Control the Flow: How to Safely Compose Streaming Services into Business Processes

Biörn Biörnstad, Cesare Pautasso, Gustavo Alonso
Department of Computer Science, ETH Zurich
8092 Zurich, Switzerland
{bioernstad,pautasso,alonso}@inf.ethz.ch

Abstract

Although workflow languages are widely used for composing discrete services, these are not suitable for stream based interactions. In this paper we address the problem of how to extend a conventional Web service composition language with the ability to deal with data streaming services. The paper discusses several modeling alternatives and presents a marker based semantics for safely dealing with pipelined processing in service compositions. The paper also presents application examples that illustrate the advantages of the proposed approach.

1. Introduction

Service Oriented Architectures and Web service composition are often closely related to workflow and business process management tools (BPM) [10]. Such tools 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. 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 [13].

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 business process. Examples include RSS feeds listing the latest bids at an auction, result tuples from a data stream management system (DSMS) [4, 5], stock price tickers, etc. In this paper, we focus on such streams of business messages. We do not target audio or video streams which are typically processed in real-time.

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 WS-BPEL, 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 paper we discuss how to extend traditional, state-based business process modeling techniques [6, 7, 9] with the necessary features to integrate streaming data services and combine them with conventional request response services. The paper discusses the problem in detail and proposes minimal but necessary extensions to the semantics of a business process modeling language to support streaming. Although we present the work in the context of a concrete workflow engine (JOpera [16]), the ideas in the paper are generic and can be applied to any business process engine using a traditional stepwise workflow model (e.g., many implementations of WS-BPEL). In the paper we use the Business Process Modeling Notation (BPMN) [3] to illustrate processes.

The paper is organized as follows. In Section 2 a basic non-streaming workflow model is introduced. This model is then extended to introduce streaming services that provide access to data stream sources (Section 3). Section 4 discusses strategies to avoid collisions of multiple task executions in a pipeline and identifies the problems related to pipelined execution over multiple stream elements. Section 5 presents a solution to the problems introduced by pipelining. In Section 6 we present example applications of streaming workflows. Section 7 discusses related work and Section 8 concludes the paper.

2. Background

2.1. Processes

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 [9]. A *task* is a step to be executed as part of a process. Each task specifies how it is executed (e.g., which Web service to call)

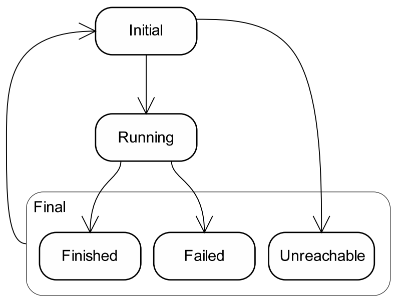

Figure 1. State machine of a task instance.

and may have several input and output parameters. The order in which the tasks are executed is specified by the task dependencies.

To be executed, a process needs to be instantiated. A *process instance* consists of the state necessary for the execution of the process [7]. During its lifetime, an instance goes through a sequence of *states* (Figure 1). 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 it becomes *Running* and eventually is *Finished*. If the execution of a task fails for some reason, this is indicated with the *Failed* state.

When a process instance is created, all of its tasks without predecessors (called *initial tasks*) are started and the process instance goes to the *Running* state. The task dependencies specify how the process execution continues from there.

2.2. Navigation

There are two types of task dependencies: control flow and data flow. The control flow 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* has reached a certain state, e.g., *Finished*. According to this ordering, every task has a (possibly empty) set of immediate *predecessors* and *successors*.

A data flow dependency connects an output parameter with an input parameter of two respective tasks. After a task has been executed, data is copied from its output to input parameters of other tasks according to the data flow graph. Since a task must not execute until data has been copied to its input parameters, a data flow dependency implies a control flow dependency between the corresponding tasks.

When a task terminates, its successors are checked 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. The *starting conditions* of a task are specified as an *activator* and a *data-condition*. The activator is a boolean expression over all incoming control flow dependencies. If the activator is satisfied, the data-condition is evaluated. This condition is a user-defined boolean expression referencing input parameters of the task and allowing a process to make data-dependent decisions. If the data-condition also evaluates to true, the corresponding task is started and its state becomes *Running*. In case the data-condition is not satisfied, the task becomes *Unreachable*. This state is used in the dead-path-elimination algorithm. Tasks which depend on unreachable tasks also become unreachable and thus the ”dead path” is made explicit. Dead-path-elimination helps in determining when the process instance has finished.

A process instance continues running as long as there are tasks which are running. As soon as all tasks have reached a final state (*Finished*, *Failed*, *Unreachable*) the process is considered to be finished as well. Thus, a process instance terminates *implicitly* as opposed to being explicitly stopped by a special task in the process.

