pair of dependent tasks, with the following semantics. If the marker is set, the information in the corresponding output parameters has been read and the parameters are ready to accept new data. When the marker is cleared, the corresponding information has not been consumed yet. This extension corresponds to the approach of using the *Blocked* state to block tasks in a sequential pipeline (Section 4.3.1) but it is not restricted to task sequences.

Regarding the state ambiguity, we introduce a Boolean *freshness marker* which is also added for every pair of dependent tasks. The semantics of the marker is the following. A set marker means that the corresponding predecessor has reached a new state (or reached the same state again), i.e., there is fresh information available. A cleared marker means that the corresponding successor has taken note of the new state. In this case the state of the predecessor is regarded as undefined.

4.4.2 Marker mechanism

The state of a process instance is extended with a readiness marker and a freshness marker for every pair of tasks with a control flow dependency. If the tasks in the pair have a cyclic dependency, then there are four markers, two for each direction of dependency. A marker has a corresponding predecessor task and a successor task where the successor depends on the predecessor. Initially, readiness markers are set and freshness markers are cleared.

Figure 4.11 shows the state machine for a task complemented with the conditions and actions imposed by the marker mechanism. The task supports multiple output and the *Unreachable* and *Failed* states have been left out for simplicity. The mechanism introduces the restriction that freshness markers towards predecessors must be set and readiness markers towards successors must be cleared in order for a task to be executed. This translates into requiring that there must be new information available from the predecessor (data freshness) and the information towards successors must have been consumed (flow control).

Markers influence the execution cycle of a task (from *Initial* to *Finished*) as follows. Before evaluating the starting conditions, the readiness markers towards successors need to be inspected. If at least one such marker is cleared, the task may not be started. This makes sure the previously produced output of the task is not overwritten (flow control). Otherwise, if all those markers are set, the starting conditions are checked.

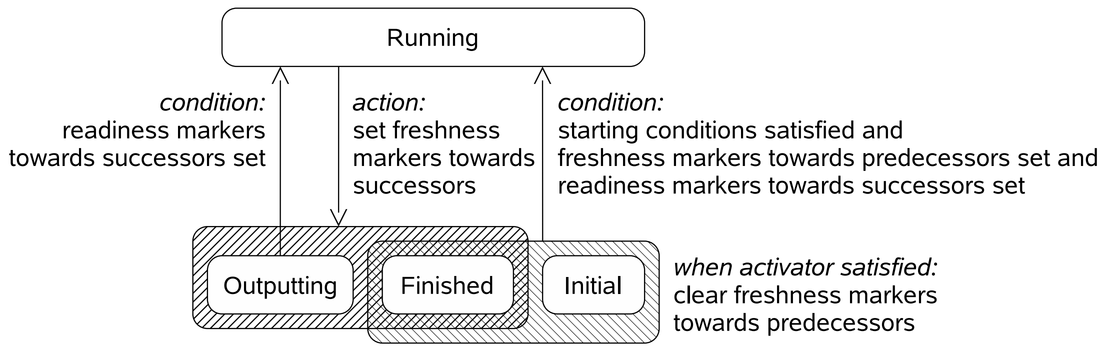

Running
condition:
readiness markers towards successors set
action:
set freshness markers towards successors
condition:
starting conditions satisfied and freshness markers towards predecessors set and readiness markers towards successors set
Outputting
Finished
Initial
when activator satisfied:
clear freshness markers towards predecessors

Figure 4.11: Task state machine extended with conditions and actions to implement the marker mechanism.

When evaluating the activator, the state of the freshness markers towards predecessors needs to be taken into account. This prevents that the evaluation of the starting conditions uses stale information (state ambiguity). A control flow dependency is not satisfied unless the corresponding marker is set. This corresponds to augmenting the enablement condition of the dependency with the freshness marker.

If the activator is satisfied, all readiness markers towards predecessors are set, indicating that the corresponding information has been consumed, and the corresponding freshness markers are cleared to avoid reading the same information again. Because the readiness markers have been set, the evaluation of the starting conditions of all predecessors is scheduled. These tasks might only be waiting for their output to be consumed (flow control) and could therefore be ready to start.

Then, the data-condition is evaluated as usual. When both the activator and the data-condition are satisfied, the task is started.

If, however, the activator is not satisfied, then the deactivator is evaluated in order to perform the dead path elimination. A satisfied deactivator marks the task as *Unreachable*. Regarding flow control the deactivator behaves like the activator – the freshness markers are included in the evaluation of the deactivator, markers are updated if the deactivator "fires". Likewise, if the activator was satisfied but the data-condition is not, the task changes to the *Unreachable* state as well to perform dead path elimination.

After being started, when the task reaches the *Finished* or *Outputting* state, for every control flow dependency towards a successor the corresponding freshness marker is set and the readiness marker is cleared. This indicates to successors that new information is available and prevents the task itself from overwriting its output. If the task is in the *Outputting* state, it does not return to the *Running* state before the readiness markers have been set. If the task is in the *Finished* state and it does not have any successors, the evaluation of its starting conditions is scheduled. Since there are no outgoing dependencies, the task can restart right away if information has become available from predecessors.

Avoiding congestion

As described above, a task reacts to state changes by the predecessor tasks. Depending on the states reached either the activator fires, the deactivator fires or nothing happens. Normally, the reason for neither the activator nor the deactivator to be satisfied is that not enough information is available yet to decide that the task is ready to start or to decide that it is unreachable. The task must wait for more predecessors to change state until such a decision can be made.

There is, however, a case where the reason is not the lack of information but the wrong information: when a predecessor enters a non-final state. This state might not be expected by the activator and the deactivator only reacts on final states (*Finished*, *Failed*,...). An example of such a situation is a task with a *Finished*-dependency on a multiple-output task. When the multiple-output task becomes *Outputting* this will not satisfy the activator which is waiting for a *Finished* state. It will neither satisfy the deactivator which only reacts to final states. Therefore, such cases are taken into account by discarding such data which will neither be consumed by the activator nor by the deactivator.

4.4.3 Impact of the marker mechanism

The marker mechanism solves both the problem of output overwriting and the state ambiguity. The overwriting of output is avoided as follows. When a task produces output by becoming *Finished* or *Outputting* it clears the readiness markers towards its successors. As long as these markers are cleared, the task does not enter the *Running* state again. Therefore, the task cannot execute and produce new output as long as the markers are cleared[1]. When successors have consumed the output, they set the corresponding markers. Thus, when all markers towards successors have been set, this means that the output has been consumed completely and the task can safely be re-executed.

The state ambiguity problem is solved as follows. When a task produces output, it sets the freshness markers towards its successors. When the activator of a task is evaluated, the freshness markers towards predecessors are taken into account, since the condition attached to the dependency has been augmented with the marker. A control flow dependency is only enabled if the corresponding freshness marker is set. If the activator is satisfied (which means that all the corresponding markers are set), all freshness markers towards predecessors are cleared, indicating that the corresponding information has been consumed. If the starting conditions were immediately reevaluated, the activator would not be satisfied, even though all predecessors have the required state. This is because the markers have been cleared. The activator will not be satisfied until all predecessors have produced fresh informa-

[1]In the case of a multiple-output task, the task execution is continuously providing the process with new output asynchronously. See Section 5.2.2 for information on how the runtime system prevents the task execution from producing further output and thus keeps it in lockstep with the task state.

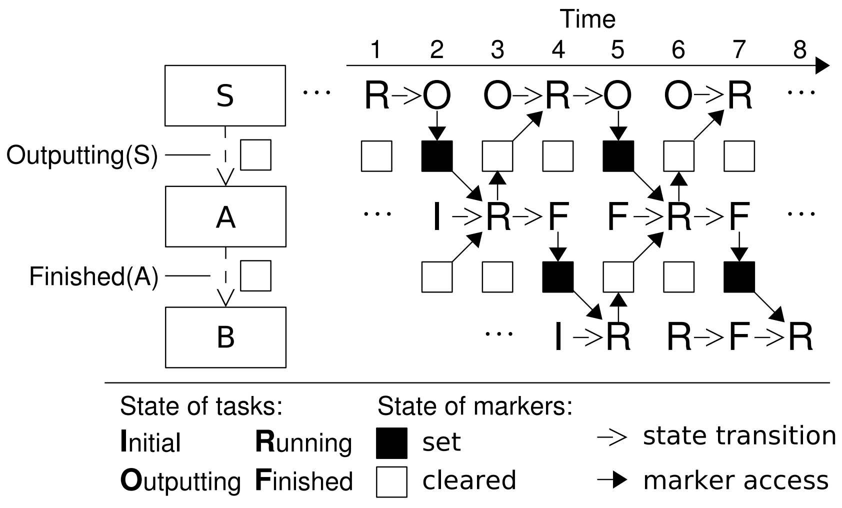


Figure 4.12: Example interaction of the marker mechanism with the state transitions of tasks. Only the freshness markers are shown.

tion by reaching a new state or the same state again and setting the corresponding freshness marker.

After introducing the combined marker in the next section, Example 4.2 takes a closer look at the interaction of the markers with the navigation algorithm.

4.4.4 Combined marker

When the data which flows between two tasks is kept in the respective pairs of input and output parameters (or, in the case of the task state, in the respective system parameter), these parameters together with the corresponding readiness marker and the freshness marker constitute a one-slot buffer. In this case the two markers can be combined into a single marker because the state of one marker can be derived from the state of the other. When data is written into the one-slot buffer, the readiness marker is cleared and the freshness marker is set. This prevents the information from being overwritten and signals that new information is available. Similarly, when the data is read by a task, the readiness marker is set and the freshness marker is cleared since the data is no longer fresh and may be overwritten. Also, the possibility of combining the markers becomes apparent when considering that a one-slot buffer can only be in one of two states, full or empty, which can be encoded in a single Boolean variable.

Example 4.2: Marker interaction

Figure 4.12 summarizes the interaction of the marker mechanism with the state transitions of a task using an very simple sequence of tasks. On the left side, the figure shows the sequence with three tasks whereof the first is a multiple output task. Between the tasks S and A, and A and B there is one combined marker each to control the information flow. The right side of the figure shows the state of the three tasks and of the two markers as it changes over time. The time in the diagram has been discretized for the sake of clarity. Every time slot corresponds to one round of the navigation algorithm.

In round 1, A and B are in the *Initial* state and S is already *Running*. Then, S produces a result and changes to *Outputting* in round 2. Since S has changed its state, the marker towards A is set to indicate that new information is available. As the marker towards B is cleared (there is no unconsumed information in that direction) and there is fresh information available from A, the starting condition of A is evaluated and A is started in round 3. At the same time, the marker towards S is cleared to indicate the consumption of the information. This allows S to return to *Running* in round 4. Also in this round, A terminates (we assume a very fast execution of the task). This entails setting the marker towards B. In round 5, the starting condition of B is then evaluated on the new information coming from A after which B is started. The starting of B clears the marker towards A. Also in round 5, S again provides results in the *Outputting* state which is indicated to A through setting the corresponding marker.

Round 6 is very similar to round 3 except that B is *Running* now. This, however, has no effect on navigation. In round 7, B becomes *Finished* which is equivalent to the *Initial* state from round 4 (the task can be (re)started). Therefore, the same pattern of state changes can be observed.

In the figure it can be be seen that while the control flow in the task sequence is from S to A to B, there is also a communication taking place in the reverse direction. This can be traced by following the *marker access* arrows from B in round 5 to A in round 6 to S in round 7. This communication constitutes the flow control between the tasks.

4.4.5 Restarting tasks

Each task has a finite state machine to track the execution of the task. This state machine is resettable, i.e., after an execution, the state machine can be reset back to the *Initial* state. This capability allows for the reexecution of tasks which is a requirement for loops and pipelines in a process. In the basic model the resetting of a task is initiated by a predecessor task when it terminates (Section 3.2.4). After being reset, a task will reevaluate its starting conditions taking into account the new results produced by its predecessor. This way, the termination of a task triggers the possible reexecution of it successors.

When using flow control in the execution of a process, however, the resetting mechanism becomes redundant. In order to have the starting conditions of a task reevaluated it is no longer necessary to reset its state machine. Instead, the flow control information between the task and its predecessors indicates whether the predecessors have terminated and the task's starting conditions should be reevaluated.

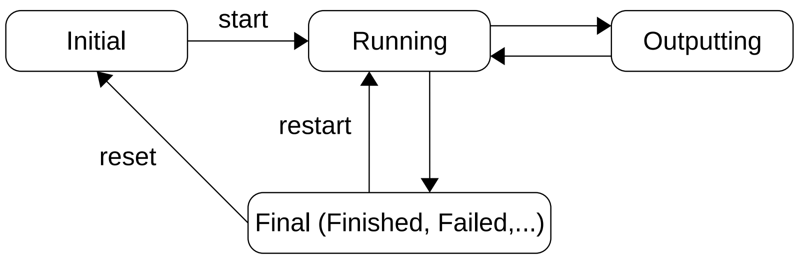


Figure 4.13: Paths taken in a task's finite state machine to restart the task without flow control (*reset*) and with flow control (*restart*).

This is checked every time the activator of a task is evaluated, by taking into account the markers towards predecessors.

It would still be possible to reset the state machine of a task after the markers towards predecessors indicate the termination of these tasks. However, this would introduce an unnecessary complexity. The task would change to the *Initial* state only to have its starting conditions evaluated. Hence, if the flow control information is checked as part of the starting conditions, the state transition and the associated delay is avoided.

To summarize, the resetting of a task's state machine by predecessors is not employed anymore. Instead, a task may be restarted directly from the state in which it terminated. Figure 4.13 illustrates the difference between the two approaches in the task state machine.

4.5 Process semantics with buffers

Having developed the algorithm for enforcing flow control in a process, in this section we extend it with buffers. First, we show how the properties of a buffer map to markers in order to reuse the same flow control algorithm with buffers. Then, we describe the necessary changes to the navigation when buffers are used.

4.5.1 Turning markers into buffers

When flow control is used to transfer information from one task to another, an implicit one-slot buffer exists between these two tasks which is managed according to the marker mechanism (Section 4.4.4). Buffers are also placed between pairs of tasks which exchange information (Section 4.3.3). Thus, the integration of buffers in the process execution is a natural extension of the marker-based flow control mechanism. Consequently, the marker mechanism can be directly applied to these buffers with bigger capacities. The following discusses how the state of the flow control markers and their manipulation are mapped to buffers.

The state of the readiness marker indicates whether the buffer has capacity left to append another element. If this is the case, then the marker is set. Otherwise, if

readiness marker	freshness marker	buffer state
cleared	set	full
set	cleared	empty
set	set	partially full
cleared	cleared	zero-length buffer

Table 4.1: Possible states of the freshness and readiness markers and the state of the corresponding buffer.

the buffer is full, the marker is cleared. Thus, as long as the buffer is not full, the corresponding predecessor task is allowed to execute, since there is capacity left to store its output.

For the freshness marker the state is derived analogously from the "emptiness" of the buffer. When the buffer contains at least one element, the freshness marker is set. Otherwise, when the buffer is empty, the marker is cleared. Thus, as long as the buffer is not empty, the corresponding successor task can fetch a data element and possibly use it as input for the execution.

Vice versa, the state of the buffer can be deduced from the state of the two markers. Table 4.1 summarizes the states of a buffer and the corresponding markers. The last of the four cases has been included only for the sake of completeness. When both the readiness and the freshness marker are cleared this means that the buffer is full and empty at the same time. The only buffer for which this situation is possible is a buffer with a capacity of zero elements. Hence, this case will never occur since such a buffer cannot be used to transfer any data and will therefore not be used in a process.

When using markers, these are manipulated directly by the navigation algorithm. With buffers, the manipulation takes place indirectly through appending and removing elements from a buffer. When the freshness marker is cleared (and the readiness marker set) the data has been removed from the one-slot buffer. Thus, this manipulation corresponds to removing an element from the buffer. Similarly, when the readiness marker is cleared (and the freshness marker set) data has been stored. This corresponds to appending an element to the buffer. Markers are always manipulated whenever data is stored or removed from the one-slot buffer. This is because the tiny buffer immediately becomes full or empty, respectively. However, a larger buffer is often in an intermediate state where it is neither empty nor full. Thus, the markers do not change for every append or remove operation.

Finally, it should be noted that since the states of the readiness marker and the freshness marker depend directly on the state of the buffer, the markers actually become redundant and need not be maintained explicitly.

4.5.2 Buffered data transfer

The data flow compiler schedules data transfers which copy information between output parameters and input parameters (Section 3.3.2). However, when using buffers between tasks these data transfers are no longer needed. The transfers are replaced by buffers which convey information from output parameters to input parameters. Values of output parameters are appended to a buffer and correspondingly data for input parameters are retrieved from a buffer. Compared with the scheduling done by the data flow compiler this corresponds to a mixture of eager and lazy scheduling because a value is always pushed from an output parameter into a buffer and later pulled from the buffer into an input parameter.

For every control flow and data flow dependency there is a buffer which carries the corresponding information. It is the buffer which connects the same tasks as the dependency. If two tasks are connected by a control flow dependency and several data flow bindings, there is one buffer between these two tasks which transfers task states for the control flow dependency and parameter values on behalf of the bindings.

When a task terminates or provides output in the *Outputting* state, the task outcome is collected and appended to buffers towards other tasks. The information which is written to a buffer is determined by the bindings towards the corresponding task. For each binding the value of the corresponding output parameter is appended together with the parameter name to be able to assign it to the correct input parameter at the downstream task. For every control flow dependency the current state of the task is appended to the corresponding buffers so dependent tasks can evaluate their activators. It is important to preserve the correlation between the different parameter values and the task state in which the output was produced. Therefore, the task outcome is enqueued to the buffer as one data structure.

Before a task can be executed, its input parameters must be loaded with the input for the task execution. Instead of copying the data directly from output parameters of other tasks, the information is fetched from the buffers where the corresponding output has been placed.

The binding attached to an input parameter determines the buffer which provides the new value for the parameter. It also determines the name of the output parameter whose value was written to the buffer on behalf of the binding. The navigator fetches a task outcome from the buffer and extracts the parameter value using the name of the output parameter. This value is then assigned to the input parameter.

As described above, the parameter values which are output at the same time by a task are buffered in one piece. This allows the navigator to assign a consistent set of input values to a task's input parameters when there are multiple data flow bindings between two tasks.

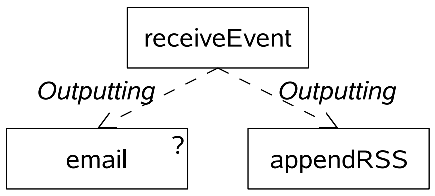


Figure 4.14: Control flow of the log event dispatching process. The *email* task is only triggered by error messages. The corresponding data-condition is indicated by a question mark.

Example 4.3: Log event dispatching

This example illustrates how the output of a task is stored in buffers to make it available to other tasks in the process. We use an example process which receives logging events (e.g., [6]) from an application and distributes the events depending on their priority level (Figure 4.14). Logging events consist of a priority level (e.g., *info* for informational messages, *warn* for warnings, *error* for errors), a textual message and an optional exception stacktrace which describes where the error, if any, occurred in the application. All events are by default appended to an RSS [13] feed. This allows the operator of the application to periodically check its status in terms of the last logged messages. In case an event is received which indicates an error in the application an e-mail is additionally sent so the problem can be dealt with as fast as possible.

The first task in the process, *receiveEvent*, is a multiple-output task. It keeps running as long as the process is active. For every logging event which is received from the application the task changes to the *Outputting* state and provides the data contained in the event in its *level* and *message* and *exception* output parameters (Figure 4.15). This output is used by the *appendRSS* task to update the RSS feed and, in case the event indicates an error, by the *email* task to send an email to the operator. The two tasks have an *Outputting* dependency on the first task and are thus restarted for every event. In order to activate the *email* task only for error events, the data-condition of the task requires the value of the *level* parameter to have the value “error”.

Although there are several data flow bindings between the tasks, one control flow dependency is enough to impose a correct execution order between two tasks. Correspondingly, a single buffer is sufficient to transfer information between a pair of tasks.

Whenever the *receiveEvents* task produces output, the outcome is appended to the buffers towards the other tasks. The outcome consists of the current state of the task, *Outputting* in this case, together with the values of the necessary output parameters. Which output parameters are of interest to the successor task depends on the data flow bindings (Figure 4.15). As shown in Figure 4.16, the values of all output parameters are buffered in the direction of the *appendRSS* task. For the *email* task, only the *level* and *message* parameters are stored.

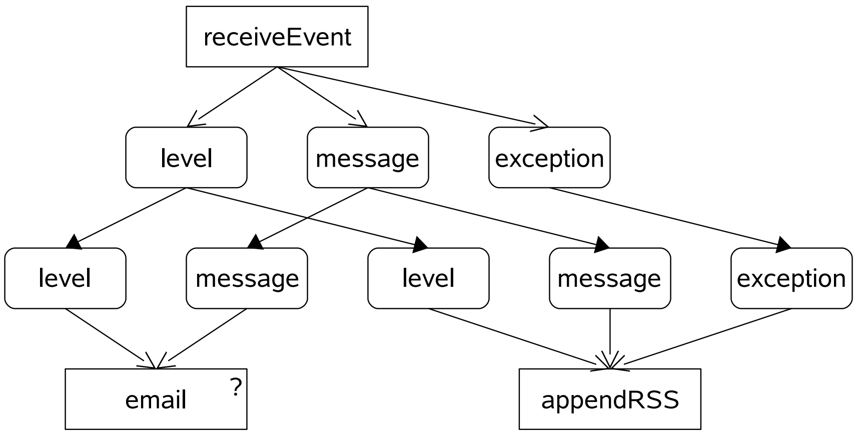


Figure 4.15: Data flow of the log event dispatching process

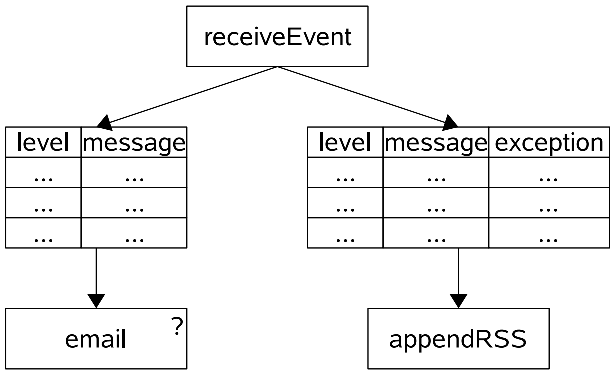


Figure 4.16: Buffers between the tasks of the log event dispatching process

State	level	message
Outputting	warn	using default value...
Outputting	error	File not found:...
Outputting	warn	failed to initialize...
Outputting	info	starting container...

Table 4.2: Content of the buffer between the *receiveEvents* and *email* tasks.

State	level	message	exception
Outputting	warn	using default value...	–
Outputting	error	File not found:...	[stacktrace]
Outputting	warn	failed to initialize...	–
Outputting	info	starting container...	–
Outputting	error	Component has no...	[stacktrace]
Outputting	info	starting monitor...	–
Outputting	debug	processing message in...	–

Table 4.3: Content of the buffer between the *receiveEvents* and *appendRSS* tasks.

Whenever the two successor tasks are ready to be started, the values for their input parameters are fetched from the buffer. Every tuple in the buffer is a consistent set of output values from *receiveEvent* which can be assigned to the input parameters so the task is started with matching *level* and *message* and *exception* values.

Tables 4.2 and 4.3 show possible contents of the buffers during the execution of the example process. Several task outcomes have been appended by the *receiveEvents* task but not yet consumed by the successor tasks. The buffer in front of the *email* task contains fewer elements than the other buffer because the task is only invoked on error events. Discarding data from a buffer is naturally faster than invoking a task with the data as input.