2.3. Loops

A simple loop in a workflow can be used to process streams by handling one stream element in each loop iteration. Such an approach requires to reset tasks from a final state to the *Initial* state (Figure 1). 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 reexecute 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.

When the last task in a loop iteration has been executed, the state of the tasks in the loop are reset before the next iteration. The end of an iteration is detected by inspecting the state of the successor task whenever a task finishes. If the successor is not in the *Initial* state, this must be the first task in the loop since it has already been executed. To reset the loop, all its tasks are put into the *Initial* state by starting from the first task and recursively following the control flow. To prevent an infinite recursion due to nested loops in the control flow, branches are not followed beyond a task which is already in the *Initial* state. The algorithm also stops before resetting the last task in the loop. This task is left in its current state, so the starting conditions of the first task in the loop can be satisfied. After the loop has been restarted, the last task is also reset.

3. Consuming 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

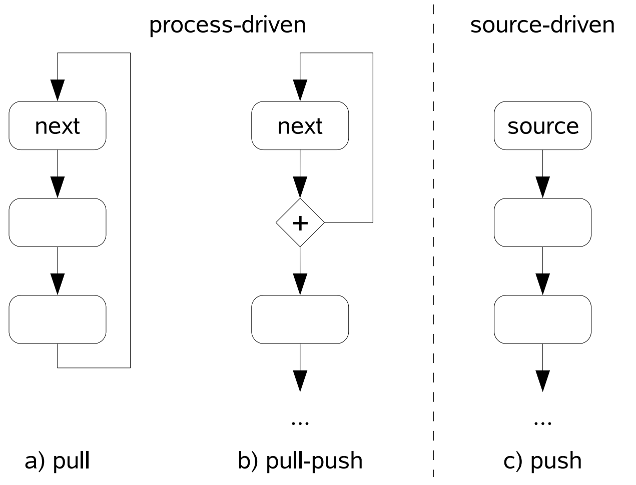

Figure 2. Different alternatives for continuously consuming data from a stream source.

challenge to do 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 2). 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.

3.1. Pull

The first case (Figure 2a) 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 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.

3.2. Pull-Push

The second option (Figure 2b) is to make the control flow cycle contain only the source task. Therefore, when the task

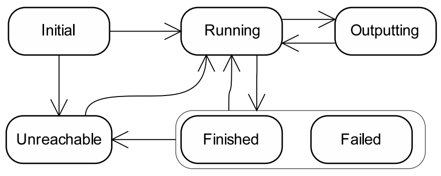

Figure 3. Finite state machine for a task instance with multiple output.

has been executed it is restarted immediately. At every iteration the tasks following the loop are also restarted in order to process the data (AND-split). 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 [8], meaning that the pipeline (the sequence of tasks) is processing multiple data elements concurrently, each in a different task. This can improve the throughput of the workflow compared to the pull approach.

3.3. Push

The third approach (Figure 2c) goes even further and makes the loop smaller than one task. The actual loop happens "inside" the task because the task runs forever (or at least as long as there is data coming from the stream source). This requires 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*. For this, we introduce the *Outputting* state (Figure 3). The task is started as usual by making the transition from the *Initial* state to *Running*. When there is output available, the task temporarily goes to *Outputting* and then goes back to *Running*. Successor tasks which want to consume this output use an *Outputting* control flow dependency on the source task. If the data stream is not infinite, the source task will eventually become *Finished*. Compared to the push-pull, this 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 task with a *Finished* dependency on the source task.

4. Pipelined execution

In the previous section, we have introduced task pipelines which are started either by a loop with an AND-split or by a task which produces multiple outputs. However, the model presented in Section 2 has some safety problems when it is used to model pipelined workflow execution. Considering that 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. In this case task *T* would be restarted. This is problematic because the task's output parameters would need to be shared between the parallel overlapping executions.

To address this, we need to define what should happen if a task becomes ready to be started while it is already running. The following briefly compares three ways to deal with such an *execution opportunity* discussing their impact on the semantics of tasks.

4.1. Multiple input

The first 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 3.3). 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 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 3). In [1] the authors take a similar approach and employ stream operators as tasks to process streaming data.

We have decided not to pursue this approach for two reasons. 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 to the application, with the engine acting solely as a configuration tool of a data stream processing pipeline.

4.2. 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 [20]. 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.

Due to the additional problems it creates both at the engine design and the semantics levels, we do not pursue this design option any further in this paper.

4.3. 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 third 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. Therefore, we choose this strategy to control the pipelines in processes.