4.5.3 Data flow merges with buffers

The data flow compiler takes into consideration that input parameters with multiple bindings must be written by "push" data transfers executed on termination of the corresponding predecessor tasks. This ensures a "last writer wins" semantics for the input parameter. With buffers, however, this is no longer possible because values from output parameters are not copied directly to an input parameter but rather to separate buffers. The time at which data is output by a task does no longer correspond to the time at which the data is written to an input parameter. Multiple tasks providing output for the same input parameter are thus completely decoupled

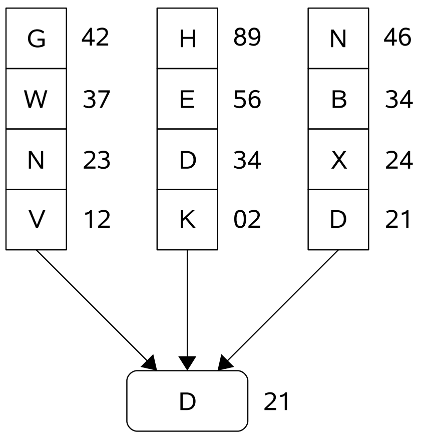


Figure 4.17: The timestamps (shown to the right of the buffer element) of the buffered task output can be used to emulate "last writer wins" semantics at a merging input parameter.

in time. Hence, there is no possibility for these tasks to properly overwrite each other's data.

The conflict of the multiple bindings has to be resolved by the downstream task when fetching data from multiple buffers and writing it into the input parameter in question. To keep the semantics of a process compatible with a non-buffered version of the same process, "last writer wins" semantics can be emulated with the help of timestamps. Every time task output is appended to a buffer a timestamp is attached to it[2]. By comparing the timestamps associated with different values for the same input parameter it is then possible to achieve the desired semantics (Figure 4.17). "Last writer wins" is obtained by choosing the value with the highest timestamp. This corresponds to copying all candidate values in the order of their timestamps, therewith simulating push data transfers as in the non-buffered case. Thus, the deferred conflict resolution achieves the same semantics as without buffers.

4.5.4 Navigation on buffers

The fact that the information flow between tasks uses buffers also means that the entire state of a process instance is contained in these buffers. Moreover, not only the latest state is contained but also a history of it, depending on the capacity of the buffers. In particular, the state which is needed by the navigator is no longer available in output parameters of tasks or as their current execution state. The necessary information has been written into buffers where it stays available. Thus, the navigator has to make its decisions based on the process state which is held in the buffers.

In order to evaluate the starting condition of a task, the necessary information about other tasks (task state, parameter values) is retrieved from the corresponding

[2]See Section 5.3.1 on how timestamps are added to buffered data.

buffers. During this evaluation, however, it is not yet clear if the task will be started or not, i.e., whether the data from the buffers will actually be consumed by the task by using it as input for the task execution. According to the marker mechanism, the information is only consumed after the activator has been satisfied (by clearing the freshness marker or removing from the buffer). Thus, the evaluation of the starting condition must not alter the buffer by removing the data which is inspected. Consequently, buffers must support a *peek* operation which reads the first element without removing it.

A problem which may arise when the starting condition is evaluated on data directly from buffers is that some of these buffers might be empty. The compiler needs to protect against this by checking if the buffer is empty before attempting to read from it. This is exactly why the "freshness" of data is checked. If the buffer is empty (the freshness marker is cleared), there is no fresh data available and the evaluation of the activator fails without even accessing the buffer.

4.6 Termination detection with buffers

In Section 3.2.2 we described how the termination of a process instance is determined in order to allow for implicit termination. For this, the states of all tasks in the process instance are inspected. If all tasks have reached a final state, then the process instance is considered terminated. This approach is not sufficient when buffers are used between tasks[3]. For example, if all tasks in a process instance have reached the *Finished* state, this does not necessarily imply the termination of the process instance because some buffers might still contain data. This data might trigger the execution of a task the next time the navigation algorithm evaluates the state of the process instance.

Thus, the termination detection algorithm now needs to take the state of the buffers into account in addition to the task states. If a buffer is non-empty, the contained data may possibly be used to start a task which reads data from this buffer. However, whether the task is started depends also on the starting conditions of the task. The data in the buffer may have the wrong value or data from another buffer may be needed in order to start. Thus, to determine if data in a buffer is appropriate for starting a task, the task's starting conditions must be evaluated. Evaluating the starting conditions of tasks is exactly what the navigation algorithm is about. Consequently, the termination detection for process execution with buffers becomes as complex as navigation. In order to not duplicate the calculations on the process state the navigation algorithm should be used for the termination detection in parallel to accomplishing navigation.

If all tasks in a process instance have reached a final state and the navigation algorithm determines that no task is ready to start, then the overall process instance has reached a *stable state*. Rerunning the navigation algorithm over the process instance would not yield a different result since the algorithm depends solely on the

[3]Buffers here also include the single-slot buffers produced by a marker pair between two tasks.

state of the instance. This state can only be changed by the navigation algorithm itself, but since no task is ready to be executed, the state will not change. We define that a process instance terminates when it has reached a stable state. This is a compatible generalization of the previous termination condition. Hence, a process instance has terminated as soon as all tasks have reached a final state and the navigation algorithm determines that no task may be (re)started. The concrete state of the terminated process instance is derived from the task states, as before (e.g., if one task has failed and this is not handled by another task, the entire process fails).

In [84] an algorithm is proposed for detecting termination of distributed computations using markers. At first sight this work might look similar to the problem discussed here. However, the role of the marker is different. In [84] a single marker is sent around the process network whereas we use markers between every pair of tasks. Nevertheless, one could consider applying the algorithm to detect the termination of a process instance by adding support for a special marker traveling around the task graph. However, since in our model the navigator has a global view of the state of the process instance there is no need for applying an algorithm originally designed for a distributed system. The navigator can detect termination simply by evaluating the state of the process instance.

4.7 Impact of new semantics

In this chapter the execution semantics of the basic process model (Chapter 3) has been extended with flow control. However, the model itself has not been altered for this (the multiple-output extension (Section 4.2.3) has no influence on flow control). This raises the question what the impact of the semantic extension is beyond solving the problems of overwriting data and state ambiguity.

In this section, we characterize the new execution semantics in two ways. First, we compare the new semantics to event-condition-action rules. Second, we investigate the semantic changes in terms of workflow patterns [100].

4.7.1 ECA rules

The original execution semantics (Section 3.2) uses a purely state-based approach to evaluating the process state. The activator specifies a constraint on the states of other tasks and the data-condition a constraint in terms of task parameters. Both these conditions are evaluated in order to determine whether a task should be started.

With the semantics introduced by the flow control mechanism, the activator is evaluated only on task states which it has not yet “consumed”. When the activator is satisfied, the states are marked as consumed. Whenever a task terminates it provides the newly reached state to other tasks. This means that the activator is actually evaluated on state *changes* of other tasks. This constitutes an event-based behaviour: every time a task terminates, other tasks are notified of this event.

Thus, a task definition in a process has the structure of an event-condition-action (ECA) rule. Such a rule specifies that whenever a certain event occurs a condition should be evaluated. If the condition is satisfied, then the action is to be executed. To fully capture the semantics of a task definition the ECA rule must be extended with an enablement condition. The rule is evaluated only if it is enabled, i.e., if it's enablement condition is satisfied. The enablement is necessary to capture the flow control semantics which prevent a task from being executed in case a successor has not yet consumed previous results. Also, we require persistent events: in case a rule is disabled and does not react to an event, the event notification is not discarded – this corresponds to the buffering of data between tasks.

A task definition corresponds to an *ECA-rule with enablement* as follows.

- *Enablement:* The rule is enabled only if the readiness markers towards successors are set and if the task is not currently executing.
- *Event:* The Boolean expression of the activator specifies a compound event in terms of atomic events. A task having reached a certain execution state constitutes such an atomic event.

  When the activator is satisfied, i.e., the event has occurred, the event notification is consumed by manipulating the markers towards predecessors tasks.
- *Condition:* The condition corresponds directly to the data-condition of the task. This condition is only evaluated if the activator is satisfied, i.e., when the corresponding event has occurred.
- *Action:* The action corresponds directly to the execution of the task.

4.7.2 Workflow patterns

A workflow language can be characterized in terms of the workflow patterns it supports [100]. Since we have changed the semantics of our language we have investigated the influence of the changes on the support of the patterns. The first question we want to answer is whether a pattern behaves the same with and without flow control. In case the behaviour changes with flow control, the second question is whether the pattern can still be supported by modeling the pattern differently.

To determine the support for the patterns in our prototype system, we have attempted to model every pattern. The patterns which could be implemented were then executed with and without flow control. Tables 4.4 and 4.5 summarize the 43 workflow control patterns specified by [100] in terms of support by the basic process model without flow control and the impact of executing the pattern with flow control.

As can be seen from Table 4.4, almost all of the original 20 patterns [119] are supported by the basic model. Moreover, all of them work equally well when flow control is used to execute the pattern. The newer patterns (21–43) are not well supported. However, the extensions introduced in this chapter have no negative impact on these patterns either but rather allows the support of more patterns:

	Pattern	Basic	Extended
1	Sequence	+	+
2	Parallel Split	+	+
3	Synchronization	+	+
4	Exclusive Choice	+	+
5	Simple Merge	+	+
6	Multi-Choice	+	+
7	Structured Synchronising Merge	+	+
8	Multi-Merge	+	+
9	Structured Discriminator	+/–	+/–
10	Arbitrary Cycles	+	+
11	Implicit Termination	+	+
12	Mult. Inst. without Synchronization	+	+
13	Mult. Inst. with a Priori Design-Time Knowledge	+	+
14	Mult. Inst. with a Priori Run-Time Knowledge	+	+
15	Mult. Inst. without a Priori Run-Time Knowledge	–	–
16	Deferred Choice	–	–
17	Interleaved Parallel Routing	–	–
18	Milestone	+	+
19	Cancel Activity	+	+
20	Cancel Case	+	+

Table 4.4: Workflow control patterns (1–20) supported by the basic and extended process models.

persistent trigger (24), *generalized AND-join* (33) and *thread split* (42). These patterns are discussed in more detail in the following after a short comment on the *multiple instances* patterns.

Multiple instances

Table 4.4 indicates the support of *multiple instances* patterns (12–14) although these features have not been introduced as part of our basic model. However, the JOpera system, which has been extended with the flow control mechanism as part of this thesis, does support these patterns. In order to demonstrate the compatibility of the flow control semantics with as many patterns as possible we chose to include these features in the evaluation even though they are not described in the basic process model.

In JOpera, the execution of multiple instances of a task takes place as part of the actual execution of the task. All instances are synchronized before control is passed to successor tasks. Therefore, a multiple instances task looks like an ordinary task to the rest of the process. Hence, since this type of task execution is transparent to other tasks and has no influence on the inter-task communication, flow control has no impact on the execution of single instances of a task.

Pattern		Basic	Extended
21	Structured Loop	–	–
22	Recursion	+	+
23	Transient Trigger	+	+
24	Persistent Trigger	–	+
25	Cancel Region	–	–
26	Cancel Mult. Inst. Activity	–	–
27	Complete Mult. Inst. Activity	–	–
28	Blocking Discriminator	–	–
29	Cancelling Discriminator	+	+
30	Structured Partial Join	+/–	+/–
31	Blocking Partial Join	–	–
32	Cancelling Partial Join	+	+
33	Generalised AND-Join	–	+
34	Static Partial Join for Mult. Inst.	–	–
35	Cancelling Partial Join for Mult. Inst.	–	–
36	Dynamic Partial Join for Mult. Inst.	–	–
37	Acyclic Synchronizing Merge	+	+
38	General Synchronizing Merge	–	–
39	Critical Section	–	–
40	Interleaved Routing	–	–
41	Thread Merge	–	–
42	Thread Split	–	+
43	Explicit Termination	–	–

Table 4.5: Workflow control patterns (21–43) supported by the basic and extended process models.

Persistent trigger

A *persistent trigger* (pattern 24) is a signal[4] which is targeted at a specific task of a process and is received from within the process or from the outside. As opposed to a *transient trigger*, a persistent trigger does not need to be reacted upon immediately but is preserved until consumed by the target task. To implement the persistent trigger the information preserving nature of flow control can be used. The trigger source can, e.g., be modeled by a multiple-output task which changes to the *Outputting* state whenever a trigger is received. The target task correspondingly has an *Outputting* dependency on the trigger source. Since flow control is employed, this trigger will not be lost and can be processed by the target task at any later point in time. Without flow control, persistent triggers are not possible because a trigger then might be overwritten by a subsequent one.

[4]This is not the same sort of signal as we use to control the execution of a task (Section 3.3.1). The term is used in an abstract sense in the definition of the pattern.

Generalized AND-join

The two properties of the *generalized AND-join* (pattern 33) are the synchronization of multiple control flows and the buffering of control flow signals from these branches in case the signals cannot be handled immediately.

In our model this corresponds to a task with an activator based on the AND operator (to synchronize the control flows). This means all predecessors must have terminated in order for the synchronizing task to be started. Additionally, flow control must be used in order to not lose any control flow signals from the predecessor tasks in case such a task terminates multiple times while the synchronizing task is waiting for all predecessors to terminate. Clearly, this pattern can be supported only when flow control (and possibly buffering) is used. Otherwise, activations from predecessors may be lost because of the output overwriting problem.

Thread split

In a *thread split* (pattern 42) multiple threads of control are created at a certain point in a process. The number of threads to be created is specified at design-time.

This pattern can be directly modeled by the multiple-output feature introduced in Section 4.2.3. Every time a multiple-output task provides output a new thread is thereby created which continues along the control flow dependencies in the process. A simple counter component which generates a sequence of numbers can be used as task implementation in order to spawn the threads.

4.8 Discussion

This chapter has introduced two extensions to the basic process model: multiple-output tasks and flow control.

The multiple-output feature (Section 4.2.3) is a non-invasive extension to the process model. It is fully compatible with the original model as it builds on existing concepts. From a modeling point of view, the feature only introduces the *Outputting* state which is treated like any other state. This means that multiple-output tasks can easily be integrated into existing processes.

We have shown how to support a new type of processes, pipelined processes, using the original basic process model defined in Chapter 3[5]. Since we have not changed the model, we have tried to maintain the semantics of a task definition which consists of three parts: activator, data-condition and action.

The activator defines the states which predecessors tasks must have reached before the task can be started. The states of tasks indicate the progress the control flow in the process. Thus, the activator is a means to synchronize the threads of control arriving at a task. When employing flow control, markers are used to ensure the unambiguity of the predecessor states. This allows the different threads of control

[5]We have indeed extended the basic model with multiple-output tasks (Section 4.2.3). However, the pipeline capabilities developed in this chapter are independent of the multiple-output feature. A pipeline can also be created using a loop (Section 4.2.2).

which continuously arrive at a task to be properly distinguished. When the activator is satisfied, i.e., the incoming threads of control have been synchronized, the flow control markers are updated to acknowledge the arrival of control at the task.

The information provided by predecessor tasks is then used to evaluate the optional data-condition. The evaluation of this condition is not influenced by flow control. Only if the condition is satisfied, the task is ready to be started. Hence, with or without flow control, the data-condition filters the task input before it is possibly used to start the task.

In summary, the activator specifies how the control flow arriving at a task is synchronized and the data-condition is then used to filter the received input. The change due to flow control is that without flow control the difference between the activator and the data-condition is only artificial. Both expressions are evaluated on the process state. With flow control, however, the activator gains an event-based behaviour (Section 4.7.1) while preserving the semantics of synchronizing the control flow.

The execution semantics developed in this chapter makes it safe to employ pipelines in processes. As with microprocessors, the use of a pipeline improves the throughput of a system. However, a pipelined workflow process also exposes the same problems as a pipelined microprocessor. When designing a pipelined process the following issues therefore need to be considered.

- *Task execution times* For an optimal performance the stages in a pipeline must have equal execution times. If this is not the case, flow control makes sure no data is lost. However, the more the execution times vary (between tasks or over time for the same task) the lower the overall performance of the pipeline. If it is known that the execution times of task will vary significantly over time, then the arising temporary bottleneck can be alleviated by employing buffers.

- *Parallel branches* When a pipeline is split into several branches which are merged further down, care should be taken that the throughput of the branches is balanced. Otherwise, since the branches are merged, the overall pipeline will run at the speed of the slowest branch.

- *Serial processing* The stages in a pipeline process stream elements sequentially. That is, no parallel processing of elements (superscalarity) is done. Thus, if a pipeline is used to process data from multiple independent streaming services, it must be ensured that the pipeline is able to keep up with total rate at which data arrives. Otherwise, a separate instance of the pipelined process should be employed for each service.

- *System capacity* In order for pipelining to be effective, it is important that the overall system executing the workflow (navigator, dispatcher, service providers) has the capacity to execute all the tasks in a pipeline concurrently.

- *Hazards* If tasks in a pipeline have dependencies which are external to the process and which are not captured by the process model, then this can lead to *pipeline hazards*. In microprocessors such hazards are avoided by forwarding data and/or by stalling the pipeline [18]. However, since tasks in a pipelined workflow perform complex operations solutions like data forwarding cannot always be applied. In some situations, creating an *isolated section* in the pipeline can help. This is a section in which pipelining is "turned off" and only one stream element is being processed at any time[6]. Such isolated sections can be created by wrapping the contained tasks in a subprocess and including the subprocess in the pipeline. To the pipeline, the isolated section looks like a normal pipeline stage, but inside the subprocess non-pipelined execution takes place.

A solution to unbalanced task execution times and the serial processing restriction would be the support of superscalarity for task executions (Section 4.3.1). This corresponds to the *multiple instances without a priori run-time knowledge* workflow pattern [100]. With superscalarity, instances of a task can be created as needed in order to process multiple stream elements in parallel. This removes the bottleneck effect of a constantly slow task.

[6]This can be considered a special case of the *interleaved parallel routing* workflow pattern [100]

5 System Implementation

This chapter describes the changes that were applied to the compiler and the runtime system of JOpera (Section 3.3) to implement the ideas described in Chapter 4. We start by extending the compiler to integrate the flow control marker mechanism when compiling a process model (Section 5.1). Then, in Section 5.2, the communication between the navigator and the task execution part of the system is discussed . The system is extended such that results can be streamed from a service into a process without interfering with the flow control in the process. In Section 5.3, we introduce the infrastructure to support the buffered data transfer between tasks in a process as described in Section 4.5. The management of such buffers is discussed in Section 5.4.

5.1 Compiling flow control

In Section 4.4.2 we introduced the marker mechanism to control the flow of information between the tasks of a process. This mechanism is an extension of the control flow decisions which are made during navigation. The mechanism uses markers between every pair of tasks with a control flow dependency to track whether there is output from the corresponding predecessor task which has not been processed by the successor task. A task is not allowed to re-execute before its previous output has been consumed by all its successors.

In order to integrate flow control into the navigation algorithm, the process compiler must be extended. The changes arise mainly from the fact that without flow control a task can be compiled in isolation, whereas the code for enforcing flow control cannot be generated from the information contained in the task itself.

When generating code without flow control, the control flow code of a task (which evaluates the starting conditions) can be compiled by looking only at the activator of the task. The activator contains all dependencies towards predecessors and the task is activated when these predecessors have reached a certain state. Thus, the control flow of a process can be compiled in one pass by iterating over its tasks and compiling each task independently[1].

When the navigation code should include the flow control marker mechanism, a process can no longer be compiled in a single pass. The activators do not contain enough information and an additional pass is needed before the code can be generated. The reason is that the evaluation of a task's starting condition has been augmented to include the inspection of flow control markers. The markers to be inspected depend not only on the predecessors (which can be derived from the activator) but also on the successors of the task.

[1]The data flow of a process is analyzed in a preceding phase (Section 3.3.2)

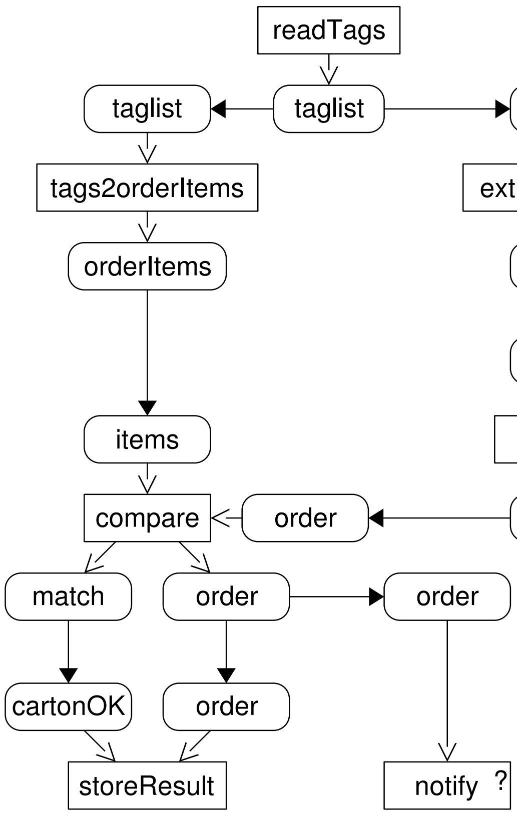


Figure 5.1: Data flow of the order verification process.

To determine the successors of the tasks in a process, the process model must be preprocessed. In this phase, the activators of all tasks are analyzed and their predecessors and successors are determined. For every predecessor P referenced in the activator of a task T, P is recorded as predecessor of T and T as successor of P. When the navigation code is generated in the next phase, the compiler uses the information about predecessors and successors of a task to add the flow control related parts such as checking and manipulating markers. All the places where flow control checks are added have been described in Section 4.4.2. The following example illustrates the effect of incorporating flow control in the compiled process model.

Example 5.1: Compiled flow control

In this example, we show the compiled navigation code of the carton inspection process (Example 3.1) when it is compiled with flow control. The data flow of the process is repeated in Figure 5.1.

In summary, the process does the following. The first task, readTags, obtains a list of RFID tags which indicates the contents of the carton just scanned. This list is used for two purposes. On the one hand, tags2orderItems translates the tag numbers to item numbers and compiles a description of the carton content. On the other hand, the identifier of the box is picked from the tag list (extractCartonID) and the description of the corresponding order is fetched from a database (getOrder). Now the effective contents of the box are compared with the information from the order database (compare) and storeResult updates the order in the database with the outcome of the comparison. In case of a mismatch, the carton does not contain the correct items and a notification is triggered by the notify task.

Actually, we use a slight variation of the process described in Example 3.1. In order to create a pipeline in the process, the first task, readTags, is now a multiple output task. It continuously provides RFID readings to the process. Correspondingly, the two successor tasks tags2orderItems and extractCartonID now have an *Outputting* dependency on readTags. With the readings streaming into the process, a single process instance can be used to verify multiple orders.

Since the basic structure of a compiled process has already been described (Example 3.2), in this example we show mainly the differences caused by the flow control checks. The code uses combined markers (Section 4.4.4).

As with the states of the tasks, the flow control information, i.e., the markers, is cached in local variables and therefore needs to be preloaded at the beginning of the navigation method. We show only the markers between the first task and its successors, the others are loaded in the same way.

```
// FLWCTRL: load marker TaskPair[readTags -> extractCartonID]
boolean marker_readTags_extractCartonID = Boolean.TRUE.equals(Memory
  .Load(MakeAddress(Context_TASK_readTags, Box.System,
    "FlowControl_extractCartonID")));
// FLWCTRL: load marker TaskPair[readTags -> tags2orderItems]
boolean marker_readTags_tags2orderItems = Boolean.TRUE.equals(Memory
  .Load(MakeAddress(Context_TASK_readTags, Box.System,
    "FlowControl_tags2orderItems")));
```

When the process is in the *Initial* state, the start task of the process, readTags, is started. Start tasks naturally do not have an activator to be checked (except for the process being *Initial*). They might have a data-condition defined which, however, is not the case in our example. Since readTags has no incoming dependencies, also no flow control checks need to be done.

```
 Exec.Start(Context_TASK_readTags, InputParams, SystemInputParams);
```

Whenever the execution of the readTags task produces a result, this data has to be stored in the task's output parameters. This applies to final results of a task (in the *Finishing* state) as well as intermediate results (in the *Outputting* state). In both cases, the task has been put in the *Finishing* state to indicate to the navigator that the task output should be stored and possibly data transfers should be executed (if scheduled by the data flow compiler). In this step flow control markers need not be checked because it was ensured before the task was started that the output parameters can receive data.

```
  if (State_readTags == State.FINISHING) {
   Memory.Store(MakeAddress(Context_TASK_readTags, Box.Output,
     "taglist"), (Serializable) Results.get("taglist"));
```

After storing the (intermediate) result of the task execution, the task leaves the *Finishing* state. According to the state diagram in Figure 3.12, the task can now enter the *Finished* or *Outputting* state. In order for the navigator to take this decision a flag has been set in the address space of the task. This flag is read and the state changed accordingly.

```
MemoryAddress stream_address = MakeAddress(
  Context_TASK_readTags, Box.SystemOutput,
  Box.Stream);
if (Memory.Load(stream_address) != null) {
 Memory.setState(Context_TASK_readTags,
   State.OUTPUTTING);
 State_readTags = State.OUTPUTTING;
 Memory.Store(stream_address, null);
} else {
 Memory.setState(Context_TASK_readTags,
   State.FINISHED);
 State_readTags = State.FINISHED;
}
```

When using flow control, other tasks in the process must be made aware of this state change of readTags (to *Outputting* or *Finished*), i.e., flow control information towards these tasks has to be updated. Since the task has two successors, tags2orderItems and extractCartonID, the corresponding two markers have to be set to indicate that new information is available from the task. This is done immediately after the state of readTags has been changed.

```
// FLWCTRL: set outgoing markers
// FLWCTRL: set TaskPair[readTags -> extractCartonID]
marker_readTags_extractCartonID = true;
Memory.Store(MakeAddress(Context_TASK_readTags,
  Box.System, "FlowControl_extractCartonID"),
  Boolean.TRUE);
// FLWCTRL: set TaskPair[readTags -> tags2orderItems]
marker_readTags_tags2orderItems = true;
Memory.Store(MakeAddress(Context_TASK_readTags,
  Box.System, "FlowControl_tags2orderItems"),
  Boolean.TRUE);
```

As an example of a starting condition which has been augmented with flow control checks, we use the compare task which is also a synchronization point for the control flow from tags2orderItems and getOrder.

With flow control, tasks can not only start from the *Initial* state but also from any final state (Section 4.4.5).

```
if (State_compare == State.INITIAL || State_compare.isFinal()) {
```

As the first step of the starting condition, flow control requires that the ready markers towards the successors, storeResult and notify, are inspected. If one of these markers is set, the output of compare has not been entirely consumed and the task may not be started.

```
if (!marker_compare_notify && !marker_compare_storeResult) {
```

Then, the activator can be evaluated: `Finished( getOrder ) AND Finished( tags2orderItems )`. The evaluations of the predecessor states are combined with a conjunction (`&&`) which means that the control flow is synchronized a this point. Both predecessors must have reached the *Finished* state. Additionally, to enforce flow control, in the activator expression every state predicate evaluation (`State_XYZ == State.ABC`) is “filtered” by an inspection of the corresponding freshness marker. This enforces that only fresh information about the state of a predecessor is used to evaluate the activator.

```
if (((State_getOrder == State.FINISHED) &&
marker_getOrder_compare)
  && ((State_tags2orderItems == State.FINISHED) &&
  marker_tags2orderItems_compare)) {
```

If the activator is satisfied, data for the input parameters are collected and the task is started (not shown). Then, the flow control markers are updated to reflect that the information coming from the predecessor tasks has been consumed.

```
// FLWCTRL: clear incoming markers
if (marker_tags2orderItems_compare) {
 // FLWCTRL: clear TaskPair[tags2orderItems ->
 // compare]
 marker_tags2orderItems_compare = false;
 Memory.Store(MakeAddress(
   Context_TASK_tags2orderItems,
   Box.System, "FlowControl_compare"),
   Boolean.FALSE);
}
if (marker_getOrder_compare) {
 // FLWCTRL: clear TaskPair[getOrder -> compare]
 marker_getOrder_compare = false;
 Memory.Store(MakeAddress(Context_TASK_getOrder,
   Box.System, "FlowControl_compare"),
   Boolean.FALSE);
}
```

In case the activator was not satisfied, the deactivator is evaluated to decide whether the task should be marked *Unreachable*. As with the activator, the evaluation of a predecessor's state is combined with the corresponding freshness marker to enforce proper flow control.

```
} else if ((((State_getOrder != State.FINISHED) &&
(State_getOrder
  .isFinal())) && marker_getOrder_compare)
  || (((State_tags2orderItems != State.FINISHED) &&
  (State_tags2orderItems
    .isFinal())) && marker_tags2orderItems_compare)) {
```

If the deactivator indicates that the task is unreachable, its state is set accordingly and, to update the flow control information, the incoming markers are cleared and the outgoing markers are set (see above).

```
Memory.setState(Context_TASK_compare,
  State.UNREACHABLE);
```

If neither the activator nor the deactivator is satisfied, there is the possibility that a predecessor has reached a non-final state which is not of interest to the compare task. When using flow control, such states need to be discarded in order to not congest the pipeline (Section 4.4.2). Only the connection from tags2orderItems is shown.

```
// FLWCTRL: purge incoming flow
// FLWCTRL: check TaskPair[tags2orderItems -> compare]
if (marker_tags2orderItems_compare) {
 if (!((State_tags2orderItems == State.FINISHED) ||
 State_tags2orderItems
   .isFinal())) {
  // FLWCTRL: clear TaskPair[tags2orderItems ->
  // compare]
  marker_tags2orderItems_compare = false;
  Memory.Store(MakeAddress(
    Context_TASK_tags2orderItems, Box.System,
    "FlowControl_compare"), Boolean.FALSE);
 }
}
```

To complete the execution cycle of a task we go back to readTags which we left in the *Outputting* state. When employing flow control, the task cannot leave the *Outputting* state before all the task's successors have taken notice of the intermediate results it has provided. If the corresponding markers indicate that this is the case, readTags can return to *Running* in order to produce new results.

```
if (State_readTags == State.OUTPUTTING) {
 if (!marker_readTags_extractCartonID
   && !marker_readTags_tags2orderItems) {
  Memory.setState(Context_TASK_readTags, State.RUNNING);
```

As will be discussed in the following section, the task execution subsystem is also involved in the flow control. When a task returns from *Outputting* to *Running*, this means that the corresponding subsystem may continue to produce output. Therefore, the subsystem is informed with a signal that the task has returned to *Running*.

```
  Exec.signalJob(Context_TASK_readTags,
    APIConsts.SIGNAL_UNBLOCK);
 }
}
```

5.2 Navigator–subsystem communication

In this section, we discuss the communication between the navigator and the task execution part of the system. First, we extend the navigator-subsystem protocol to support the streaming of results into a process instance. Then, we discuss how to control this stream of results such that the process instance is not overrun with data which would compromise the flow control inside the process instance.

5.2.1 Multiple-output protocol

There are two ways a task can provide output: a single output or multiple outputs. The invocation of a request-response service results in a single response from the service. This response is provided in the output parameters of the task as it terminates in the *Finished* state. A service may also return a stream of messages as response. To output such a stream, the original task concept has been extended with multiple-output tasks, i.e., in addition to providing output on termination, a task can produce an arbitrary number of intermediate outputs (Section 4.2.3). To do so, the task changes to the *Outputting* state which indicates to the rest of the workflow that intermediate output is available in the task's output parameters. Then the task returns to *Running*. This is repeated for every message in a stream until the task terminates. In the following, we describe the changes which had to be made to the JOpera runtime in order to support tasks which produce multiple outputs.

In the runtime system the multiple-output concept is mapped to the invocation of a streaming service by the task execution subsystem. Since the subsystem understands the protocol of the service, it can distinguish between intermediate and final results delivered by the service. However, the runtime system itself needs to be made aware of the new interaction style. Figure 5.2 (left) shows the notifications between the navigator and the subsystem for a standard single-output task. First, the navigator submits a task execution job to the subsystem (via the dispatcher). Second, after the invocation of the service, the subsystem notifies the navigator of the termination of the job. This interaction provides no means for notifying about intermediate results.

In order to support the delivery of intermediate output, the protocol between subsystems and the navigator has been extended (Figure 5.2 (right)). The extended protocol supports the invocation of request-response as well as streaming services. The first step of the protocol remains the same, i.e., the navigator submits a task execution job to the subsystem which in turn invokes the service. In case the response received from the service is an intermediate result, the subsystem sends an "output" notification to the navigator together with the intermediate result. This new notification type indicates to the navigator that the task has not yet terminated but should provide the intermediate output using the *Outputting* state. Sending the intermediate output using an additional notification type has the advantage that the queue-based communication mechanism between the dispatcher and the navigator can be reused. The new notification type is made available to subsystems through the new `notifyOutput` API operation (Figure 5.3). When a service sends a single result or it has sent the last in a sequence of results, the subsystem sends the navigator a "termination" notification together with this single or last output of the task execution. The navigator then stores the output and sets the task to *Finished*.

5.2.2 Flow control at the process boundary

The navigation code generated by the compiler makes sure that pipelining can safely be used inside the process by using flow control between the tasks. As a result,

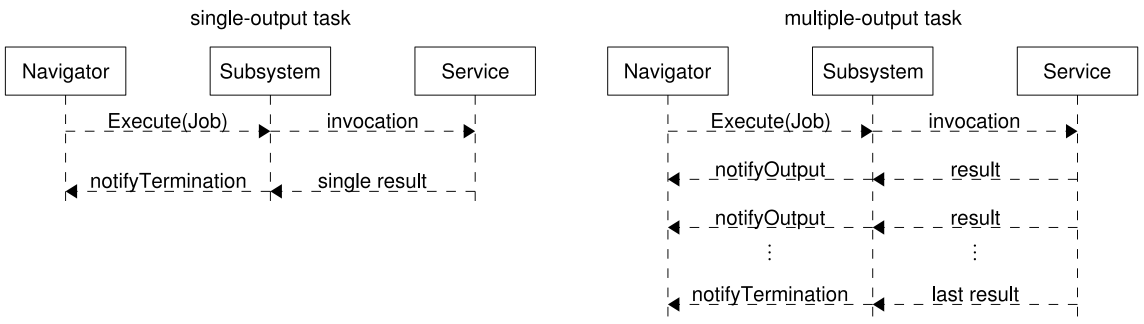


Figure 5.2: Interaction between navigator and subsystem for traditional single-output tasks and multiple-output tasks.

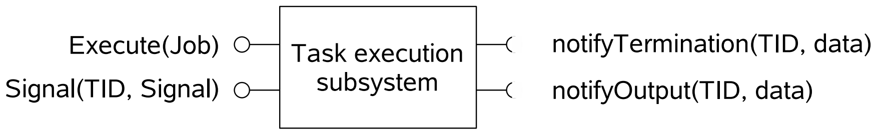


Figure 5.3: Functionality provided and required by a task execution subsystem.

a pipeline is slowed down to the speed of its slowest task. This means that all tasks upstream of the bottleneck task are blocked while waiting for their output to be consumed by the next step in the pipeline. This applies also to the multiple-output task at the head of the pipeline. Such a task is driven by the underlying task execution subsystem (Figure 5.4) which outputs data at the rate at which it is received from the invoked streaming service (Section 5.2.1). However, the subsystem must be prevented from producing output before the previous output has been consumed by all successor tasks. Otherwise, the new output overwrites the previous one too early. In the remainder of the section we develop a solution to control the rate at which a subsystem outputs data.

Similarly to Section 4.3.1, where we discussed collisions in pipelines, there are several alternative solutions to this problem. The output from the subsystem can be discarded or buffered, or the subsystem can be blocked. The only difference to Section 4.3.1 is the location where the discarding, buffering or blocking takes place. In this case, it takes place between the task execution and the navigator, i.e., before the data has been written to output parameters. In the case of Section 4.3.1, it takes place inside the process between the affected task and its successors, i.e., after the data has been written to the output parameters.

We chose to use the blocking approach, thereby extending the flow control taking place inside the process all the way to the subsystem. This allows the subsystem and therewith any invoked service to take part in the flow control. Discarding or buffering data can still be done inside the process. Equally, the subsystem can use one of these approaches internally if it is prevented from delivering the data to the process.

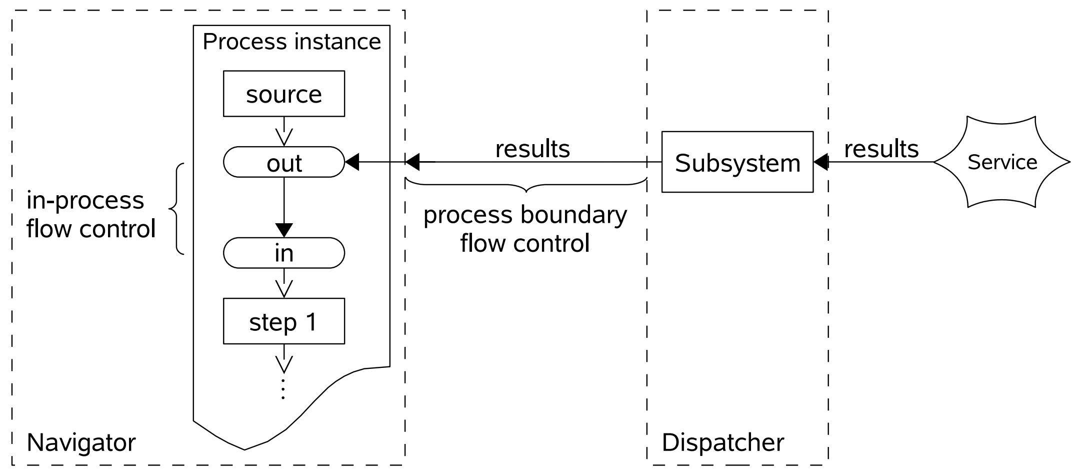


Figure 5.4: Flow control is also necessary between the subsystem and the process to prevent the subsystem from overwriting data in the process.

Flow control protocol

To extend the flow control from the process to the task execution subsystem, the protocol between the navigator and the subsystem needs to be extended. In the previous section, we added support for the subsystem to send intermediate results to the navigator. Now, this protocol is extended further: the subsystem must be instructed when it is safe to provide the navigator with results and when not.

In the navigator we can make use of the fact that a task stays in the *Outputting* state as long as its output has not been consumed by all successors, i.e., its outgoing readiness markers have not been set (Section 4.4.2). As long as the task is in the *Outputting* state, it must not produce further output. Otherwise, its output parameters would be overwritten too early. Producing new output does not necessarily mean that the task is put into the *Outputting* state again, but also includes task termination. Here the task's state and output parameters are equally overwritten with new values. When the task becomes *Running* again, it is however safe to produce new output. Hence, if the task execution subsystem can be blocked while the task is in the *Outputting* state, then the subsystem will only deliver data when it is safe to do so.

The navigator and the task execution are executed in independent threads and communicate through queues (Section 3.3.1). The messages which are exchanged are job submissions and signals from the navigator to the dispatcher (which delegates to the subsystems) and notifications of intermediate or final output from the subsystem to the navigator (Section 5.2.1). In order to synchronize the task execution subsystem with the task state, this protocol is extended as follows, using the existing communication mechanism between navigator and dispatcher (Figure 5.5). Whenever the subsystem sends output to the navigator as part of a job, this job is automatically marked as *blocked.* The job remains blocked until it is released by the navigator. While the job is blocked, the subsystem is not allowed to deliver

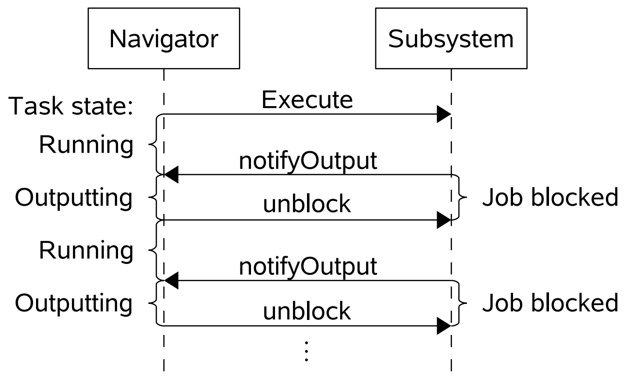


Figure 5.5: Protocol between navigator and task execution subsystem for extending flow control from the process to the subsystem. As long as the task is *Outputting*, the job is blocked and the subsystem may not produce further output.

to the navigator further data which may arrive from the invoked service. As soon as the task leaves the *Outputting* state, the navigator sends an *Unblock* signal[2] to release the job (Example 5.1). The subsystem may then provide more output to the navigator. Thus, the job is blocked exactly when the task is in the *Outputting* state.

Because the subsystem is not allowed to produce output while the job is blocked, this mechanism effectively prevents the subsystem from sending output to the navigator which would overwrite the output parameters while the task is *Outputting*. As a result, subsystems are integrated in the flow control chain in a general way.

An API for streaming subsystems

In order to guarantee proper flow control inside processes, the runtime system has to account for misbehaving subsystems which disregard flow control. This means that the flow control must be enforced by the subsystem API and cannot be left to the subsystems. Subsystems continue to invoke the `notifyOutput` and `notifyTermination` API operations as before. However, we have changed the behaviour of these methods to adhere to the flow control protocol described above. This also simplifies the development of a streaming subsystem by moving common functionality into the framework. In the following we discuss the implications of flow control for multiple-output subsystems and develop a subsystem API which enforces flow control without constraining the architecture of the subsystem.

A simple approach to enforce flow control when a subsystem is providing output is to suspend the subsystem thread as long as the corresponding job is blocked. When the job is unblocked by the navigator the thread is released and further data may be output. Figure 5.6 illustrates the approach. It should be noted that since flow control is handled by the subsystem API, the *Unblock* signal is processed by

[2]Thanks to the extensibility of the signalling mechanism, this new signal type can easily be added to the system.

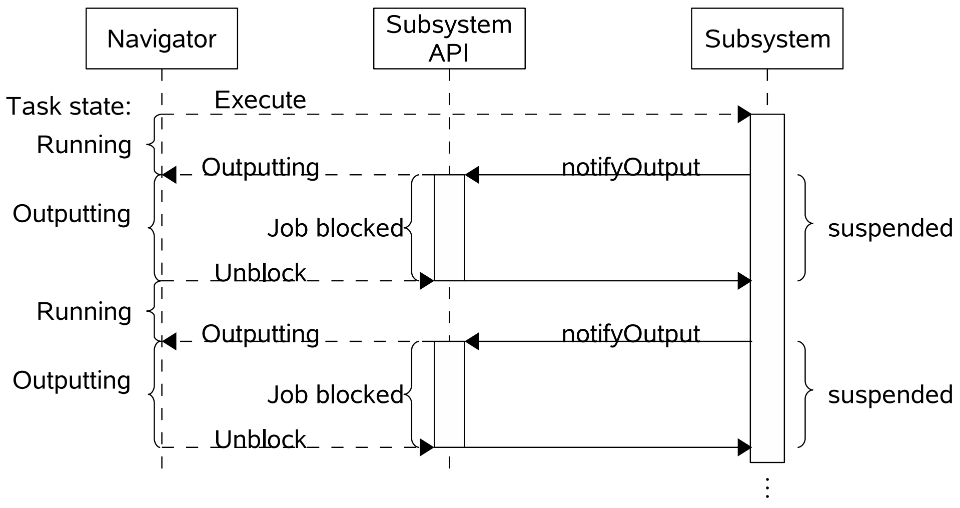


Figure 5.6: Simple approach to enforcing flow control. The subsystem thread is suspended until the task output has been consumed.

the API. Therefore, as opposed to other signals, the *Unblock* signal actually never reaches the subsystem.

This approach, however, has certain drawbacks. First, since the thread is suspended while the task is *Outputting*, the receipt of a message from the invoked service and the distribution of this output inside the process are strictly serialized. This is unnecessary as the subsystem could use the time while it is suspended to perform operations which do not send output to the navigator (e.g., receiving the next message from the service). Second, the approach does only work for single-threaded subsystems. In the case of a subsystem which uses multiple threads to communicate with external applications, the approach does not properly enforce flow control. Instead of blocking a thread *after* it has delivered data to the navigator, the permission to deliver output for a task must be negotiated between the threads *before* the data is sent to the navigator. Otherwise, while one thread is blocked during the delivery of data, another thread would be admitted to send its data to the navigator, thereby bypassing flow control and overwriting any previous output of the task.

Thus, to guarantee proper flow control with multi-threaded subsystems and without unnecessarily blocking threads the following approach is used. Before a subsystem thread can deliver output to the navigator it must acquire the exclusive right to do so. If the job is currently blocked, the thread is suspended until the job has been unblocked before it can acquire the right. As soon as a thread has gained the permission to send output, the job is marked as blocked and the job output is submitted to the navigator. After the data has been sent, the thread looses the possession of the job and, like any other thread in the subsystem, has to reacquire the outputting-right before delivering further output. Figure 5.7 illustrates this protocol.

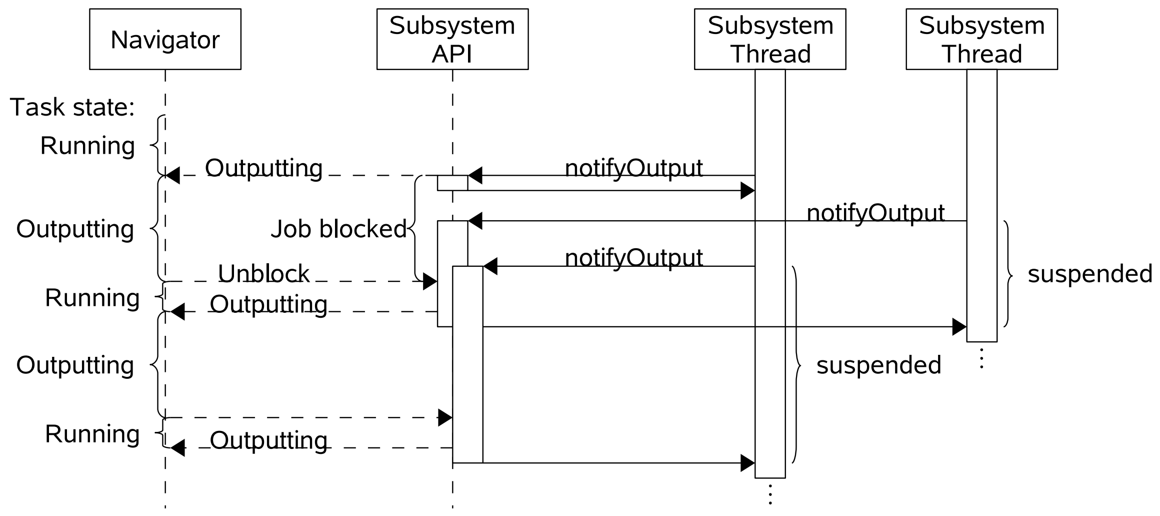


Figure 5.7: Enforcement of flow control for multi-threaded subsystems. A subsystem thread needs to acquire the exclusive right to send output before notifying the navigator. After delivering the output, while the job is blocked, the thread may continue executing until it tries to deliver further data.

When flow control is extended to the subsystem, a thread in the subsystem which tries to send output to the navigator may become suspended because of the job being temporarily blocked. This means that the thread is not able to immediately continue the communication with the invoked service, e.g., by acknowledging the receipt of a message. If timeouts can easily occur in the communication with the external entity, e.g., a person waiting for a response, the subsystem must avoid the possibility of being blocked for an indefinite period of time.

The subsystem could solve this problem by employing a thread which is dedicated to delivering data to the navigator and which might be blocked during delivery. Another thread then takes care of more time-critical operations. However, this complicates the development of a subsystem and increases its resource requirements.

Instead, we complement the existing API operations with the *non-blocking* variants `tryNotifyOutput` and `tryNotifyTermination`. Basically, these operations have the same functionality as `notifyOutput` and `notifyTermination`, respectively. However, if the permission to deliver output cannot be granted immediately, the methods return with an error. The subsystem can then, depending on the service type it is interacting with, e.g., retry to send output later or even abort the delivery and report an error to the service.