Although, as a first approximation, queues may have unlimited capacity and thus are able to handle an infinite number of execution opportunities, in order to implement this semantics the engine can only provide queues of limited capacity. When a queue gets full, execution opportunities have to be ignored because the input data can no longer be queued.

Without loss of generality, this is equivalent to having the data buffered in the input parameters of the tasks as the queue does not have any capacity left for the next input data. For a simple sequence of two tasks *A* and *B* we observe two different problems:

Output overwriting: *A* finishes and triggers the execution of *B*. While *B* is running, *A* finishes again. This time the execution of *B* is deferred because *B* is running. Later, while *B* is still running, *A* finishes yet another time. *A* now has overwritten its previous output with the new output. However, *B* could not consume the old output before it was overwritten, so the corresponding stream element is lost.

State ambiguity: *A* executes and becomes *Finished* which triggers the execution of *B*. After *B* has finished, its starting conditions are reevaluated since *A* might have executed and become *Finished* again in the meantime. Although *A* has not been reexecuted, the starting conditions are satisfied again because *A* is *still* in the *Finished* state. The problem here is that the navigation algorithm cannot distinguish between different executions of a task since navigation is state-based and there is only one *Finished* state.

4.4. Problems of merges in pipelines

Most workflow processes employ a control flow merge in some way. Independent of which strategy is chosen for handling execution opportunities, when merges are used with pipelining, our basic model (Section 2) exhibits two problems. These are the same problems as when deferring ex-

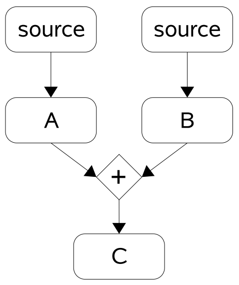

Figure 4. A simple process with a synchronous merge.

ecution opportunities (Section 4.3), but interestingly, they occur in different situations.

In the following examples we consider a process with a task *C* which has two predecessors *A* and *B* (Figure 4). *C* waits for both predecessors to finish before starting, i.e., the control flow is merged synchronously. *A* and *B* can execute multiple times because they are fed with data from the *source* tasks.

For the state ambiguity problem, we assume both tasks *A* and *B* are executed and become *Finished* whereupon *C* may be started. Then, after *C* has finished, task *A* is executed again. The finishing of *A* triggers 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. Thus, the synchronizing merge has become non-synchronizing after the first stream element has been processed. The problem is again related to the ambiguity of the *Finished* state. When the starting conditions are evaluated, it is indistinguishable whether the predecessors have been reexecuted or not since this was checked last time.

For the output overwriting problem, 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 which again leads to the problem of overwriting the output before it has been consumed by the successor.

5. Flow control markers

In this section we propose solutions to the problems of output overwriting and state ambiguity and show how to incorporate these solutions into the basic process model.

5.1. Flow control and data freshness

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. In addition to evaluating the activator and data-condition of a task, the navigation algorithm will enforce 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 to be buffered with other means.

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, we introduce a boolean *freshness marker* which is attached to each control flow dependency in a process instance. 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). A cleared marker means that the corresponding successor has learned about the new state.

5.2. Marker mechanism

When two tasks depend on each other, it means that they exchange information. The information consists of the state of the predecessor and the content of its output parameters. The state of a process instance is extended with a boolean *marker* for every pair of tasks with a control flow dependency. If the tasks in the pair have a cyclic dependency, then there are two markers, one for each direction of dependency. Every marker has a corresponding predecessor task and a successor task where the successor depends on the predecessor. Markers are used to control the information exchange between the two tasks. A marker has the following semantics. When it is set, the predecessor has produced new information. When it is cleared, the successor has consumed the information produced by the predecessor. Markers are initially cleared.

Figure 5 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 markers towards predecessors must be set and markers towards successors must be cleared in order for a task to be executed. This translates into requiring that there must be information available from the predecessor (freshness marker) 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 markers towards successors need to be inspected. If at least one such marker is set, 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 cleared, the starting conditions are checked.

When evaluating the activator, the state of the markers towards predecessors needs to be taken into account (freshness markers). A control flow dependency is not satisfied unless the corresponding marker is set. This prevents that

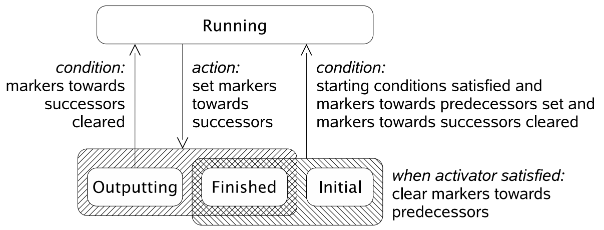
Running
condition: markers towards successors cleared
action: set markers towards successors
condition: starting conditions satisfied and markers towards predecessors set and markers towards successors cleared
Outputting
Finished
Initial
when activator satisfied: clear markers towards predecessors

Figure 5. Task state machine extended with conditions and actions to implement the marker mechanism.