In addition to the non-blocking delivery method we also provide an `isBlocked` operation. This method reports whether the job is currently blocked. This method is of little use with multi-threaded subsystems (the blocked-state of a job might be changed at any time by another thread), but for a single-threaded subsystem the method provides useful information about the state of the task in terms of flow

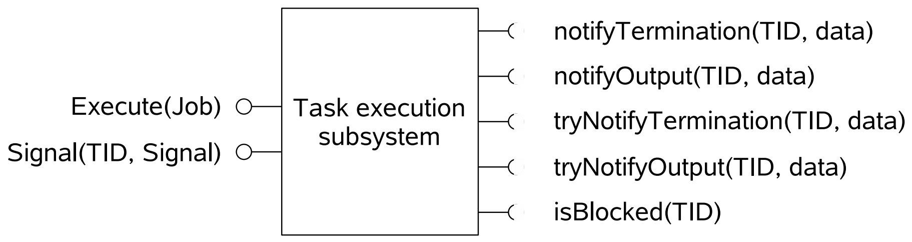


Figure 5.8: Functionality provided and required by a task execution subsystem. The new `try-` methods allow non-blocking data delivery.

control. Figure 5.8 gives an overview of the subsystem API including the extensions for flow control. A summary of the API operations' behaviours is given in Table 5.1.

5.3 Buffer infrastructure

Based on the system with flow control presented in the previous sections, this section describes how to extend the runtime system to support a buffered information exchange between tasks in a process. After describing the buffering infrastructure functionality, we discuss different implementation alternatives.

As discussed in Section 4.3.3, buffers are used to transfer (control and data) information between tasks. When a task produces output, this is appended to one or more buffers. Similarly, the input needed to start a task is fetched from one or more buffers (Section 4.5.2). Buffers are an extension of the markers used in the flow control marker mechanism and can directly be used with this algorithm (Section 4.5.1).

5.3.1 Buffer interface

Section 4.5 explained how buffers are used in the navigator and implicitly formulated the following requirements. The outcome of a task is appended to buffers and the input for executing a task is retrieved and thereby removed from buffers. In this manner data is transferred between parameters of different tasks. The first element of a buffer can also be inspected without removing it from the buffer. This is necessary so the navigator can inspect the state of a process instance without affecting it. In addition to manipulating the buffer, a buffer must be able to report whether it is full and whether it is empty. With this knowledge about the capacity of a buffer, the navigator makes flow control decisions. Based on these requirements, the following describes the operations provided by a buffer and their semantics.

- `append` This operation appends an element to the end of the buffer. The element is tagged with a timestamp.

Subsystem Operations	
`Execute(Job)`	Instructs the subsystem to execute according to the specified job description.
`Signal(TID, Signal)`	Controls a running job according to the specified signal (*Kill*, *Suspend*, *Resume*).
API Operations	
`notifyTermination(TID, data)`	Notifies the runtime system that the job for the specified task instance has terminated (successfully or through failure) and accepts the result data of the execution. If the job is currently blocked, the operation blocks until the job is unblocked.
`notifyOutput(TID, data)`	Notifies the runtime system that the job for the specified task instance makes the specified intermediate result data available. If the job is currently blocked, the operation blocks until the job is unblocked.
`tryNotifyTermination(TID, data)`	Does the same as `notifyTermination` except that if the job is blocked, then the operation returns with an error.
`tryNotifyOutput(TID, data)`	Does the same as `notifyOutput` except that if the job is blocked, then the operation returns with an error.
`isBlocked(TID)`	Indicates whether the specified job is currently blocked.

Table 5.1: Description of the operations defined by the subsystem API.

- `remove` This operation reads and removes the first element of the buffer. Elements are retrieved from the buffer in the same order as they were appended (*first in first out*).
- `peek` This operation reads the first element of the buffer, as `remove` does, but without removing it.
- `isFull` This operation tells whether the buffer has reached its capacity and cannot hold more elements. If the buffer is full, no further elements must be appended.
- `isEmpty` This operation tells whether the buffer contains any elements which can be retrieved. If the buffer is empty, `remove` or `peek` must not be invoked.

The elements which are held in a buffer are the outcomes of task executions. A task outcome includes the values of all output parameters and the task state in which they were produced (*Outputting*, *Finished* or *Failed*).

Every time an element is appended to a buffer a timestamp with the current time is attached to the element. This timestamp can be used by the navigator to resolve conflicts at input parameters with multiple bindings (Section 4.5.3). In such a case, the data for the input parameter can be taken from multiple buffers and the process' navigation algorithm must decide which buffer should be chosen. The timestamp indicates the time at which, in the non-buffered case, data transfers would have taken place (Section 4.5.2). Choosing the buffer element with the highest timestamp yields the desired "last writer wins" semantics.

The `append`, `remove` and `peek` operations are non-blocking. That is, if the buffer is full or empty, respectively, the operations do not block until the buffer state has changed, but rather return immediately with an error. There are two reasons for this. First, blocking is not necessary because, as part of the flow control algorithm, the navigator always checks the capacity of a buffer before appending to or reading from it. This corresponds to checking the state of markers before starting a task (Section 4.5.1). Second, the navigator must never be blocked indefinitely by any operation it invokes on the state storage. Since the storage is managed solely by the single-threaded navigator, this would lead to a deadlock.

5.3.2 Buffer component

In this section we motivate the extension of the JOpera state storage with a new buffer component. We will refer to the existing storage model as the *JOpera memory* since it resembles very much a conventional random access memory where *memory cells* are accessed by specifying an address.

The JOpera state storage contains the state of all process instances. A great part of this state is also, temporarily, held in buffers which transfer data between tasks. This rises the question whether it is feasible to integrate the buffering functionality in the existing storage without changing the storage model (memory cells addressed with (*TID*, *category*, *parameter*) tuples). Keeping this model would allow to the navigation algorithm to remain unchanged, because the interface to the state storage remains the same. However, the buffering model introduces several new features which cannot all be simulated in the memory model.

Time A buffer may hold several values of a parameter which were produced over time. When an element is read from the buffer this is not the element which was last appended unless the buffer contains only one element. This behaviour can be added without changing the memory interface, by using a hidden buffer and turning writes and reads to the memory into `append` and `peek` operations on the buffer, respectively.

Parameter correlation The values of output parameters from a task execution must be kept together so they can be assigned to other tasks' input parameters in a consistent way. This means the memory must distinguish the results of two distinct task executions by monitoring the sequences of write accesses to certain memory cells. This would not be difficult as the access to the memory is transacted. Every navigation round is wrapped in its own transaction and in a navigation round at most one result of a particular task is stored.

New operations The `remove`, `isFull` and `isEmpty` operations are a clear extension of the storage model. The `isEmpty` operation could be saved by having the memory return an unspecified state for tasks which forces the navigation algorithm to skip the corresponding task. This would yield the same behaviour as when navigation explicitly checks the emptiness of the buffer. The other two operations, however, cannot be substituted by changing the semantics of the memory.

Pairwise task communication In the memory model, a memory cell always belongs to a task instance. Therefore, the cell's address contains the TID of this task instance. The buffer model, however, extends the storage with a new dimension. Data contained in a buffer does not belong to only a single task instance (the output of which the data is), it belongs to the pair of tasks which is connected by the buffer. In the buffer model, when an output parameter value is read, it is always relevant by which task it is read because of the pairwise information exchange enforced by flow control. This additional dimension in the addressing of state is incompatible with the existing addressing memory model. There is no way for the memory to derive the task on behalf of which the navigator accesses a memory cell in order to deliver the data from the appropriate buffer. The second task must thus be specified by the navigator when accessing a memory cell.

Finally, it should be noted that the buffers are a replacement for the data transfers in a process – a buffer constitutes a data transfer which is possibly delayed. As such, a buffer is naturally associated with two tasks, the source and the destination of the transfer.

Because of the differences in the memory model and the buffer model, buffers need to be managed explicitly and cannot be hidden behind the existing memory

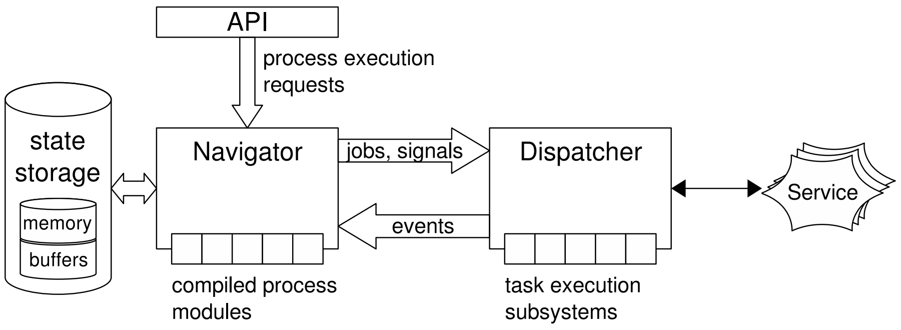


Figure 5.9: Architecture with Buffers.

interface. Figure 5.9 shows the system architecture with the extended state storage comprising the old memory part and the new buffer part. Every buffer is identified by the TIDs of the tasks it connects. For example, a buffer from task B to task A is identified by the ordered pair (TID_B, TID_A) where TID_B and TID_A are the respective TIDs.

The fact that much of the state of a process instance flows through buffers suggests that the memory component could even be replaced entirely by buffers. We choose not to follow that option. First, keeping the memory component and using buffers for data transfers reduces the necessary changes to the system. Concerning navigation, only the parts which are dealing with data transfers need to be accommodated to use buffers instead. Other aspects such as the assembly of task execution jobs do not need to be modified. Default values of parameters would not be possible without a memory to store the values in (Section 3.3.2). Second, existing tools rely on the state of process instances being readily available in the memory. Such tools include process monitoring (Section 3.3.3) and the recording of lineage information [104]. This also means that tools need to be extended for monitoring buffers.

5.3.3 Implementation of the buffer component

As the other components in JOpera, the buffer infrastructure is defined in an abstract manner and can be implemented in different ways, depending on the requirements of a certain deployment of the system. Here, we describe some possible implementation alternatives with different characteristics in terms of performance, persistence and the support for a distributed deployment with multiple navigators.

Main memory

The most light-weight approach is to keep the buffers in main memory. Fixed size buffers are implemented as circular buffers and linked lists can be used for variable size buffers. The advantages of this approach are the high performance and the independence of any external system (e.g., a database). However, there is no support for persistence and it is not possible to share the buffers between several navigators in a distributed deployment. Thus, this solution is best suited for development and testing purposes.

Message queues

Message queues are used to asynchronously transfer messages in distributed applications. Asynchronous means the sender and receiver of a message do not need to be connected to the queue at the same time. Messages are read from a queue in the same order as they were appended (FIFO). Typically, a queue stores its contained messages persistently to guarantee a reliable message delivery. Message queues are a particular form of message-oriented middleware (MOM) (see [3]).

Since message queues provide exactly the functionality needed by inter-task buffers, the mapping can be done in a straightforward manner. Every buffer is implemented by a message queue and the buffer operations are forwarded to the corresponding operations on the queue.

Message queues have the advantage that their content is stored persistently which is necessary in production deployments. Since message queues are tailored to a FIFO access pattern and often contain short lived messages, the performance of a queueing system can be optimized [5]. It is also possible to use a message queue server in a distributed deployment. The persistence comes at the cost of a lower performance compared to the in-memory approach.

Database

Many MOM systems use a standard relational database as their underlying message storage, e.g., [5, 68]. These systems provide functionality which is not needed in our case, such as filtering and routing of messages. Thus, it seems likely that a more light-weight queue management can be implemented on top of a database which yields a higher performance due to lower overhead. Such a solution has the same advantages and drawbacks as the message queue approach but might provide a higher performance.

Overlay buffers

With this approach, buffers are implemented on top of the JOpera memory. The idea is that a memory cell can hold arbitrary values and therefore also a buffer can be stored in memory cells. Since the memory is addressed hierarchically, the buffer is stored in the scope of either the predecessor or successor task of the buffer.

Basically, there are two ways to store a buffer. With the first approach, the entire buffer object is stored in one memory cell. This object contains the individual buffer elements and the necessary variables for managing the buffer (capacity, element count etc.).

Alternatively, the buffer can be mapped to several memory cells. With this approach, every buffer element is stored in its own memory cell. This approach lends itself to circular buffers where a certain range of addresses is reserved for the element storage. In addition to the buffer payload, the management variables are stored in separate cells. The operations on the buffer work the same way as with a normal main memory buffer. The difference is that the buffer properties are not stored in normal variables but have to be accessed in the JOpera memory.

The difference between the two alternatives is that in the first case the whole buffer object is read from memory in any case. If the buffer is manipulated (by appending or removing), the entire buffer also has to be written back. In the case where every property of the buffer is mapped, only the individual properties which are actually read or written are transferred from or to the memory.

Which alternative exposes the best performance depends on the characteristics of the underlying memory implementation. If the overhead of accessing a memory cell is small (short delay), then the mapping approach is better. Otherwise, it may be better to transfer the whole buffer and perform the updates locally and then write back the buffer. Specially, if it is cheap to transfer large quantities of data from/to the memory (high throughput), this approach may be better. Also, the way a buffer is transferred from/to the memory depends on the caching strategy of the memory implementation. This may influence the actual number and size of memory accesses that are performed.

A clear advantage of the overlay approach is that only one buffer implementation is necessary which stores buffers in the JOpera memory. This allows the state storage to be ported more quickly to a different technology. Since the buffers are layered on top of the memory, they directly inherit the persistence and distribution capabilities of the memory.

5.4 Buffer allocation and capacity

As a consequence of using buffers in a process instance, the memory footprint of the instance increases[3]. The additional space consumed consists of the buffer contents and the overhead for managing the buffer. In the best case, the buffers each do not hold more than one element at a time – in which case buffers are not really needed. In the worst case, buffers are filled to their capacity most of the time. The buffer allocation strategy and buffer capacity also have an influence on the space consumed by a process instance as is discussed in the following.

5.4.1 Buffer allocation strategies

Buffers can be allocated and destroyed dynamically or statically. Dynamic allocation means that a buffer is not created before it is accessed for the first time. An advantage of this strategy is the faster instantiation of a process instance because no buffers must be allocated at that time. Also, buffers will not be created in vain for parts of a process instance which are never executed. However, the creation of buffers while a process instance is running adds an overhead to the process execution.

In the case of static allocation, all the buffers of a process instance are allocated during the instantiation. As a consequence, the instantiation takes more time. On

[3]Section 6.1 measures the memory overhead when employing flow control with markers.

the other hand, the allocation overhead during the process execution can be saved. Buffers in parts of the process instance which are never executed are still allocated although they are never used.

In the simplest case, buffers are destroyed when the corresponding process instance is deleted from the state storage. As an optimization, this can be done when the instance has terminated and thus will not use the buffers anymore. In this case, buffers which still contain data can be left intact in order for the user to inspect their content.

In order to save as much space as possible, buffers can instead be dynamically destroyed during the process execution when they are not needed any longer. Determining when this is the case is, however, not straightforward. Although dead path elimination is already performed as part of the normal process execution, this cannot be used as an indication of whether a buffer will be used in the future. The reason is that in the case of loops and pipelines such “dead paths” are only temporarily “dead”. Tasks which are marked as *Unreachable* might be executed in a new iteration of the loop or when a new data element is processed in the pipeline. Thus, an additional algorithm is needed which determines when a task will never be executed again, and is thus permanently dead, and the associated buffers can be destroyed.

In JOpera, buffers are allocated dynamically during the execution of a process. Buffers remain allocated until a process instance is explicitly removed from the state storage. The pre-allocation of buffers can easily be supported. Since the buffers needed in a process are computed anyway (Section 5.1), it would be straightforward for the setup compiler to generate additional code to allocate the buffers during the process instantiation. The same is valid for removing buffers when a process has terminated.

5.4.2 Buffer capacity

When the rate at which a task produces results (either as multiple-output task or by repeated invocation) temporarily increases, this is called a *burst*. If this output rate exceeds the rate at which a successor task can consume the results, a buffer can be used to absorb the excess results without having to block the faster task. A burst can be long or short and the increase of the output rate can be big or small. The number of elements that need to be temporarily buffered depends on both factors.

The capacity of a buffer determines the size of the bursts it can absorb. In the optimal case, the characteristics of the bursts are known in advance and the required buffer capacity can therefore be computed (see below). Primarily, buffers can be distinguished by the boundedness of their capacity. A buffer with an infinite capacity has the advantage that a task whose output is appended to the buffer is never blocked (within the overall capacity of the system). Although buffers with fixed capacities may lead to blocked tasks, they limit the space consumption of a process instance during its execution. While variable size buffers typically exhibit a bigger computational overhead due to the dynamic space management, fixed size buffers have the possibility of preallocating the needed space to achieve a more efficient space management.

The final capacity of a buffer depends on different factors. First off all, the technology used to implement the buffers may only support either finite or quasi-infinite capacity buffers. For bounded buffers, in the simplest case, all buffers have the same default capacity which is specified as part of the system configuration. In a more advanced system, the process designer tool should allow the user to override the default capacity of individual buffers when designing a process. This way, it is possible to allocate only the required buffer capacity. Ideally, the workflow system itself would be able to estimate the necessary buffer capacities by monitoring the execution behaviour of tasks.

Capacity calculation

Often, the capacity of a buffer can be calculated in a straightforward manner. When the duration of a burst is relatively short, i.e., less than the execution time of the successor task, then the whole burst must be absorbed by the buffer. However, if a burst is longer and can be partly handled by the successor task, the calculation becomes more complicated.

In general, to calculate the required capacity of the buffer between two tasks we need to know the execution characteristics of the two tasks, more specifically, their execution times. Since these times may vary over time, they are expressed as functions. Let $T_P(t)$ and $T_S(t)$ specify the execution time of the buffer's predecessor and successor task, respectively, at time t. To simplify matters, we assume continuous functions. From the execution times we can derive the *rates* at which elements are appended to and removed from the buffer. The input rate $R_i(t)$ and the output rate $R_o(t)$ of the buffer depend directly on the task execution times.

$$\begin{aligned} R_i(t) &= T_P(t)^{-1} \\ R_o(t) &= T_S(t)^{-1} \end{aligned}$$

It is important to note that elements can be appended faster than removed, $R_i > R_o$, only if the buffer is not full and elements can be removed faster than appended, $R_i < R_o$, only if the buffer is not empty. If the buffer is full or empty, then the tasks run in lockstep, i.e., $R_i = R_o$. Thus, the above equations hold under the assumption that the buffer is neither full nor empty. As we want to calculate the required buffer capacity during a burst, we assume an unbounded, and thus non-full, buffer. During the burst $R_i > R_o$ which means that the buffer is not empty either.

The difference between the input and output rate at the buffer determines how many elements are not removed and therefore remain in the buffer. If we know the length of the burst, we can calculate how many elements pile up in the buffer. This is achieved through integration over the time interval τ_B of the burst.

$$N_B = \int_{\tau_B} R_i(t) - R_o(t)\; dt$$

If we know the average input rate $\bar{R}_i^B$ and output rate $\bar{R}_o^B$ during the burst, the calculation can be simplified to

$$N_B = (\bar{R}_i^B - \bar{R}_o^B) \cdot T_B,$$

where T_B is the length of the burst.

To give an idea of the buffer capacities needed in applications, we estimate the burst volumes for some example streams. When monitoring the log file of a Web server (Chapter 7), the server writes in batches of up to 20 entries at a time. The lengths of entries vary significantly, but can be estimated to 225 characters. So, the volume of the burst is 5500 characters. Similarly, applications write log files to document their execution as well as errors occurring. Such a log stream can deliver up to 250 entries in a second, each with approximately 150 characters. This amounts to 37.5 kByte of data. Products labelled with RFID tags often arrive at an RFID scanner several at a time, e.g., 100 pieces on a palette. Thus, the scanner will detect all the tags at the same time. This generates bursts of one hundred 128-bit numbers sent to the streaming workflow.

5.5 Discussion

In this chapter we have introduced several extensions to the workflow runtime system. This was mostly possible without changing its architecture by relying on existing extension points. First, we extended the compiler to incorporate flow control in the generated navigation algorithms. For the basic flow control semantics using markers these changes are local to the compiler and the generated navigation code. Then, in order to support multiple-output together with a flow controlled navigation code, the protocol between navigation and task execution was extended. A new notification type allows subsystems to stream results into a process instance. In order to add flow control to this result stream, a new *Unblock* signal type is used by the navigator to control the output rate of subsystems. However, for the support of buffers the system architecture had to be extended with a buffer component. This component complements the existing memory model as the required functionality can not be provided with the existing state storage.

With the extensions in this chapter, we have also introduced two sources of overhead. As a consequence of using flow control in the communication with the subsystem, *Unblock* signals are sent from the navigator to the dispatcher. Together with the output notifications from the subsystems in the opposite direction, this increases the load on the navigator–dispatcher communication for a single task execution if the subsystem uses the multiple-output functionality. The use of buffers in a process also introduces an overhead. A data transfer does no longer correspond to a single copy operation but rather to an append and a remove operation on the corresponding buffer. Also, buffers add a natural overhead in time as well as in space due to the management of the buffer contents. However, if the navigator does not become the bottleneck because of the buffer overhead, then the performance of

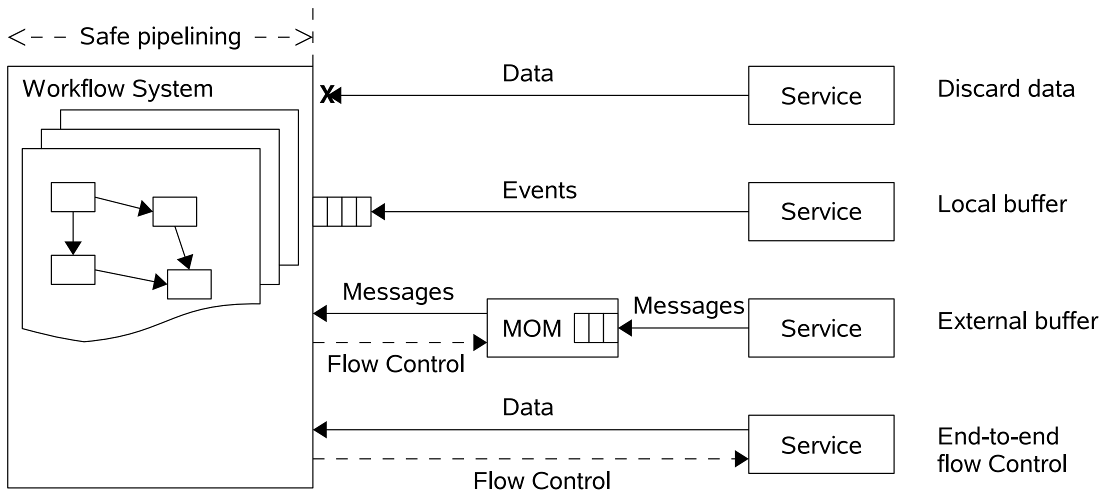


Figure 5.10: Different possibilities of controlling the flow of data between an external service and the workflow system.

the overall system is improved in situations where the varying task execution times increases the execution time of a process. Whether the overhead of using buffers in a process can be afforded is evaluated in Section 6.3.

Flow control restricts when subsystems may provide output to a running process instance. However, since the subsystem API includes blocking as well as non-blocking delivery methods, we believe that flow control does not constrain the functionality or architecture of a subsystem. This is also important when considering the communication with the external service, which is discussed in the following section.

5.5.1 Flow control beyond the workflow system

In Section 5.2.2 we extended the runtime system to integrate the task execution subsystems in the flow control chain of processes. This is necessary to prevent the subsystem from overwriting data in the process. However, the problem of data collisions has not been completely solved thereby, but has rather been shifted from the process – which is now completely safe as far as flow control and pipelining are concerned – to the task execution subsystem. The problem arises with services which are actively pushing data to the subsystem. While the service continues to send data, the subsystem might be temporarily blocked to prevent it from pushing the results into the process instance. Thus, the flow control problem remains between the subsystem and the service, because the subsystem receives data which it cannot immediately process. We briefly discuss alternatives how to approach this problem without proposing a final solution to the problem. Figure 5.10 summarizes the approaches.

Discard data

In the simplest case, data is discarded if it arrives while the subsystem is not ready to accept it and feed it to the process. As already mentioned in Section 4.3.1, in most business processes this behaviour is not viable, because usually all data elements contain essential information (e.g., notifications of items which are leaving a warehouse) and none of them can be dropped.