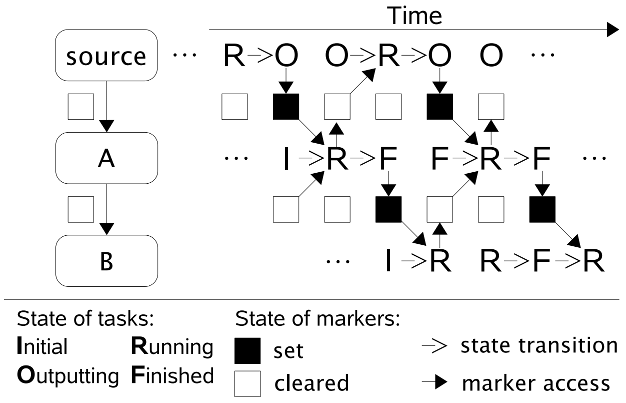

Figure 6. Example interaction of the marker mechanism with the state transitions of tasks.

the evaluation of the starting conditions uses stale information (state ambiguity).

If the activator is satisfied, all markers towards predecessors are cleared, indicating that the corresponding information has been consumed. Because these markers have been cleared, 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.

When a task reaches the *Finished* or *Outputting* state, every control flow dependency towards a successor is evaluated. If a dependency is satisfied, then the corresponding marker is set. 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 markers have been cleared. If the task is in the *Finished* state and it does not have any successors, the evaluation of its starting conditions is triggered immediately. Since there are no outgoing dependencies, the task can restart right away if information has become available from predecessors.

Figure 6 summarizes the interaction of the marker mechanism with the state transitions of a task using an example sequence of three tasks. The time in the diagram has been discretized for the sake of simplicity.

5.3. 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 sets the markers towards its successors. As long as these markers are set, the task does not enter the *Running* state again. Therefore, the task cannot execute and produce new output as long a the markers are set. When successors have consumed the output, they clear the corresponding markers. Thus, when all markers towards successors have been cleared, this means that the output has been consumed completely and the task can safely be reexecuted.

The state ambiguity problem is solved as follows. When a task produces output, it sets the markers towards its successors. Each marker is only set if the corresponding control flow dependency matches the state which the task just reached (e.g., *Finished* or *Outputting*).

When the activator of a task is evaluated, the markers towards predecessors are taken into account. If a marker is cleared, the corresponding control flow dependency is not satisfied. If the activator is satisfied (which means that all the corresponding markers are set), all markers towards predecessors are cleared, indicating that the corresponding information has been consumed. If the starting conditions were immediately revaluated, the activator would not be satisfied, even if all predecessors have the required state. This is because the markers are cleared. The activator will not be satisfied until all predecessors have produced fresh information by reaching a new state or the same state again and setting the corresponding marker.

6. Applications

6.1. High-volume account creation

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 pro-

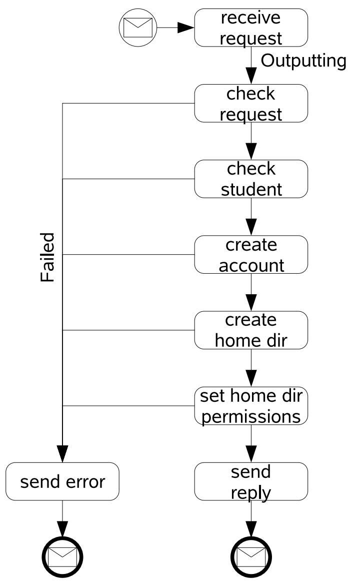

Figure 7. Account creation process.

cess should consist of a high number of tasks with even execution times.

Figure 7 represents the account creation process. Requests sent to the system contain information on the account to be created and the credentials of the requestor. 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 setup fails, an error is sent back to the originator of the request using the "send error" task.

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.

6.2. Shell script pipelines

Unix shells provide convenient ways of combining several commands into a pipeline, saving the user from storing intermediate results in temporary files. However, on a shell's one-dimensional command line, only linear pipelines can be built by connecting the input of programs with the output of others.