Local buffer

To avoid discarding data, the subsystem can employ a buffer to absorb bursts of messages which the process instance is not able to consume fast enough. The first advantage of this approach is that it does not make any assumptions about the service, messages are accepted as soon as they arrive. Second, a buffer of limited size may be implemented in a light-weight manner. The disadvantage of a limited-capacity buffer is that it might overflow if a big burst of messages is received. In case of an overflow, some of the messages are inevitably lost. Thus, there is a trade-off between the buffer size (and the incurred resource consumption) and the size of bursts which can be tolerated[4].

An example of an application where a receive buffer can be employed is an RFID reader in a warehouse. Typically, many items, each with an RFID tag, will pass the reader at the same time. This generates a burst of notifications which are sent to a workflow for inventory management. Another example is a sensor network where the nodes inform about events which are observed (e.g., sudden changes in temperature). In both cases the arrival time of messages cannot be foreseen and there is no possibility to prevent a sender from transmitting.

External buffer

A more expensive variant of the local buffer is to use a queueing system [3] outside the workflow system. The service sends its messages to this external buffer and the subsystem fetches a message whenever the process instance is ready for new input. Such a queueing system normally has quasi-unlimited capacity, so no data should ever have to be discarded. The disadvantage of the approach is its complexity and that either the service must be able to communicate directly with the queue or there must be an adapter which receives messages from the service and places them on the queue.

End-to-end flow control

In the optimal case, the flow control chain can be extended as far as to the streaming service. The flow of information is controlled explicitly through special acknowledgement or control messages or implicitly through the use of, e.g., TCP [107] to connect to the service. This approach allows the subsystem to control the rate at which the

[4]See Section 5.4.2 on how to calculate the required capacity for a buffer.

service delivers results. There is no need for a buffer between the service and the workflow system since data is sent only if it can be processed immediately. However, for this to work, the service is required to support flow control. As this increases the complexity of the service and its interface, service providers may not be willing to provide flow control.

6 Measurements

In this chapter we evaluate the performance of the extensions we developed in chapters 4 and 5. First we measure the overhead that flow control adds to the navigation of processes instances. Then, we compare two approaches of providing a Web service with a workflow system. One approach creates a new process instance for each invocation of the service. In the second case, the invocations are streamed through a pipelined process. Following are two measurements of using buffers in a process. We show the influence of buffers on varying task execution times and we compare the performance of two different implementations of the buffering infrastructure.

6.1 Navigation overhead of flow control

When a process is compiled with flow control, the navigation procedure of the process becomes more complex. In addition to checking the states of tasks, flow control markers must be tested and manipulated as part of the starting conditions of a task (Section 4.4). The goal of this first experiment is to quantify the overhead on the performance and on the memory consumption, which is introduced when flow control is used in the execution of a process. To measure this, processes with different characteristics are run both with and without flow control. The overhead can then be derived from the differences in process execution time.

6.1.1 Setup

In this experiment processes are run with and without flow control and their execution times are compared. The processes consist of tasks which just "sleep", i.e., they do nothing for a specified amount of time. Since flow control only affects navigation, tasks do not need to carry out any "real work" as part of this experiment. It is even beneficial for the task execution not to induce any load on the system, so the performance of the navigation can be observed better.

To increase the accuracy of the measurements, the experiment uses processes with a sequence of tasks as long as possible[1]. For the measurement of the task throughput (see below), the task sequence was additionally enclosed in a loop to increase the execution time of the process. This minimizes the relative overhead of process instantiation and termination.

[1]The navigation code of a process is contained in a single Java method (Section 3.3.2). Due to the 64 KiB size limit for the bytecode of a Java method, JOpera processes can currently not be of arbitrary size.

Parameter	Values
Control flow complexity	1, 2
Flow control	enabled, disabled
Task type	synchronous, asynchronous
Task duration [ms]	0, 1000
Number of process instances	1, 10, 100, 200, 400, 600, 800, 1000

Table 6.1: Parameter values used in the overhead experiment.

To account for the fact that processes normally have a data flow between tasks, every task has 2 input and 2 output parameters which are used to pass an object from task to task. Thus, for every task execution two data transfers are carried out by the navigator. Since objects are passed by reference in Java, and thus in JOpera, we chose the arbitrary size of 1024 Bytes for the object.

Parameters

The parameters described in the following were varied during the experiment. Table 6.1 summarizes the values which were used for these parameters.

Control flow complexity Processes with different *control flow complexities* are used in the experiment. The control flow complexity of a process is defined as the average number of incoming control flow dependencies per task. Since these dependencies are checked in the activator of a task, the control flow complexity of a process is a measure of the work done in the navigation of the process. The control flow dependencies are also the point in the system which is extended by flow control. The complexity in the experiments is 1 or 2. Considering that most real processes consist not only of control flow merges as in our experiments, but also contain splits and sequences, we believe that the average number of control flow dependencies in such processes will not be much higher than 2.

The control flow complexity of processes in the experiment is controlled as follows. The process consists of several stages (Figure 6.1). For a complexity of c, each stage contains c tasks. Every task in a stage then has a dependency on each of the c tasks in the predecessor stage. This results in c control flow dependencies per task. Obviously, the tasks in the first stage cannot comply with this rule as there is no predecessor stage. Therefore, the effective average complexity of the process is slightly less than desired. For the processes used in this experiment the effective complexities were 1 and 1.93.

Flow control This is the most important parameter. Processes were compiled with and without flow control before they were used in the experiment. We expect processes which are compiled with flow control to have a higher execution time.

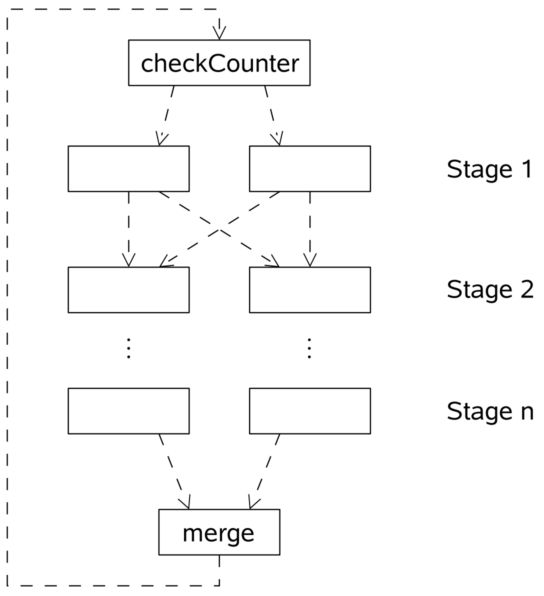


Figure 6.1: Process used to measure the navigation overhead of flow control. This example has a control flow complexity of 2.

Task type Two types of task implementation were used. The first type was a Java snippet which is executed in the same thread as the navigation. Thus, navigation and task execution are strictly serialized. In the second case, the task was executed asynchronously, thus allowing multiple tasks to run simultaneously as well as allowing navigation to continue in parallel to the task executions.

Task duration This parameter specifies the duration of each task in the process. Since we want to measure the navigation overhead, the duration of the tasks in the process does not matter. We set the duration to 1000 ms to allow enough time for the navigator to work in parallel in the case where tasks are executed asynchronously. By also using a delay of 0 ms, the theoretical task throughput of the system can be measured. By spending the minimal time on the actual task execution, the performance of navigation and scheduling of task executions can be measured.

Number of parallel process instances By running more than one process instance at the same time, the influence of flow control on the scalability in terms of process instances can be observed. If flow control increases the contention for resources, then the scalability will suffer. However, since navigation is single-threaded, we expect no influence on the scalability.

Measured variables

The execution time of a process instance or, in the case of multiple process instances, the total time of the batch of process instances was measured. We also measured

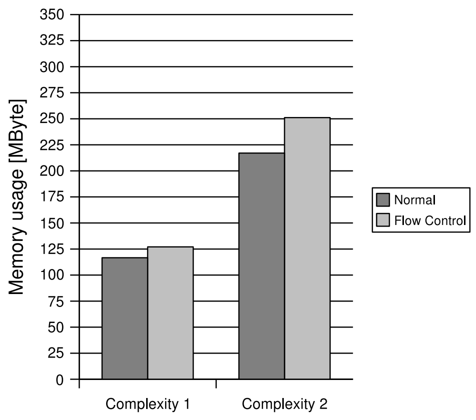


Figure 6.2: Memory consumption of 1000 process instances depending on their complexity and use of flow control.

the memory consumption of processes depending on their complexity and use of flow control.

As the execution time of a process instance depends not only on the performance of the navigator but also on the size of the process, we have computed the task throughput from the measured execution times. To do so, the task duration was set to 0 ms. The different throughputs can then be used to compare the navigation performance of different settings.

To achieve a high accuracy, when the loop-version of the processes were used, the loop executed 1000 iterations. All parameter combinations were measured 10 times and their results averaged.

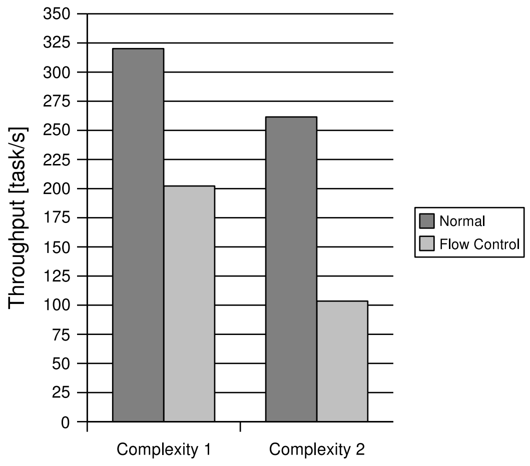
Hardware and software setup

The experiment was run on machines equipped with two dual-core AMD Opteron processors running 64-bit Linux 2.6.14 at 2.2 GHz and with 4 GiB of main memory. JOpera was executed in a Blackdown-1.4.2_03-AMD64 server JVM.

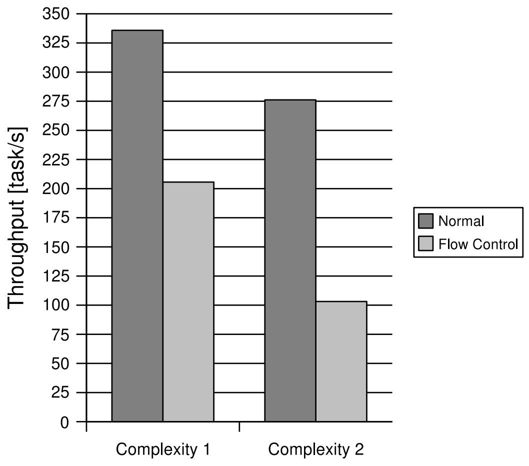
6.1.2 Results

Memory consumption

First, we look at the influence of flow control on the memory size of process instances. Figure 6.2 shows the memory consumption of 1000 process instances with different control flow complexities. The flow control markers seem to add only a small overhead to the size of a process instance, 9% and 16%, depending on the complexity. It seems reasonable that the overhead for a complexity of 2 is almost twice as big compared to a complexity of 1, since the number of markers grows linearly with the complexity, i.e., with the number of task dependencies.



(a) Synchronous task execution



(b) Asynchronous task execution

Figure 6.3: Maximum task throughput of the navigator with and without flow control, for different control flow complexities and task execution styles.

Throughput degradation with flow control

Figure 6.3 compares the task throughputs when running a process with and without flow control. The task duration was set to 0 ms to measure the maximum theoretical throughput of the system. Clearly, flow control adds an overhead which reduces the performance of navigation. In the case of synchronous task execution (Figure 6.3a), for a process with complexity 1 flow control reduces the throughput to 63% compared to not using flow control. When the complexity of the process is increased to 2, the throughput shrinks to 40%. This lower relative performance can be explained with the fact that the complexity has an influence on how many flow control markers[2] must be loaded from memory when employing flow control, whereas without flow control the number of memory accesses is constant (only the task states are loaded).

When looking at the results for asynchronous task execution (Figure 6.3b), the task throughput is higher compared with synchronous task execution. This is because part of the navigation can take place while some tasks are executing, so navigation contributes less to the total process execution time. However, using flow control, the benefit of asynchronous tasks is only marginal. The relative performance is even lower in this case (61% and 37%, respectively), because of the speedup of the non-flow control execution.

Scalability of flow control

Figure 6.4 illustrates the scalability of the system with and without flow control for two control flow complexities. The execution times of process instance batches are plotted against the batch size. The corresponding processes were executing tasks

[2]See Section 4.4.2 for details on the flow control mechanism.

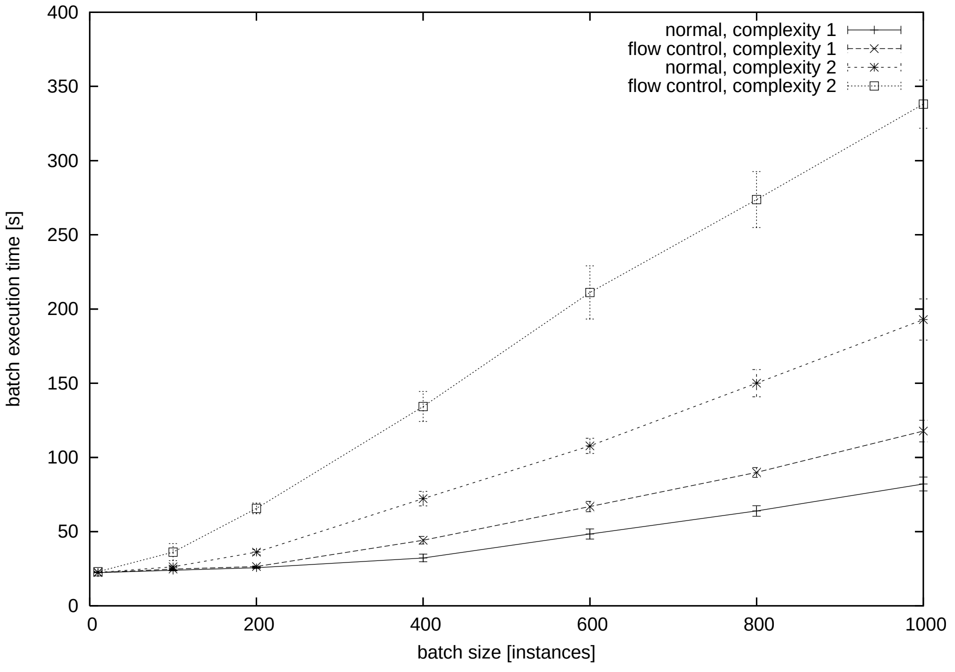


Figure 6.4: Influence of flow control on the scalability of navigation.

asynchronously to allow for the maximum parallelism between individual process instances. The duration of the tasks was set to 1000 ms.

The overhead of flow control can also be seen here, as the batches take longer to execute with flow control than without. Also, for a higher complexity, the overhead is bigger. However, most important, the results also show that the system still has linear scalability. This is due to the fact that although flow control adds an overhead, it does not introduce any new contention for resources. All the computations related to flow control are done inside the single-threaded navigator.

It can be noticed that for small batches the execution time is the same regardless of the complexity or use of flow control. This is because the time spent on navigation is still very low compared to the duration of an individual task: all active process instances can be navigated within the duration of a single task, and thus, the effective task durations are the same for all experiment parameters. As the number of process instances increases, the time spent on navigation starts to dominate the overall time it takes to execute a process. This occurs both with and without flow control, although flow control makes it happen earlier.

6.1.3 Discussion

The results show that flow control adds a considerable overhead to the navigation of process instances. This rises the question whether the performance is still acceptable for applications of streaming processes. JOpera with flow control is not meant to be used to process high-speed real-time streams. In order to process high-volume streams, a data stream management system can be used to filter a stream before it is processed with the workflow system. The performance of flow controlled processes is suited better for streams such as RSS or Atom feeds. Such feeds deliver new items with a relatively low frequency and the contained information does not have be processed within milliseconds. For example, the *New York Times* publishes on their front page approx. 300 stories per week. The U. S. Geological Survey [115] publishes feeds of recent earthquakes which occur with an average frequency of approximately 4 per hour. Amazon's offers RSS feeds for tracking products as they get tagged with a certain keyword. The feed for “dvd” delivers approx. 400 items per week

The number of streams which can be processed concurrently depends very much on their rate of new elements and on the number of process instances which have to be navigated as new data arrives. Another application with an appropriate data rate is described in Chapter 7.

Also, it is important to note that the flow control implementation which was measured has not been optimized for performance. Future work should try to reduce the overhead of flow control through optimization or through new approaches such as scheduling tasks in batches or by restructuring the navigation code. Also without optimizing the navigation, the performance of the system can be scaled by employing multiple navigators[3], as has been shown in [94]. This increases the overall task throughput within the scalability of the state storage and the event queues.

6.2 Performance improvement through shared pipelines

Workflow management is often closely related to Service Oriented Architectures and Web services [77]. WS-BPEL [89], e.g., is directly based on Web services: the activities in WS-BPEL processes are Web service invocations and also the process itself is published as a Web service so it can be interacted with. In general, a process which implements a Web service typically receives a SOAP [122] request, executes some processing steps on the contained information, also invoking other services, and finally sends a reply to the invoking client.

For any system which provides a service to multiple concurrent clients the scalability in terms of parallel invocations is an important characteristic. Typically, as the number of parallel service invocations increases, the workflow system eventually reaches a point where it is saturated (Figure 6.5). As the the number of clients is increased further, the throughput stays on the same level or even decreases because the contention for shared resources grows. Also, the response time typically increases together with the number of clients sending requests to the system.

In this experiment we try to improve the performance of JOpera when it is used to provide a Web service. Part of the load which a service invocation creates is the instantiation of a process to handle the invocation. The basic idea of the experiment

[3]Also with the flow control extensions introduced in Chapter 5, the system can still be run in a distributed fashion, as the extensions rely on existing mechanisms of the system.

is to save the instantiation overhead by using pre-created process instances through which the service invocations are pipelined. Doing so removes the instantiation overhead, but instead adds the overhead of flow control which is necessary for the pipelining. In the following, we compare the two approaches to find out whether the pipelined approach can deliver a higher performance.

6.2.1 Setup

This experiment was run as a distributed system and consisted of the following components.

Server The central server hosts a Web service whose function is just a simple echo with a configurable delay. The service is implemented with a process consisting of two types of tasks. The first type is responsible for the communication with clients invoking the service, the other type invokes other external services. The first task in the process receives the service request from the client. Following are a configurable number of tasks which sequentially invoke an external sleep service to simulate the invocation of "real services". The duration of these service invocations was set to 1 second. Finally, the last task sends a reply back to the client which invoked the service.

Sleep service The sleep service is a simple Web service which does nothing for a definable amount of time and then returns. It is used by the service implementation instead of invoking "real services". From the point of view of the workflow engine, it does not matter whether an invoked service computes a result or just sleeps, as long as the service scales equally good.

Client A client continuously invokes the Web service on the server. When one invocations returns, the next is immediately started.

Parameters

The following parameters were used to control the experiment.

Instance creation The process implementing the Web service was deployed in two different configurations. In the first case, every incoming service request *dynamically* creates a process instance and the request is handled by this instance. The process is compiled without flow control, as this is not necessary for this type of execution. This is how Web services are typically implemented if the implementation is process-based, e.g., when using WS-BPEL. After the process instance has replied back to the client, it is deleted. Otherwise, the main memory would fill up within minutes and the system would not be able to handle further requests.

The second deployment consists of pre-starting a certain amount of process instances and using these in a pipelined fashion to serve all the clients. These *static* instances are sharing the load of the clients, i.e., the incoming requests are distributed in a round robin fashion over all instances. A request enters a process instance through the receive task and then flows through the pipeline in the process together with other requests. Thus, a process instance is able to handle multiple requests in parallel. Since the process is used with pipelining, flow control is enabled for its compilation.

The throughput of a pipeline depends on the slowest stage in the pipeline. As the process in our experiment contains service invocations which all last 1 second, the throughput of a single process instance is 1 service invocation per second. In order to achieve a higher overall throughput of the system, multiple pipelines must be run in parallel as just described. The number of instances was set to 25 for our experiment as is discussed below.

Length of the pipeline The amount of parallelism which can be achieved in a pipeline is limited by the number of stages of the pipeline. Therefore, we used processes with different numbers of external service invocations (4 and 8). We expect the throughput and response time to change proportionally to the length of the process.

Number of clients The load on the workflow system depends on the number of clients invoking the Web service concurrently. This number was varied from 10 to 150 in steps of 10.

Measured variables

Each client measured the throughput and response times of the invocations to the Web service. From these values the total throughput and the average response time was calculated. Each parameter combination was run for 5 minutes.

Hardware and software setup

All the components of the experiment were running on identical machines. These machines were equipped with two AMD Opteron processors running Linux 2.6.9 at 2.4 GHz and with 2 GiB of main memory. The machines were connected by a 1 Gbps-Ethernet. All components were executed in Sun's Java HotSpot Server VM 1.4.2_10. JOpera, providing the Web service, and the sleep service were running on separate machines. To ensure that the experimental setup does not introduce an artificial bottleneck, two machines were used to run the clients, one half each. Also, the thread pools used in the different components were configured with appropriate capacities. JOpera had 256 threads for the embedded HTTP daemon and 300 threads for the dispatcher, the sleep service held 250 threads ready.

6.2.2 Results

In order to determine the number of parallel process instances to pre-allocate for the static case, we first needed to measure the throughput of the system with dynamic instance allocation. This way, we were able to start only as many process instances

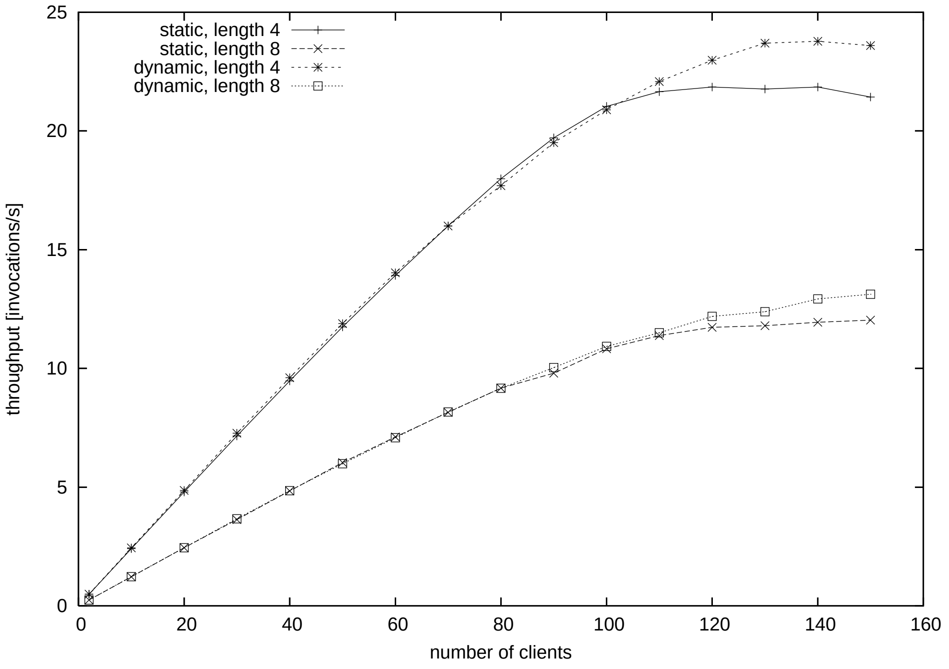


Figure 6.5: Throughput with dynamically and statically allocated processes for two pipeline lengths.

as required to achieve the same throughput with static instances and avoid over-provisioning.

Figure 6.5 shows the throughput with dynamic instance allocation. As expected, the system scales linearly up to a certain load, then the throughput flattens and finally starts to fall as the contention between the process instances becomes too big. The maximum throughput is 23.8 invocations/s which is achieved with 140 parallel clients. The response time also behaves as expected (Figure 6.6). It grows slowly in the beginning and then starts to increase exponentially at the same point at which the throughput begins to flatten.

As we wanted to outperform the above results using shared process instances, we set the number of pre-allocated instances to 25. This would allow the system to scale to a maximum of 25 invocations/s. Figures 6.5 and 6.6 also show the results with statically allocated instances. As can be seen, both the throughput and the response time are very similar to the case with dynamic allocation when the system is not saturated. However, we were not able to achieve a higher throughput because the throughput flattens earlier. The explanation is that the overhead introduced by flow control slows down the static process instances too much and the navigator thread saturates its CPU before a high enough throughput is reached. This means that the overhead from flow control is bigger than the overhead of dynamically creating process instances which is saved when using shared pipelines.

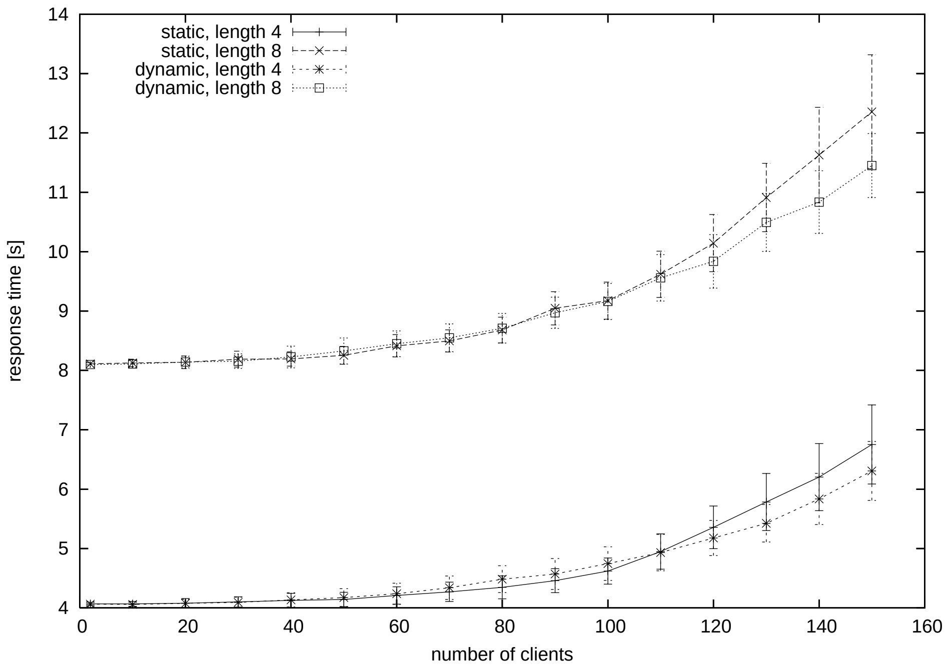


Figure 6.6: Response time with dynamically and statically allocated processes for two pipeline lengths.

6.2.3 Discussion

This experiment shows that the overhead of flow control is still too big to be able to compete with the performance of "normal" processes. However, the difference between the two approaches has shown not to be as big as could be expected after the measurement of the pure navigation overhead in Section 6.1. The maximum throughputs of the approaches lie only 8% apart. As mentioned before, the flow control implementation is not optimized. It is thus likely that an optimized version can be used to beat the performance of dynamic process instantiation.

Statically allocated instances, however, do have a clear advantage. At the highest throughput of the dynamic approach 140 concurrent process instances are active, one for each client. In contrast, the static approach has a constant number of 25 instances – more than five times less, even when taking into account the memory overhead introduced by flow control. Thus, the static approach sacrifices a small amount of performance for a much better memory efficiency.

6.3 Absorbing bursts with buffers

After evaluating pure flow control based on the marker mechanism, we now turn to buffers which are an extension of the markers (Section 4.5.1). As mentioned in Section 4.8, the performance of a pipeline suffers if the execution durations of the tasks in the pipeline vary. The bigger the variation, the higher the chances that a task is blocked waiting for the successor to accept new data. By using buffers in a process, tasks do not have to be blocked immediately when the successor task is not able to accept new data (Section 5.4.2). Instead, the data is appended to a buffer and removed from there later on. To illustrate the benefit of using buffers, we have conducted the following experiment which measures the ability of buffers to decouple the execution of tasks.

6.3.1 Setup

We use a simple process with a linear pipeline of eight tasks. Each of these tasks sleeps for a certain amount of time. At the head of the pipeline is an additional task which continuously triggers the pipeline, simulating an incoming data stream. The duration of the sleeping tasks is randomized around a nominal duration of 1 second. Buffers of different size are used between the pipeline steps to absorb the emerging bursts.

Parameters

The following parameters were used to control the experiment.

Task duration jitter The randomness of the duration of the pipeline steps is controlled by the jitter parameter. It specifies the maximum relative deviation (positive and negative) from a nominal duration. For instance, a jitter of 50% allows durations between 50% and 150% of the nominal duration. Formally, the duration of each step in the pipeline is $nominal * (1 - jitter + random(2 * jitter))$, where the function $random(u)$ returns a uniformly distributed random value between 0 and u.

This approach creates rather very short bursts and can thus be used only as an approximation of a real system. However, this is sufficient in order to slow down the pipeline when no buffers are used or to fill the buffers between tasks with a certain number of elements over time.

Buffer size The pipeline process is run with different configurations regarding the buffering. First of all, we run the process with pure flow control based on markers, i.e., without any additional buffers. This corresponds to an implicit buffer capacity of one element (Section 4.4.4). When the process is run with buffers in place, the capacity of these is in turn set to 2,4,8,16 and 32. We expect that a greater buffer capacity will lead to a smaller influence of the task duration jitter and thus to a higher throughput for the pipeline.

Measured variables

The first step of the pipeline was triggered 1000 times to simulate the corresponding number of stream elements flowing through the pipeline. We measured the time for all the elements to pass through the pipeline.

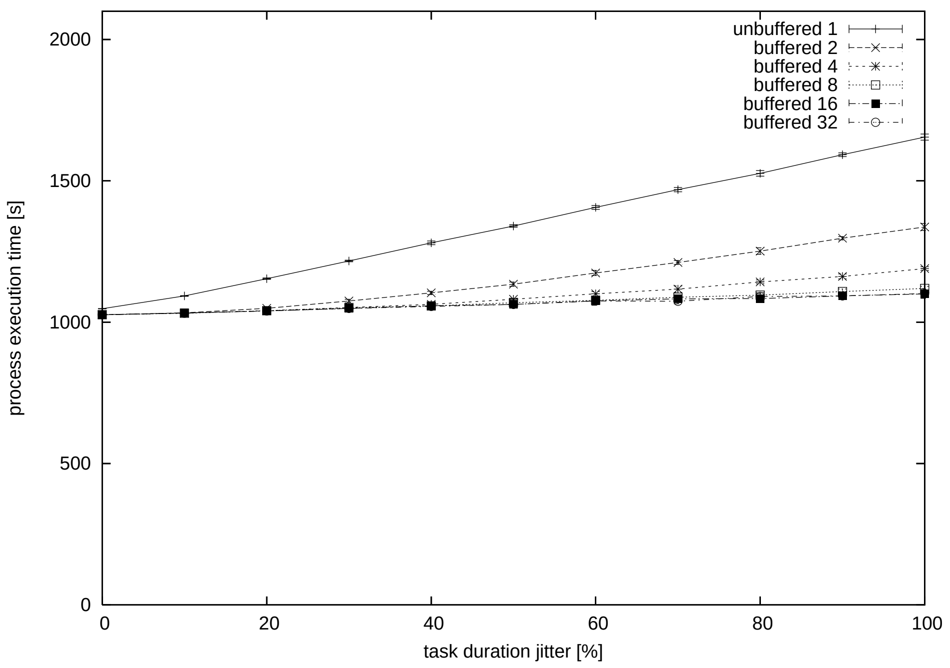


Figure 6.7: Execution time of a process depending on the irregularity of task execution durations, for different buffer sizes.

Hardware and software setup

The experiment was run on machines equipped with two dual-core AMD Opteron processors running 64-bit Linux 2.6.14 at 2.2 GHz and with 4 GiB of main memory. JOpera was executed in a Blackdown-1.4.2_03-AMD64 server JVM.

6.3.2 Results

The uppermost curve in Figure 6.7 ("unbuffered") shows the effect of an increasing variation of the task durations on the total execution time of the process. As expected, the total time to process all stream elements increases with the variation. The other curves in the figure show the positive effect of decoupling tasks by inserting buffers in between them. The overall execution time drops as the buffer size increases.

From the results, the characteristics of the bursts can also be inferred. A buffer capacity bigger than eight does not improve the performance much. The reason is that very few bursts have a size bigger than eight. On the other hand, a very small buffer of size two already reduces the negative effect of the task duration jitter to 50%, the reason being that most bursts in our experiment are very short and can be absorbed by such a small buffer.

6.3.3 Discussion

The results show clearly that buffers help to smooth out the flow of information in a process by absorbing bursts which occur between tasks with irregular durations. This reduces the time tasks are blocked and are not processing any data. Thus, the performance of the system is improved in terms of throughput and response time. Even though buffers add an overhead to the navigation of process instances, buffers already help to reduce the effects of small irregularities of 10% in the task durations and lead to an overall lower process time.

6.4 Performance of buffer implementations

In Section 5.3.1 we specified an abstract interface for the buffer infrastructure. To validate the design of the interface, we have created different implementations. In this experiment we compare the performances of a memory-based implementation and a simple persistent implementation.

6.4.1 Setup

The pipeline process from the previous experiment was used with an additional data flow between the tasks. This was done to increase the load on the buffering infrastructure when the buffer contents have to be persisted (depending on the buffer implementation). Objects of 1024 bytes were passed in the data flow of the pipeline, corresponding to, e.g., a small XML document. The capacity of the buffers between the pipeline tasks was set to eight elements, resulting in a total size of a buffer of 8 KiB. The nominal duration of the pipeline steps was set to 1 second and the jitter was 100%, resulting in a random duration between 0 and 2 seconds, to make sure the buffers become as full as possible.

Parameters

The following parameters were used to control the experiment.

Number of parallel instances During the execution of a process instance the contained buffers are accessed a certain number of times. By running a batch of multiple concurrent process instances, this load on the buffering infrastructure is increased and performance differences of the implementations become more apparent. The sizes of the batches were powers of 2 up to 256.

Buffer implementation We compare two different implementations of a buffering infrastructure as specified in Section 5.3.1. The first one is based on in-memory circular buffers which should allow a good performance. The second implementation extends the in-memory buffers with persistence. Every time an element is appended to or removed from a buffer, the buffer is saved to disk. This is a naive approach

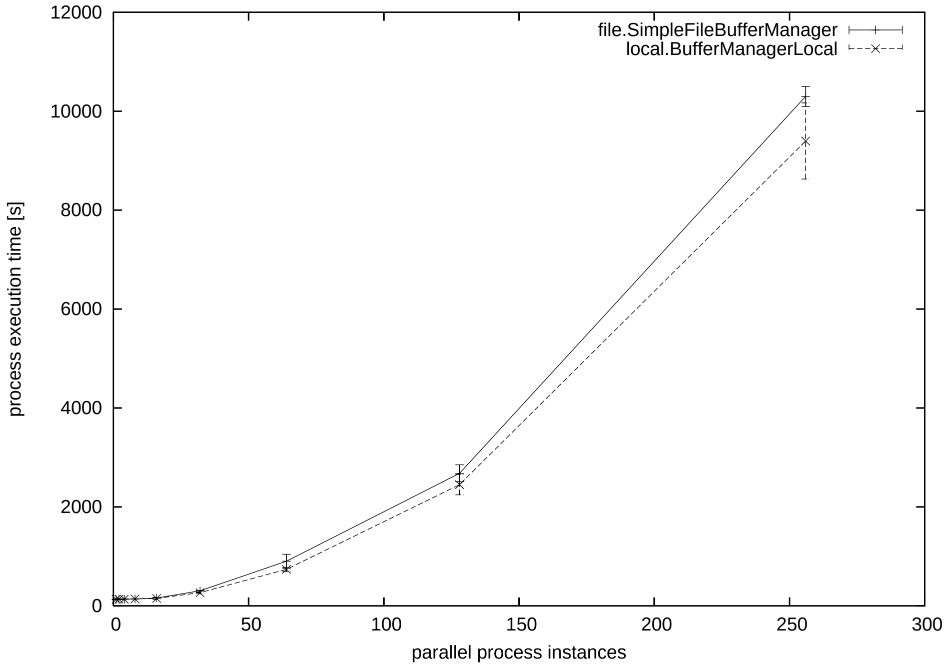


Figure 6.8: Comparison of the performance of different buffer implementations.

and is intended only to validate the specified interface for buffers and to illustrate the performance differences arising from different implementations.

Measured variables

We measured the total execution time of a batch of process instances.

Hardware and software setup

The experiment was run on machines equipped with two dual-core AMD Opteron processors running 64-bit Linux 2.6.14 at 2.2 GHz and with 4 GiB of main memory. JOpera was executed in a Blackdown-1.4.2_03-AMD64 server JVM.

6.4.2 Results

Apart from showing the performance difference between the two implementations, Figure 6.8 shows two other interesting aspects: First, for a low number of parallel process instances, there is almost no difference between the two measured implementations. Second, the execution time grows exponentially with the number of process instances.

As described in Section 5.3.1, the buffer infrastructure is accessed exclusively by the navigator. Thus, an overhead in the implementation of the buffer infrastructure should affect only the navigator's performance. As we have already seen earlier (Section 6.1), the characteristics of the executed processes determine whether the navigator becomes the bottleneck of the system. The results in Figure 6.8 confirm this observation. Although the two buffer implementations have different performances, for less than approximately 25 process instances, this has no effect on the execution time of the process. If we calculate the task throughput for 25 process instances (with 8 tasks each of 1 second duration), we get approximately 200 tasks/second which is in line with what was measured earlier for processes with flow control.

When the number of process instances is increased further, the number of tasks to navigate and thus the load on the navigator increases, so that the navigator becomes the dominant factor of the execution time. When this is the case, the different overheads of the two buffer implementations can be observed: the persistence of the buffer contents cost a longer execution time. However, the difference is not very big, even when using a naive approach to persistence. This is most probably due to write behind caching in the operating system which speeds up file system write operations.

Last but not least, it can be noticed that the process execution time grows exponentially with the number of process instances. One would expect a linear growth, as the load on the single-threaded navigator increases linearly. However, other components of the system also get more loaded by running more process instances. To make the pipeline steps sleep for a certain time, a shared timer facility was used. As more tasks are executed in parallel, the contention for the timer grows which explains an exponential rise of the process execution times.

7 Application: Streaming Mashups

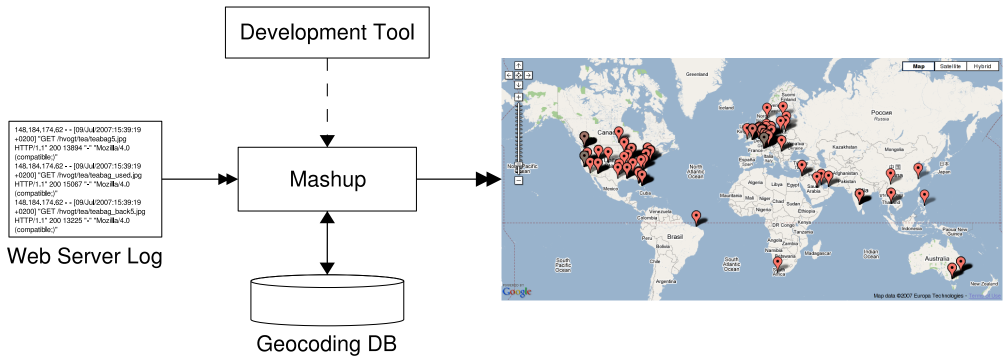
This chapter describes an application of the work developed in this thesis. A *mashup* is a Web application which is made out of existing services and data sources. With the pipelining capabilities introduced in JOpera it becomes possible to easily create mashups where data streams are integrated with other data sources. By relying on flow control, the pipelining effect allows such integrations to run at the maximum capacity. We illustrate this with a real-time web site monitor mashup. This application was first described in [14].

After an introduction, in Section 7.2 we describe an example mashup and define the challenges implied by the example. In Section 7.3 we present how to address these challenges with a process-based implementation of an example mashup. In Section 7.4 we discuss the architecture of the mashup application. In Section 7.5 we evaluate the performance of our approach regarding the ability to process data in real time. In Section 7.6 we discuss the impact of several ideas for extending the mashup example.

7.1 Introduction

Software reuse and composition has reached the next evolutionary level with mashups, where software components delivered as a service on the Web and databases published as Web data feeds are combined in novel and unforeseen ways. Whereas the term mashup originates from the practice of sampling existing music and mixing it together, software mashups are typically associated with software integration performed at the User-Interface layer [32]. In modern Web applications, this translates to (mis)using the Web browser as an integration platform for running interactive Web applications made out of different existing Web sites, applications, services, and data sources [123]. Given the limitations of such "pure" *client-based* mashups, which make it difficult if not impossible for the browser to safely interact with a variety of distinct service providers, complex mashups are designed using a *server-based* architecture. This alternative design separates the integration logic from the presentation of the mashup to better fit the constraints of current Web application architectures.

Whereas the "pipe and filters" architectural style [1] goes back to early UNIX command line pipelines, it has seen a renaissance in recent mashup tooling (e.g., Yahoo! Pipes [128], Microsoft Popfly [83], IBM DAMIA [4]). In the following we show how to apply the process-based paradigm to the integration logic of server-based mashups. We illustrate this with a case study showing how we have built a concrete mashup application: a monitoring tool for plotting the location of Web site



```
148.184.174.62 - - [09/Jul/2007:15:39:19
+0200] "GET /hvogt/tea/teabag5.jpg
HTTP/1.1" 200 13894 "-" "Mozilla/4.0
(compatible;)"
148.184.174.62 - - [09/Jul/2007:15:39:19
+0200] "GET /hvogt/tea/teabag_used.jpg
HTTP/1.1" 200 15067 "-" "Mozilla/4.0
(compatible;)"
148.184.174.62 - - [09/Jul/2007:15:39:19
+0200] "GET /hvogt/tea/teabag_back5.jpg
HTTP/1.1" 200 13225 "-" "Mozilla/4.0
(compatible;)"
```

Figure 7.1: Overview of the mashup example application.

visitors on a map widget in real time. This mashup is built as a pipeline linking the access logs of a Web site with a geolocation service for IP addresses. The results are streamed to be displayed on a map widget running on the Web browser.

7.2 Mashup Example Challenge

We focus on a simple mashup application, which is used to track visitors of a Web site using a map widget (Figure 7.1). The mashup extracts the information from the access log of a Web server and integrates it with a geolocation service that attempts to provide map coordinates for each IP address found in the log. As opposed to similar mashups (e.g., GVisit [41]), which provide a daily update of the visitors map, we are interested in providing a real-time monitoring view.

In more detail, the mashup user-interface contains the map widget that displays the Web site visitors using markers located at the visitor's geographical coordinates. Each marker is also associated with the original IP address, so that the user may find out more information about the Web log entry by clicking on the corresponding map marker. Markers appear on the map as soon as the Web site has logged the corresponding "hit" and the geolocation service successfully estimated the geographical location of the visitor. Old markers are automatically removed. The user can configure how many markers should be kept on the map at the same time.

This example mashup highlights some of the most common integration challenges involved in building a mashup. These can be summarized as follows.

Data Extraction Information relevant for the mashup is polled or pushed from a set of data sources. In our case, the Web site log must be monitored for changes and each new entry must be parsed to extract the IP address of the visitor.

Data Heterogeneity Different data sources may present data in different formats, which require more or less effort to "scrape out" [12] the required information for providing it to the mashup. In our example, different Web servers may use different formats to store their logs, even though these are typically stored in a plain-text format (i.e., not in XML). Thus, each entry is separated by a line feed character and the IP address of the visitor is contained at some predefined position within a log entry.

Data Integration By definition, in a mashup, data from more than one source must be combined in some way. The example requires to join the data extracted from the logs with geolocation information stored in a database. A more complex form of integration would, for instance, attempt to classify visitors based on their activities while on the website (e.g., to distinguish successful file downloads from incorrect user authentication attempts) – thus requiring to analyze multiple log files.

Service Heterogeneity Access to the data source may take place under different styles: a subscription to an RSS feed [13], a traditional Web service invocation using SOAP [122], the retrieval of the resource state of a RESTful Web service [99], or by piping the result of a UNIX command line. Regarding the example mashup, the geolocation database can usually be accessed remotely through a Web service interface [58, 80, 44, 113] which may however put a daily cap to limit the number of messages exchanged. As a more efficient alternative, a JDBC interface may be available to directly query a locally installed copy of the database [66, 98, 65].

Data Quality Due to limited coverage of the geolocation database, not all IP addresses may be successfully located and therefore it may be impossible to plot markers with the accurate position of all log entries. In general, the quality of the data from different sources may vary and not all service providers may be trusted to provide correct results.

Real-time Update In a monitoring application, new data must be displayed on the mashup user-interface widgets as soon as it is generated. Thus, a streaming communication channel must be opened linking the various components of the mashup so that data can flow from the Web site visitor log on the Web server to the map widgets on the Web browser.

Maintainability The long-term maintainability of a mashup is directly correlated with the amount of change affecting the APIs of the composed Web services. No-longer maintained mashups are very likely to break as the underlying APIs evolve independently and the data sources integrated by the mashups change their representation format. Having a clear model of the integration logic of the mashup helps to control the effect of changes. For example, when running the mashup example with a different Web server, only the data extraction part of the pipeline needs to be updated in case the access log format has changed.

Security From a security standpoint, the mashup needs to be configured to gain access to the Web server logs. This requires to entrust the mashup with credentials that allow it to access the Web server via a secure shell connection.

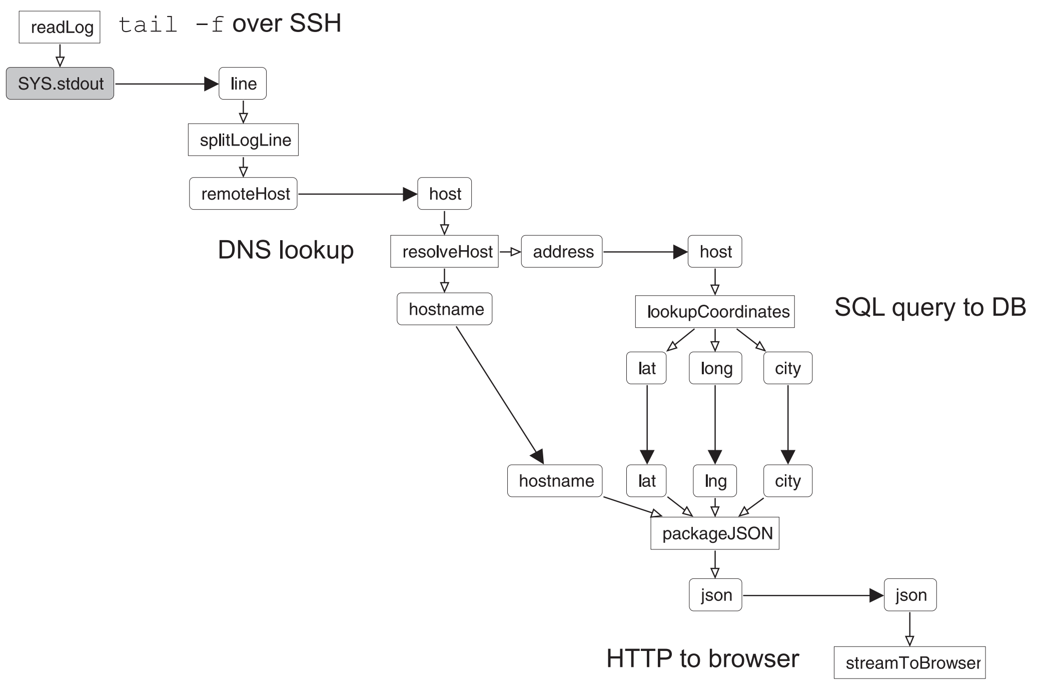


Figure 7.2: The integration logic data flow pipeline for the mashup example.

In general, it remains a challenge for users to safely delegate a mashup to access remote data sources and services on their behalf.

7.3 Mashup Integration Logic

The mashup application is designed using a layered architecture, separating the presentation from the integration logic [43]. The former is implemented using standard AJAX techniques [8], used to create a map widget and to asynchronously fetch the data stream to be displayed on it. In this section we focus on how such a data stream is computed using a workflow process.

The integration logic can be developed bottom-up or top-down. When working *bottom-up*, the mashup components (i.e., Web services and data sources, and glue code) with their interfaces and access mechanisms are defined before they are wired together in a pipeline. In a *top-down* approach, first the overall structure of the mashup is specified and then the details of each mashup task is worked out. Often, a mixed *iterative* approach is needed, because some of the components have a predefined interface that cannot be changed. Thus, *glue tasks* have to be introduced in order to make the existing components work together. In the following, we take a top-down approach to describe the mashup integration logic.

Our example mashup combines the log file of a Web server with a geolocation service. These have been integrated using the process shown in Figure 7.2.

Table 7.1: Component implementation details.

readLog (SSH)	`tail -f logs/access_log`
splitLogLine (Regular expression)	`(\S+)\s+(\S+)\s+(\S+)\s+\[(.+)\]\s+"(.*)"\s+(\S+)\s(\S+)\s+"(.*)"...`
resolveHost (Java snippet)	`...address = inetAddress.getHostAddress();...`
lookupCoordinates (JDBC)	`select lat, long, city from coords where subnet = network('%host%/24')`
packageJSON (String concatenation expression)	`{"lat": %lat%, "lng": %lng%, "city": %city%, "hostname": %hostname%}`
streamToBrowser (HTTP data port)	`<script>acceptData(%json%)</script>`

The process consists of three main components: readLog which accesses the Web server log, lookupCoordinates which looks up the geographical coordinates for a given IP address, and streamToBrowser which sends the results to the browser for visualization. In order to correctly integrate these three components other glue components have been included to deal with data heterogeneity and quality issues. The implementation of all the components is summarized in Table 7.1. We describe it in more detail in the remainder of this section.

The readLog task retrieves the log from the Web server, one entry after the other. Since a Web server does not make its log file publicly available, the log file is accessed through an SSH connection to the server. There, to extract the data, the `tail -f` UNIX command is run to fetch new log entries. With it, the readLog component outputs the entries to be processed by the rest of the mashup as soon as they are appended by the Web server, using the multiple output feature introduced in Section 4.2.3. If the information for the mashup were provided differently, readLog could use a different data extraction mechanism to access the data stream (e.g., RSS, XMPP [64]).

The log entries from the Web server consist of several fields separated by a space character as show in the following example[1].

```
148.184.174.62 - - [09/Jul/2007:15:39:19 +0200]
"GET /index.html HTTP/1.1" 200 13894 "-"
"Mozilla/4.0 (compatible;)"
```

The most interesting field for this mashup is the first one. It contains the `remoteHost` – the address of the client visiting the Web site. In order to extract the relevant information, the splitLogLine task parses the log entry and provides

[1]The example log entry contains no linebreak characters even though it is shown on multiple lines here.

the values of individual fields of the entry in its output parameters. splitLogLine encapsulates the knowledge about the log format and in case the log entry format would change, it would be the only affected component. As shown in Table 7.1, the component is implemented with a regular expression, which allows for a fast parsing of the log entry.

The next task of the mashup deals with the data heterogeneity of the information extracted from the log entry. Usually, Web servers store raw IP addresses in their logs to save the overhead of a reverse DNS lookup for every access to the server. However, a server might be configured with address resolution. To decouple the mashup from the actual server configuration, we introduce an additional component. The resolveHost component takes as input either a numerical IP address or a symbolic hostname and ensures that its output always contains a raw IP address that fits with what is needed for looking up the corresponding geographical coordinates. Additionally, since we want to be able to display the hostname of the visitor in the user interface, this component also does a reverse lookup to retrieve the DNS hostname for the IP address (which, of course, is not necessary if this has already been done by the Web server). The hostname resolution is implemented with a Java snippet, relying on the services of the DNS [85].

The conversion of the IP address to geographical coordinates takes place in the lookupCoordinates task. In the example mashup, we use hostip.info [58] as the data provider. The service can be used through its RESTful interface which returns coordinates in plain text or XML. However, for high-volume applications, the entire database can be downloaded and deployed locally. For optimizing the performance of the mashup, lookupCoordinates accesses a local copy of the database through JDBC. The lookup is implemented as an SQL query to retrieve the name of the city and its coordinates for every IP address provided as input. Since the database contains an entry for every C-class subnet, the IP address (`%host%`) is converted to such a subnet before it is used in the `WHERE` clause of the query. Since the hostip.info database is not complete, it is possible that a certain IP address cannot be translated to geographical coordinates. In such a case, no information can be sent to be displayed in the visualization part of the mashup running in the browser. This data quality problem is handled by skipping the remaining tasks of the pipeline if no geographical information could be retrieved. This is done using a condition on the task following lookupCoordinates.

Before the coordinates can be sent to the browser, they need to be packaged in a suitable format. This is done in the packageJSON task. The format depends on the mechanism which is chosen to communicate with the JavaScript application running in the browser. In our case we chose to serialize the data using the JSON format [30]. To do so, we use a string concatenation expression, which replaces the names of the input parameters (bracketed with `%`) with their actual value. The transfer of data is done by the streamToBrowser task. To ensure that the data triggers an update of the map widget we wrap the data into `<script>` elements which are executed by the browser as they arrive. Since it is not possible to open a connection back to the browser, it is the responsibility of the mashup user interface to initiate the connection

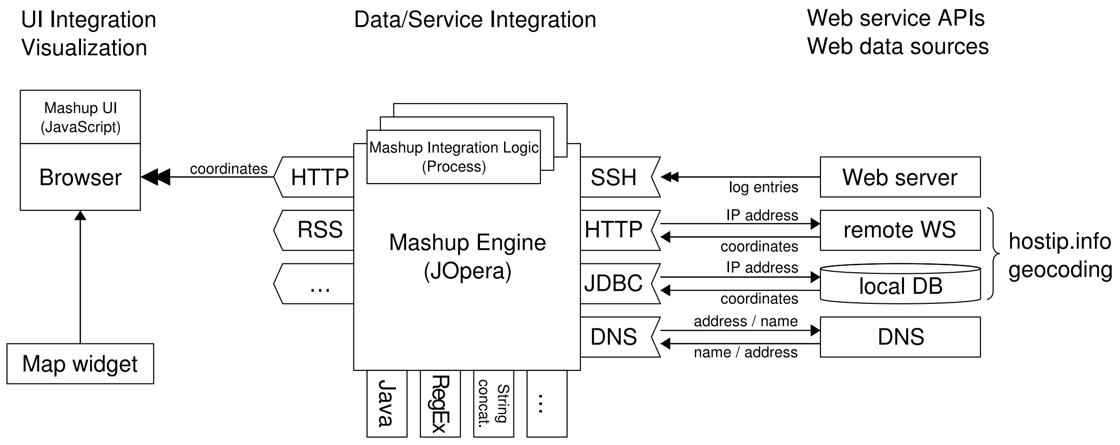


Figure 7.3: Layered Mashup Architecture

to the HTTP daemon embedded in JOpera. The URI of the HTTP request is used to specify from which process instance the stream of JSON coordinates should be received. The HTTP connection is kept open so the coordinates calculated by the process instance can be streamed – as part of an infinite HTTP response – directly to the corresponding user interface running in the browser.

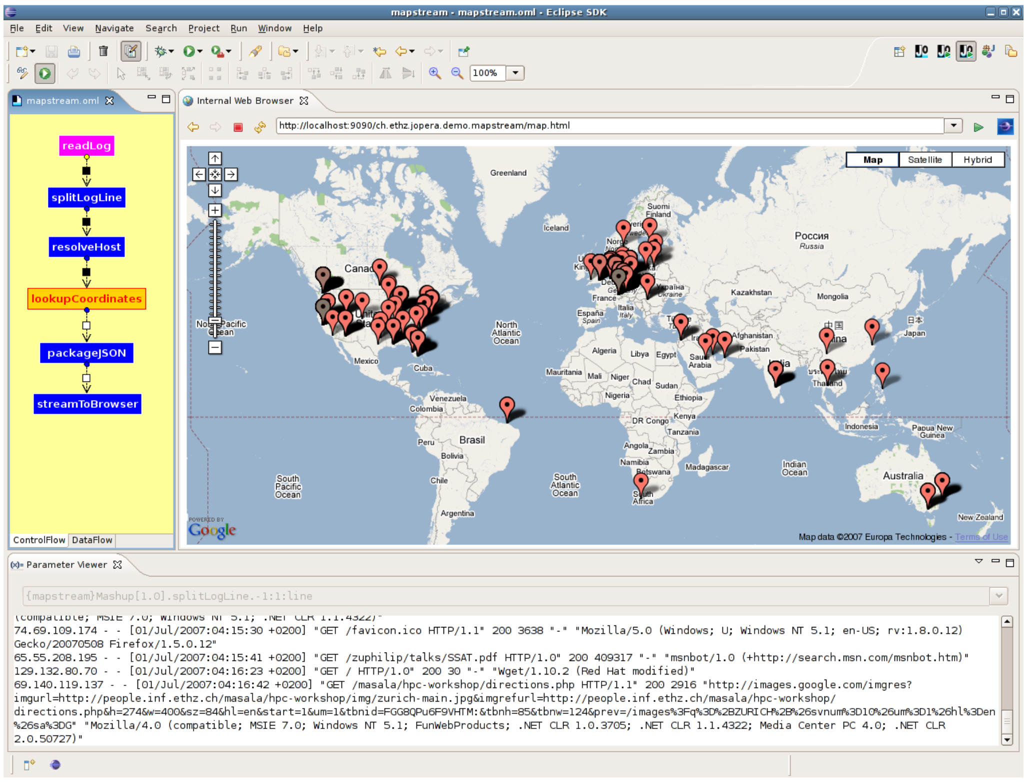
7.4 Layered Mashup Architecture

As shown in Figure 7.3 the mashup is designed with a layered architecture. This way it cleanly separates the user-interface part of the mashup, running in a browser as a rich AJAX application, from the integration logic, deployed in the mashup engine (JOpera) and the Web services and data source providers, which are connected to the engine through different task execution subsystems.

7.4.1 Mashup Engine

The *mashup engine* is at the core of the architecture. Here, the integration and coordination of different data sources takes place and new data products are made available to clients (like the browser or another mashup engine). As new data arrives, it enters the mashup process instance through a multiple-output task. Thanks to flow control, consecutive data items are streamed through the process instance in a pipelined fashion, increasing the throughput of the mashup.

To control the execution of a mashup pipeline, JOpera provides its own API to the mashup user interface. The mashup engine interacts with the mashup user interface through the task execution subsystems, i.e, the interaction with the interface is modeled by tasks in the mashup process. In the example, the interface runs a mashup pipeline and subscribes to its result stream. In a simpler case, the mashup integration logic can be invoked within a single request-response interaction.


mapstream - mapstream.oml - Eclipse SDK
File Edit View Navigate Search Project Run Window Help
mapstream.oml
readLog
splitLogLine
resolveHost
lookupCoordinates
packageJSON
streamToBrowser
ControlFlow DataFlow
Internal Web Browser
http://localhost:9090/ch.ethz.jopera.demo.mapstream/map.html
Map Satellite Hybrid
Map data ©2007 Europa Technologies - Terms of Use
Parameter Viewer
{mapstream}Mashup[1.0].splitLogLine.-1:1:line

```
(compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322)"
74.69.109.174 - - [01/Jul/2007:04:15:30 +0200] "GET /favicon.ico HTTP/1.1" 200 3638 "-" "Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.0.12)
Gecko/20070508 Firefox/1.5.0.12"
65.55.208.195 - - [01/Jul/2007:04:15:41 +0200] "GET /zuphilip/talks/SSAT.pdf HTTP/1.0" 200 409317 "-" "msnbot/1.0 (+http://search.msn.com/msnbot.htm)"
129.132.80.70 - - [01/Jul/2007:04:16:23 +0200] "GET / HTTP/1.0" 200 30 "-" "Wget/1.10.2 (Red Hat modified)"
69.140.119.137 - - [01/Jul/2007:04:16:42 +0200] "GET /masala/hpc-workshop/directions.php HTTP/1.1" 200 2916 "http://images.google.com/imgres?
imgurl=http://people.inf.ethz.ch/masala/hpc-workshop/img/zurich-main.jpg&imgrefurl=http://people.inf.ethz.ch/masala/hpc-workshop/
directions.php&h=274&w=400&sz=84&hl=en&start=1&um=1&tbnid=FGG8QPu6F9VHTM:&tbnh=85&tbnw=124&prev=/images%3Fq%3D%2BZURICH%2B%26svnum%3D10%26um%3D1%26hl%3Den
%26sa%3DG" "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; FunWebProducts; .NET CLR 1.0.3705; .NET CLR 1.1.4322; Media Center PC 4.0; .NET CLR
2.0.50727)"
```

Figure 7.4: Screenshot of the mashup example application running within the mashup development environment

In addition to interacting with remote services, JOpera also accesses local components (embedded in the corresponding subsystems) to support the evaluation of Java snippets, regular expressions and string concatenation expressions, and XML manipulation techniques (such as XPath, or XSLT). In the example mashup, this is used to implement the glue tasks, for which the overhead of a remote interaction is spared.

7.4.2 Mashup Development Environment

A screenshot of the JOpera development environment with the example mashup process is shown in Figure 7.4. The mashup user interface is running in the embedded browser view provided by Eclipse. Each task in the pipeline can be selected to watch the data flowing through its input and output parameters. The screenshot shows the original Web log entries in the bottom view as they are returned by the `readLog` task.

Such an IDE for the mashup integration logic complements existing tools for the development of mashup user interfaces (e.g., [49]). For example, when the Eclipse Web Tools Platform (WTP [37]) is used in conjunction with JOpera, it becomes possible to provide an integrated mashup development environment for the visual

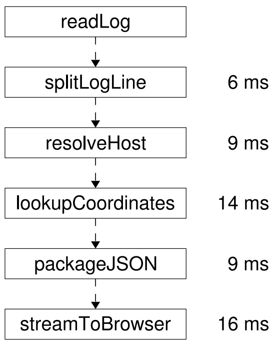


Figure 7.5: Delay introduced by each pipeline step.

design of the integration logic as a set of processes and the development of the corresponding AJAX frontend to display the results.

7.5 Performance Evaluation

As the aim of our mashup application is to provide the user with a real-time update of the interface, we have evaluated the performance of the mashup pipeline. The aim of this experiment was to estimate whether a layered architecture is capable of delivering a real-time user experience for our monitoring application.

During the experiment, the mashup engine, the geolocation database and the browser were each running on a machine equipped with an Intel Pentium 4 processor with Hyper-Threading enabled running Linux 2.6 at 3GHz and with 1 GiB of main memory. The mashup engine was executed by Sun's Java HotSpot Server VM 1.4.2 and the geolocation database was served by PostgreSQL 8.1.9. The user interface was displayed by Firefox 1.5.0.7.

In the experiment, we measured the processing time of each task in the pipeline (Figure 7.5). The readLog task has not been measured, because of the varying delay introduced by the buffering of the Web server when writing the log file. The processing inside the pipeline (from splitLogLine to packageJSON, inclusive) takes 41 ms. The delay for sending the resulting coordinates to the browser is 16 ms (including network latency for a local area connection). Thus, the total delay before a new Web request is reflected in the mashup user interface is less than 60 ms. Hence, our layered approach to mashup design is capable of fulfilling the real-time requirement of the monitoring application example.

7.6 Extending the Mashup Application Example

The presented example application already provides useful functionality. In this section, we enumerate several ideas for extensions and discuss their impact on the mashup pipeline.

To enhance the markers on the map widget, we can display additional information: the time of the visit, the URL which was requested or the type of browser (*user agent*) which was used, by forwarding the appropriate data from the splitLogLine component to the packageJSON component.

Without having to alter the structure of the mashup pipeline, the monitoring application can be turned into a log analysis tool by simply replacing the main data source of the mashup. By changing the command executed by the readLog component from "`tail -f`" to "`cat`", the entire log file is streamed through the data flow pipeline and, thus, also past requests are processed and progressively visualized by the mashup.

As the geolocation database we use is based on a community effort, the contained information is not complete, i.e., it does not cover the location of all possible IP addresses. To increase the likelihood of a successful lookup, the mashup could be extended to query a commercial provider of geolocation information (e.g., [80, 44, 113]) if the address could not be found in the local database. To do this, a new component is added in parallel to the existing lookupCoordinates component, with the same input and output parameters. The new component is then only triggered if lookupCoordinates does not return a result.

A more advanced extension is used to publish a subset of the annotated log entries in an RSS stream. Clearly, there is no point in providing the complete log over RSS. With the appropriate filtering component it is possible to provide notifications of "interesting" accesses to the Web server. This could include the detection of visitors which arrive through a search engine after entering certain interesting keywords, using the *referrer* field of the log entry. Filtering can also be employed to remove duplicate entries due to visitors accessing multiple resources on the Web server. To avoid looking up coordinates for the same IP address multiple times and sending these to the browser, the filtering component can be extended to skip requests from an already known visitor. To do so, a list of recently encountered IP addresses is accumulated and only addresses not in the list are forwarded to the next component in the pipeline.

7.7 Discussion

We have described an application of the streaming workflow engine developed in the thesis. The application poses several challenges which can be met with the extended system (except for the security issues). Most challenges such as heterogeneous services or maintainability can be handled by a flexible workflow engine. However, the real-time requirement calls for a streaming processing of the data being integrated. This translates to integrating a streaming service in the workflow process and being able to pipeline the process over the data stream to achieve a low processing delay. Being able to interact directly with a streaming service (Section 4.2.3) makes it possible to treat a stream as such, instead of having to handle each stream element separately, e.g., each in its own process instance. Flow control (Section 4.4) makes it possible to safely employ pipelining in the process and enables an end-to-end stream from the data source to the user interface.

The performance evaluation showed that we were able to meet the real-time requirements of the application, even by using a workflow engine which was originally not targeted at real-time applications. It should be noted that the notion of real time depends on the context in which it is used. Our mashup is able to display several markers per second on the map, which is sufficient to satisfy a user. However, the performance of the system might not be adequate to, e.g., analyze stock price tickers which have a rate of many thousands per second [129].

The most popular mashup tool that shares the notion of data flow pipeline with our approach is Yahoo! Pipes [128] (Section 2.1). The presented mashup application could be built with Yahoo! Pipes only if we assume that the Web server logs had been made accessible through HTTP and that a geolocation service was part of the palette. However, even with such a pipe, it would be difficult to provide real-time updates, as pipes have to be invoked in a request-response manner to retrieve their results. Thus, the user interface in the browser would have to poll the mashup's RSS feed periodically, entailing higher latencies and/or increased network traffic and server usage.

8 Conclusion

8.1 Summary

This dissertation has studied the problem of adding streaming capabilities to a conventional state-machine based workflow system. As a first step, we proposed different ways of connecting a workflow to a stream source. We then identified necessary extensions to the semantics of a state-based workflow language to cope with the problematic aspects of pipelined execution over a data stream. Based on a safely pipelined workflow execution, we showed how to ensure a high performance of the pipeline in the presence of irregular task executions by using a buffered data transfer inside the process.

In order to access a data stream from within a workflow process, we presented three alternatives to read the individual stream elements. The *pull* approach uses a conventional control flow loop and processes each stream element in a separate iteration. While this construct is supported by most workflow languages, it exhibits a poor performance. The *pull-push* approach is similar in that it uses a task in a loop to fetch stream elements. However, the loop fetches stream elements as fast as possible and pushes them into the remainder of the process, thereby creating a pipelining effect in the process. The third alternative takes the approach a step further by extending the task model. In the *push* approach, a multiple-output task can provide several results during one execution and does thus not require an explicit loop. This model increases the expressiveness of the workflow language as it makes the state of the stream explicit. Also, the approach unifies the access to different types of streams by abstracting from the mechanism or protocol used in communicating with the stream source. The subsystem used to access a stream is responsible for translating between the stream access protocol, which can have a pull, push or poll style, and the multiple-output model.

As pipelining has not been anticipated in the design of state-based workflow languages, using the pull-push or push stream access causes some safety problems in process. Since we argue that pipelining should be used to achieve a high stream processing performance, we have analyzed these problems in detail. We proposed different ways of dealing with collisions in a sequential pipeline which occur when a task is not ready to accept new input. The data can be discarded, be fed to the running task, be given to a new instance of the task, be appended to a buffer, or the pipeline can be stalled until the task is ready to accept the input. While these solutions are appropriate for linear task sequences, control flow merges exhibit the additional problem of state ambiguity. In search for a minimal solution to all pipelining problems, we extended the semantics of our meta-model with flow control. The flow control semantics allow a task to execute only if its last output has been consumed by all its successors and if there is sufficient input available from its predecessors.

By making the data exchange between tasks explicit, our meta-model acquires similarities with a graph-based data flow language. However, a major difference lies in the support for control flow constructs. In a data flow language the control flow is implicitly given by the way the data flows. Typically, a task is able to start only if data has arrived at all input ports. Alternatively, a task can read from its input ports in a predefined order. Both approaches are less flexible than our notion of activator which can be an arbitrary expression over the state of predecessors.

Real services do not have a constant response time. This is because the load on the service varies over time and because the service time may depend on the input to a service invocation. Consequently, the tasks in a workflow also take a varying time to execute. Thanks to flow control this does not pose a safety problem in processes with pipelining. However, as the measurements in Section 6.3 show, varying task durations have a negative impact on the performance. This motivated the extension of our system with the possibility of a buffered data transfer inside a process. Since the basic flow control mechanism already has the characteristics of small buffers, real buffers can be seamlessly integrated with the semantics of flow control. Buffers create an indirection in the information flow between tasks, and the buffer contents must be managed. Although it seems that this adds a considerable overhead to the process execution, buffers considerably improve the performance of a pipelined process, even for very small variations in the task duration as our measurements have shown.

In Chapter 5, we extended the runtime system with stream processing capabilities. The implementation of these extensions can be divided into three parts: extending the compiler to include the flow control semantics, runtime support for multiple-output tasks, and a buffer infrastructure. The extension of the compiler posed no big challenges, as the flow control semantics can be integrated by augmenting task activators with the evaluation of flow control information (flow control markers or buffer fill levels).

To support streams in the runtime system, more extensive changes were necessary. First, in order to support multiple-output tasks, the request-response model of task execution was extended to allow subsystems to deliver a stream of results. Second, since flow control is now being used in the process execution, the runtime system must also be able to control the flow of streaming results. Otherwise, pushing results into a process instance too fast, would compromise the flow control inside the process instance. Therefore the navigator–subsystem protocol was extended with flow control, thereby integrating the subsystem in the flow control chain in a general way. Third, in order not to constrain the architecture of a subsystem implementation, we added non-blocking operations for data delivery which makes it possible for a subsystem to deliver results without possibly being blocked indefinitely. We believe that our extended API for subsystems is flexible enough to support any type of streaming service.

The requirements towards buffers were implicitly defined in our description of how to achieve a buffered data transfer in a process. As part of the system extension, we formalized the interface of a buffer. The interface is kept simple to allow for implementations to use a variety technologies, such as databases or message-oriented middleware.

To evaluate our approach, we measured the impact of our stream processing extensions on the performance of the system. It turns out that flow control adds a considerable overhead to process navigation. This is due to the additional decisions which must be made when evaluating the starting conditions of a task. Our approach to stream processing contrasts with the approach of data stream management systems (DSMS). Whereas DSMS focus on the high-performance stream processing using a restricted set of well-defined operators, our approach puts the emphasis on the integration of heterogeneous components, thereby accepting that not the same performance can be achieved. This thesis extends the types of services which can be composed to include streaming services. As the mashup application in Chapter 7 shows, the performance of our system is still high enough to also deliver real-time stream processing in certain application domains. In addition, flow controlled pipelines show a clear performance benefit in a different area. If a number of pre-allocated pipelines is being used to implement a service instead of dynamically allocating a new process instance for each incoming service request, a considerable amount of memory can be saved.

Finally, a real-time streaming Web mashup was used to demonstrate the possibilities of our stream processing workflow system. The application shows how the system is used to solve different challenges in building the mashup. In particular, our approach abstracts from the mechanism used to access a stream by providing a unified model on the composition level. The integration of heterogeneous data sources is possible, especially of streaming and non-streaming ones. Pipelining permits the processing of log entries in a real-time streaming fashion. Thanks to the workflow-based service composition approach, our stream processing system provides control flow constructs such as conditional branching and exception handling which are typically not available in stream processing tools based on pure data flow languages.

8.2 Future work

The work in this thesis can be extended in several directions some of which we describe in the following.

Advanced pipelining In Section 4.3.1 we described several alternatives to cope with pipeline collisions stemming from differences in execution duration of tasks. Although the flow control semantics proposed in this thesis make it possible to use pipelining in a safe manner, it does not solve all problems associated with pipelined processing. A constantly slow task in a pipeline may create a bottleneck in a pipeline which slows down the entire pipeline. A solution to this problem is to allow multiple instances of tasks (Section 4.3.1). This superscalarity approach is also employed in microprocessors to reduce bottlenecks by scaling out certain stages of the pipeline. As mentioned earlier, handling multiple instances of a task is not trivial. E.g., in a multiple-instance task, stream elements might overtake each other if the task instances are not properly synchronized. This is a problem if the elements of the stream should be processed in the proper order.

Even with flow control in place to prevent the loss of data in a pipeline, multiple-input tasks (Section 4.3.1) can be an interesting programming concept as it allows the processing of multiple stream elements in the context of the same task execution. For instance, if some computation on a stream can be offloaded to a data stream management system, this service would be best modeled as a task with streaming input and output. In contrast to systems like OSIRIS-SE [15], the data would hereby still flow through the workflow engine. Analogous to the multiple-output feature (Section 4.2.3), multiple-input would also require the extension of the service model.

Modularization In Section 4.8 we discussed how subprocesses can be used to create isolated sections in a pipeline. Inside such a section, pipelining is turned off and only one data element is processed at a time. However, this defeats the original motivation for subprocesses, the modularization of complex processes. A pipeline cannot be split into several modules using this approach, as pipelining is not possible across the boundaries of a subprocesses. Therefore, subprocesses should provide the option of being transparent in terms of pipelining to allow a data stream to seamlessly flow from a process into a subprocess. This behaviour corresponds to a multiple-input task, since data must be pushed into a subprocess while it is already running. The two extensions could probably share a common solution.

Fine grained flow control We always assumed that an entire process is subject to flow control or buffering. At the same time, we emphasized that the flow of information is always controlled between pairs of tasks. Without a big effort, it should be possible to apply flow control selectively only to parts of a process. This would also make it possible to allow the *discard* semantics (Section 4.3.1) for pipeline collisions at selected tasks if, e.g., it is more important to always process the most recent data as opposed to processing the entire stream. On the process design level this could be specified through the annotation of single task dependencies or entire groups of tasks. This would give the user fine grained control over the amount of flow control that is used in a process. Additionally, the process editor could analyze the process or use runtime information to make suggestions in this regard.

Flow control for BPEL As our work targets traditional state-based workflow languages, our approach should be validated by applying it to BPEL which is understood by many workflow engines. We see at least the following challenges in doing so.

- *Stream access* As BPEL does not support the explicit notion of task execution state, the push approach (Section 4.2.3) cannot be used to access a stream from a process. Rather, a solution similar to the pull-push approach (Section 4.2.2) must be chosen, where a task has the ability to restart itself.
- *Variables* BPEL uses variables for passing data between activities[1]. It must be ensured that variables are used only for the data exchange between two tasks. Otherwise, the data in a variable will most likely be overwritten at the wrong moment by a third task.
- *Structured activities* BPEL supports both a block structured modeling approach, using *structured activities*, as well as the notion of *links* between activities as used in our "flat graph" approach. While the *link status* of links seem suitable for flow control (corresponding to our Boolean marker), structured activities do not provide an explicit notion of dependencies between activities. This could be solved by converting structured activities to flat activity graphs with the appropriate links. However, this cannot be applied to all structured activities, such as the while loop, because the loop would get mixed with the rest of the pipeline, creating an unwanted feedback loop in the pipeline.

Merging streams Our meta-model supports control flow merges which can be used to execute a task over the data from more than one stream. However, the model does not take into account that the inputs for the task might originate from a stream. An example is an asynchronous control flow merge (OR) where two input parameters of a task are fed from two independent streams. When the task is executed, it is currently not possible for the task execution to determine which of the streams triggered the execution. This is because the input parameters are not cleared between executions and typically contain old data. In fact, this is the state ambiguity problem which was discussed in Section 4.3.2 which here recurs at a different level.

Optimized navigation As mentioned earlier, our implementation of flow control was not focused on achieving a high-performance, but rather on providing safe pipelining with only minimal changes to the system. Therefore, and in view of the results of the performance measurements, ways of lowering the overhead of flow control should be investigated. The structure of the navigation code generated by the compiler constrains to a high degree the way flow control can be incorporated. Therefore, it should be considered to reconceive how navigation is implemented.

[1]In BPEL, tasks are called *activities*.

Bibliography

[1] G. D. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Trans. Softw. Eng. Methodol.*, 4(4):319–364, 1995. 111

[2] Active Endpoints. *ActiveBPEL Engine Architecture*, 2007. `http://www.active-endpoints.com/open-source-architecture.htm`. 12

[3] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer, 2004. 1, 13, 86, 92

[4] M. Altinel, P. Brown, S. Cline, R. Kartha, E. Louie, V. Markl, L. Mau, Y.-H. Ng, D. E. Simmen, and A. Singh. DAMIA - A Data Mashup Fabric for Intranet Applications. In *VLDB*, 2007. 6, 111

[5] Apache Software Foundation. Apache ActiveMQ Persistence. `http://activemq.apache.org/persistence.html`. 86

[6] Apache Software Foundation. log4j. `http://logging.apache.org/log4j/`. 57

[7] Apple, Inc. MacOS X Automator Website. `http://www.automator.us/`. 5

[8] R. Asleson and N. T. Schutta. *Foundations of Ajax*. APress, October 2005. 114

[9] M. J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, 1986. 5

[10] A. Barros, M. Dumas, and A. ter Hofstede. Service Interaction Patterns. In *Proceedings of the 3rd International Conference on Business Process Management, Nancy, France*. Springer Verlag, September 2005. 7

[11] W. Bausch. *OPERA-G : a microkernel for computational grids*. PhD thesis, ETH Zurich, 2004. 1, 12

[12] K. Bennett. Legacy Systems: Coping with Success. *IEEE Softw.*, 12(1):19–23, 1995. 112

[13] Berkman Center for Internet & Society at Harvard Law School. *RSS 2.0 Specification*. `http://blogs.law.harvard.edu/tech/rss`. 1, 6, 35, 57, 113

[14] B. Biörnstad and C. Pautasso. Let it Flow: Building Mashups with Data Processing Pipelines. In *Proc. of ICSOC 2007 Workshops, Vienna, Austria*, LNCS 4907, September 2007. 111

[15] G. Brettlecker, H. Schuldt, and R. Schatz. Hyperdatabases for Peer–to–Peer Data Stream Processing. In *Proc. of ICWS Conf.*, San Diego, CA, USA, 2004. 8, 41, 126

[16] G. Brettlecker, H. Schuldt, and H.-J. Schek. Towards Reliable Data Stream Processing with OSIRIS-SE. In *Proc. of BTW Conf.*, Karlsruhe, Germany, 2005. 8

[17] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous Concurrent Modeling and Design in Java (Volume 3: Ptolemy II Domains). Technical Report UCB/EECS-2007-9, EECS Department, University of California, Berkeley, Jan 2007. 9

[18] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective.* Prentice Hall, 2003. 5, 68

[19] Bull SAS. *Bonita engine reference guide*, January 2007. `http://wiki.bonita.objectweb.org/xwiki/bin/download/Main/Documentation/Bonita_API.pdf`. 12

[20] Business Process Management Initiative. *Business Process Modeling Notation (BPMN), Version 1.0*, May 2004. 11

[21] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Monitoring Streams - A New Class of Data Management Applications. In *VLDB*, 2002. 35

[22] U. Çetintemel, D. Abadi, Y. Ahmad, H. Balakrishnan, M. Balazinska, M. Cherniack, J. Hwang, W. Lindner, S. Madden, A. Maskey, A. Rasin, E. Ryvkina, M. Stonebraker, N. Tatbul, Y. Xing, and S. Zdonik. The Aurora and Borealis Stream Processing Engines. In M. Garofalakis, J. Gehrke, and R. Rastogi, editors, *Data Stream Management: Processing High-Speed Data Streams.* Springer-Verlag, July 2006. 8

[23] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003. 8, 35

[24] L. Chen, M. Li, J. Cao, and Y. Wang. An ECA Rule-based Workflow Design Tool for Shanghai Grid. In *SCC '05: Proceedings of the 2005 IEEE International Conference on Services Computing*, Washington, DC, USA, 2005. IEEE Computer Society. 12

[25] R. Clauberg. RFID and Sensor Networks: From Sensor/Actuator to Business Application. In *RFID-Workshop.* University of St.Gallen, Switzerland, April 2004. 1

[26] con:cern Project. *con:cept Manual.* `http://con-cern.org/wiki/en/Manual`. 11

[27] con:cern Project. *The Object oriented Approach to Process Management.* `http://con-cern.org/wiki/en/The_Object_oriented_Approach_to_Process_Management`. 11

[28] Coraider Services Ltd. Bumblebee Auctions. `http://www.bumblebeeauctions.co.uk/`. 1

[29] C. Courbis and A. Finkelstein. Towards an Aspect Weaving BPEL engine. In *Proc. 3rd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, Lancaster, UK, March 2004. 12

[30] D. Crockford. JSON: The Fat-Free Alternative to XML. In *Proc. of XML 2006*, Boston, USA, December 2006. `http://www.json.org/fatfree.html`. 116

[31] F. Curbera, R. Khalaf, W. A. Nagy, and S. Weerawarana. Implementing BPEL4WS: the architecture of a BPEL4WS implementation. *Concurrency and Computation: Practice and Experience*, 18(10):1219–1228, 2006. 12

[32] F. Daniel, M. Matera, J. Yu, B. Benatallah, R. Saint-Paul, and F. Casati. Understanding UI Integration: A Survey of Problems, Technologies, and Opportunities. *IEEE Internet Computing*, 11(3):59–66, May-June 2007. 111

[33] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005. 1

[34] M. Dumas, W. M. P. van der Aalst, and A. H. M. ter Hofstede, editors. *Process-Aware Information Systems.* Wiley, 2005. 1

[35] S. Dustdar and W. Schreiner. A survey on web services composition. *Int. J. Web and Grid Services*, 1(1):1–30, 2005. 1

[36] Eclipse Foundation. Eclipse Java Development Tools (JDT) Website. `http://www.eclipse.org/jdt/`. 29

[37] Eclipse Foundation. *Eclipse Web Tools Platform.* `http://www.eclipse.org/webtools/`. 118

[38] Eclipse Foundation. Eclipse Website. `http://www.eclipse.org/`. 29

[39] FeedBurner, Inc. FeedBurner. `http://www.feedburner.com/`. 1

[40] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1.* IETF, June 1999. RFC 2616. 6

[41] Flake Media. *GVisit.* `http://www.gvisit.com/`. 112

[42] I. Foster. What is the Grid? A Three Point Checklist. `http://www-fp.mcs.anl.gov/~foster/Articles/WhatIsTheGrid.pdf`, July 2002. 1

[43] M. Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley, November 2002. 114

[44] FraudLabs.com. *IP2Location.* `http://www.fraudlabs.com/ip2location.aspx`. 113, 120

[45] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995. 12

[46] D. Ganesarajah and E. Lupu. Workflow-based composition of web-services: a business model or a programming paradigm? In *Proc. 6th International Enterprise Distributed Object Computing Conference (EDOC'02)*, 2002. 12

[47] D. Georgakopoulos, M. F. Hornick, and A. P. Sheth. An Overview of Workflow Management: From Process Modelling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, April 1995. 10, 18

[48] L. Golab and M. T. Özsu. Issues in Data Stream Management. *SIGMOD Record*, 32(2):5–14, June 2003. 1, 8

[49] Google, Inc. *Google Mashup Editor – Getting Started Guide*, 2007. `http://code.google.com/gme/docs/gettingstarted.html`. 118

[50] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification.* The Java Series. 2nd edition, June 2000. 24

[51] P. Grefen and R. R. de Vries. A reference architecture for workflow management systems. *Data & Knowledge Engineering*, 1998. 12

[52] D. Grigori, F. Charoy, and C. Godart. Anticipation to Enhance Flexibility of Workflow Execution. In *DEXA 2001*, volume 2113 of *LNCS*, 2001. 9

[53] T. Gunarathne, D. Premalal, T. Wijethilake, I. Kumara, and A. Kumar. BPEL-Mora: Lightweight Embeddable Extensible BPEL Engine. In C. Pautasso and C. Bussler, editors, *Emerging Web Services Technology*, Whitestein Series in Software Agent Technologies and Autonomic Computing. Birkhäuser Basel, 2007. 12

[54] C. Hagen. *A Generic Kernel for Reliable Process Support.* PhD thesis, ETH Zurich, 1999. 12

[55] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987. doi:10.1016/0167-6423(87)90035-9. 11

[56] J. P. Hartmann. CMS Pipelines Explained. In *Proc. of SHARE 88*, 1988. 5

[57] D. Hollingsworth. The Workflow Reference Model: 10 Years On. In *Workflow Handbook 2004*. Workflow Management Coalition. 1

[58] hostip.info. *hostip.info Website.* `http://www.hostip.info/`. 113, 116

[59] P. Hudak, J. Peterson, and J. Fasel. *A Gentle Introduction to Haskell, version 98.* June 2000. `http://www.haskell.org/tutorial/`. 7

[60] K. Hwang and F. A. Briggs. *Computer Architectures and Parallel Processing.* McGraw-Hill, 1985. 37

[61] IBM Corporation. *BatchPipes OS/390 V2R1 Introduction*, September 1995. 6

[62] International Business Machines Corp. *IBM DAMIA*. `http://services.alphaworks.ibm.com/damia/`. 6

[63] International Telecommunication Union. *List of definitions for interchange circuits between data terminal equipment (DTE) and data circuit-terminating equipment (DCE)*, February 2000. ITU-T Recommendation V.24. 6

[64] Internet Engineering Task Force. *Extensible Messaging and Presence Protocol (XMPP): Core*. `http://ietfreport.isoc.org/rfc/rfc3920.txt`. 115

[65] IP2Location.com. *IP2Location*. `http://www.ip2location.com/`. 113

[66] IPligence Ltd. *IPligence*. `http://www.ipligence.com/`. 113

[67] JBoss. *JBoss jBPM jPDL 3.2, jBPM jPDL User Guide*, February 2006. `http://docs.jboss.com/jbpm/v3/userguide/`. 12

[68] JBoss. *JBoss Messaging 1.2 User's Guide*, March 2007. `http://labs.jboss.com/file-access/default/members/jbossmessaging/freezone/docs/userguide-1.2.0.GA/html/index.html`. 86

[69] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74*, 1974. 7

[70] G. Keller, M. Nüttgens, and A.-W. Scheer. Semantische Prozeßmodellierung auf der Grundlage "Ereignisgesteuerter Prozeßketten (EPK)". Technical Report 89, Institut für Wirtschaftsinformatik, Universität des Saarlandes, Januar 1992. 11

[71] J. Krämer and B. Seeger. PIPES - A Public Infrastructure for Processing and Exploring Streams. In G. Weikum, A. C. König, and S. Deßloch, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*. ACM, 2004. 8

[72] R. Krishnan, L. Munaga, and K. Karlapalem. XDoC-WFMS: A Framework for Document Centric Workflow Management System. In *Conceptual Modeling for New Information Systems Technologies*, 2002. 12

[73] M. Lampe, C. Floerkemeier, and S. Haller. Einführung in die RFID-Technologie. In E. Fleisch and F. Mattern, editors, *Das Internet der Dinge – Ubiquitous Computing und RFID in der Praxis*. Springer-Verlag, 2005. 17

[74] E. A. Lee and T. M. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, May 1995. 2, 7

[75] F. Leymann. Web Services Flow Language (WSFL 1.0). Technical report, IBM, May 2001. 10

[76] F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall, 1999. 1, 12, 13, 20

[77] F. Leymann, D. Roller, and M.-T. Schmidt. Web services and business process management. *IBM Systems Journal*, 41(2):198–211, 2002. 101

[78] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. Lee, J. Tao, and Y. Zhao. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice & Experience*, 18(10):1039–1065, 2006. 9

[79] S. Majithia, M. S. Shields, I. J. Taylor, and I. Wang. Triana: A Graphical Web Service Composition and Execution Toolkit. In *Proceedings of the IEEE International Conference on Web Services (ICWS'04)*. IEEE Computer Society, 2004. 8

[80] MaxMind LLC. *GeoIP*. `http://www.maxmind.com/app/ip-location`. 113, 120

[81] T. M. McPhillips and S. Bowers. An Approach for Pipelining Nested Collections in Scientific Workflows. *SIGMOD Record*, 34(3), September 2005. 9

[82] J. Mendling, G. Neumann, and M. Nüttgens. Towards Workflow Pattern Support of Event-Driven Process Chains (EPC). In M. Nüttgens and J. Mendling, editors, *XML4BPM 2005, Proceedings of the 2nd GI Workshop XML4BPM – XML Interchange Formats for Business Process Management at 11th GI Conference BTW 2005, Karlsruhe Germany*, March 2005. 11, 12

[83] Microsoft Corporation. *Popfly*. `http://www.popfly.ms/`. 111

[84] J. Misra. Detecting termination of distributed computations using markers. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, New York, NY, USA, 1983. ACM. 62

[85] P. Mockapetris. *Domain Names - Concepts and Facilities*. ISI, November 1987. `http://www.ietf.org/rfc/rfc1034.txt`. 116

[86] T. Murata. Petri Nets: Properties, Analysis and Applications. In *Proceedings of the IEEE*, volume 77, April 1989. 9, 11

[87] M. Nottingham and R. Sayre. The Atom Syndication Format. RFC 4287 (Proposed Standard), Dec. 2005. 1

[88] OASIS. *ebXML Business Process Specification Schema Technical Specification v2.0.4*, December 2006. `http://docs.oasis-open.org/ebxml-bp/2.0.4/`. 11

[89] OASIS. *Web Services Business Process Execution Language Version 2.0*, May 2006. `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`. 1, 10, 20, 35, 101

[90] Object Management Group. *Unified Modeling Language: Superstructure, version 2.0*, August 2005. 9, 11, 20

[91] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004. 11

[92] S. G. Parker. *The SCIRun Problem Solving Environment and Computational Steering Software System.* PhD thesis, The University of Utah, 1999. 9, 12

[93] C. Pautasso. JOpera for Eclipse Website. `http://www.jopera.org/`. 29

[94] C. Pautasso. *A Flexible System for Visual Service Composition.* PhD thesis, Diss. ETH Nr. 15608, July 2004. 10, 13, 21, 30, 101

[95] C. Pautasso and G. Alonso. From Web Service Composition to Megaprogramming. In *Proceedings of the 5th VLDB Workshop on Technologies for E-Services (TES-04)*, Toronto, Canada, August 2004. 1

[96] C. Pautasso and G. Alonso. Parallel Computing Patterns for Grid Workflows. In *Proc. of the HPDC2006 Workshop on Workflows in Support of Large-Scale Science (WORKS06)*, Paris, France, June 2006. 1, 10

[97] B. Plale. Framework for bringing data streams to the grid. *Scientific Programming*, 12(4):213–223, 2004. 1

[98] Quova. *GeoPoint.* `http://www.quova.com/`. 113

[99] L. Richardson and S. Ruby. *RESTful Web Services.* O'Reilly, May 2007. 113

[100] N. Russell, A. ter Hofstede, W. van der Aalst, and N. Mulyar. Workflow Control-Flow Patterns: A Revised View. Technical Report BPM-06-22, BPMcenter.org, 2006. 9, 12, 42, 62, 63, 68

[101] N. Sample, D. Beringer, and G. Wiederhold. A comprehensive model for arbitrary result extraction. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, New York, NY, USA, 2002. ACM Press. 2, 7

[102] C. Schuler, R. Weber, H. Schuldt, and H.-J. Schek. Scalable Peer-to-Peer Process Management – The OSIRIS Approach. In *Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, 2004. 12

[103] K. Scott. *Fast Track UML 2.0.* Apress, 2004. 11

[104] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34(3):31–36, 2005. 85

[105] R. Stephens. A survey of stream processing. *Acta Infomatica*, 34(7), 1997. 2, 7

[106] W. Swierstra. Stream-0.2.2: A library for manipulating infinite lists. `http://hackage.haskell.org/packages/archive/Stream/0.2.2/doc/html/Data-Stream.html`. 7

[107] A. S. Tannenbaum. *Computer Networks.* Prentice Hall, 3rd edition, 1996. 6, 7, 92

[108] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. In *International Conference on Very Large Data Bases (VLDB'03)*, Berlin, Germany, September 2003. 41

[109] I. Taylor, M. Shields, I. Wang, and R. Philp. Distributed P2P Computing within Triana: A Galaxy Visualization Test Case. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, Washington, DC, USA, 2003. IEEE Computer Society. 2

[110] Technorati, Inc. Technorati. `http://technorati.com/`. 1

[111] S. Thatte. XLANG: Web services for Business Process Design. Technical report, Microsoft, May 2000. 10

[112] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *International Conference on Compiler Construction*, Grenoble, France, April 2002. 8

[113] Tometa Software, Inc. *Tometa WhereIs*. `http://www.tometasoftware.com/products_twi.asp`. 113, 120

[114] University of Southampton IT Innovation Centre. Freefluo Website. `http://freefluo.sourceforge.net/`. 11

[115] U.S. Geological Survey. Earthquake Hazards Program. `http://earthquake.usgs.gov/`. 1, 101

[116] W. M. P. van der Aalst. The Application of Petri Nets to Workflow Management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998. 9, 11

[117] W. M. P. van der Aalst, L. Aldred, M. Dumas, and A. H. M. ter Hofstede. Design and implementation of the YAWL system. In *Proceedings of The 16th International Conference on Advanced Information Systems Engineering (CAiSE 04)*, June 2004. 9

[118] W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL: Yet another workflow language. *Information Systems*, 30:245–275, June 2005. 9, 12

[119] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(3):5–51, July 2003. 9, 63

[120] W3C. *XSL Transformations (XSLT)*, 1999. `http://www.w3.org/TR/xslt`. 13

[121] W3C. *Web Services Choreography Description Language Version 1.0*, November 2005. W3C Candidate Recommendation. 11

[122] W3C. *SOAP Version 1.2 Part 0: Primer (Second Edition)*, April 2007. `http://www.w3.org/TR/soap12-part0/`. 101, 113

[123] Wikipedia. Mashup (web application hybrid) — Wikipedia, The Free Encyclopedia, 2008. `http://en.wikipedia.org/wiki/Mashup_(web_application_hybrid)`. 111

[124] D. Wodtke and G. Weikum. A Formal Foundation for Distributed Workflow Execution Based on State Charts. In *ICDT '97: Proceedings of the 6th International Conference on Database Theory*, London, UK, 1997. Springer-Verlag. 11

[125] Workflow Management Coalition. XPDL Support & Resources. `http://www.wfmc.org/standards/xpdl.htm`. 11

[126] Workflow Management Coalition. *Terminology & Glossary*, February 199. Document Number WFMC-TC-1011. 1

[127] Workflow Management Coalition. *Process Definition Interface – XML Process Definition Language*, October 2005. Version 2.00. 11

[128] Yahoo! Inc. *Pipes*. `http://pipes.yahoo.com/`. 6, 111, 121

[129] Y. Zhu and D. Shasha. StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. In *VLDB*, 2002. 121