A workflow process provides an easy way to design such a pipeline as an arbitrary graph where the data can

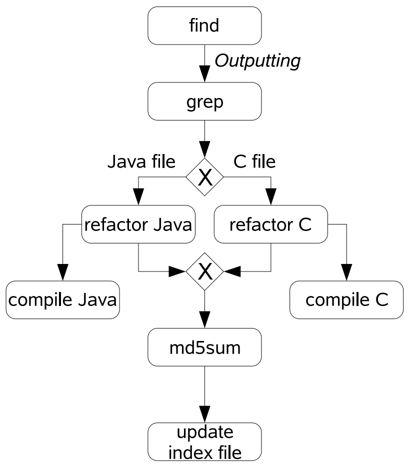

Figure 8. File refactoring process.

be duplicated to several tasks and routed according to data-based conditions. If the partial results of the commands are streamed through the process, pipelining can be used.

Figure 8 shows an example modeling the refactoring process over a collection of files. It first scans a directory containing source files written in C and Java. Every file is checked if it contains a certain text string. If this is the case, the code refactoring is performed on the file. Depending on the programming language, a different refactoring program is used. The refactored files are then recompiled by a suitable compiler. Also, refactored files need to have their MD5 sum recomputed and updated in an index file which can be done in parallel to the compiling. Since every file found by the "find" task is output to the next task (*Outputting* dependency), files are refactored while the "find" task is looking for more files to process.

This example shows how traditional pipelines of programs can be augmented with conditional branches. Also, programs which are not aware of streams, e.g. compilers, can easily be integrated with programs which produce a stream, e.g., `find` which outputs a stream of file names.

7. Related Work

There are a number of workflow engines that support streaming to some degree. OSIRIS-SE [1] aims at 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 [2]. The engine has then little control over these tasks that follow the multiple input/output model described in Section 4.1.

An example of a tool supporting pipelined execution over non-streaming Web and Grid services is Triana [11]. 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. Furthermore, Triana supports the continuous re-execution of a process such that tasks are invoked over each stream element.

Like Triana, SCIRun [15] is based on a dataflow network. Tasks communicate through FIFO-queues and can additionally produce multiple results during one execution similarly to the push mechanism described in Section 3.3.

YAWL [19, 18] is a workflow language and system based on the analysis of workflow patterns [20] 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 can be compared to a finite capacity Petri net [12] 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 [17]. However, it has also been shown that Petri nets have limitations when used to model complex workflows [19].

The newest version of UML [14] 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.

8. Conclusion

In this paper we have analyzed the problem of integrating data streaming services into a service composition tool based on a standard step-by-step process model such as those based on WS-BPEL or conventional workflow engines. We did this by using loops, extending the notion of task, and modifying the semantics of the language to accommodate pipelined processing of data within the business process. Once activated, tasks consuming stream sources run continuously and a push mechanism is provided to notify the workflow about the availability of new data elements. Downstream, we showed how the workflow execution can be pipelined over each stream element. Due to the heterogeneity of the tasks and their highly variable execution time, we compared different approaches to avoid pipeline collisions. To safely achieve flow control, a simple solution based on labeling task dependencies with markers was presented. Although this causes slow tasks to block the execution of upstream tasks, markers can be implemented using finite capacity queues, decoupling stream producers from consumers by evening out temporary differences in processing speed.

Acknowledgements

Part of this work is funded by the European projects: IST-FP6-004559 SODIUM (Service Oriented Development In a Unified fraMework) and IST-FP6-15964 AEOLUS (Algorithmic Principles for Building Efficient Overlay Computers).

References

[1] G. Brettlecker, H. Schuldt, and R. Schatz. Hyperdatabases for Peer–to–Peer Data Stream Processing. In *Proc. of ICWS Conf.*, pages 358–366, San Diego, CA, USA, 2004.

[2] G. Brettlecker, H. Schuldt, and H.-J. Schek. Towards reliable data stream processing with osiris-se. In *Proc. of BTW Conf.*, pages 405–414, Karlsruhe, Germany, 2005.

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

[4] 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*, pages 215–226, 2002.

[5] 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.

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

[7] 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.

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

[9] F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall, 1999.

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

[11] 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)*, pages 514–524. IEEE Computer Society, 2004.

[12] T. Murata. Petri Nets: Properties, Analysis and Applications. In *Proceedings of the IEEE*, volume 77, pages 541–580, April 1989.

[13] OASIS. *Web Services Business Process Execution Language Version 2.0 (working draft)*. OASIS, February 2005.

[14] Object Management Group. *Unified Modeling Language: Superstructure, version 2.0*, August 2005.

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

[16] C. Pautasso. *A Flexible System for Visual Service Composition*. PhD thesis, Diss. ETH Nr. 15608, July 2004.

[17] 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.

[18] 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.

[19] W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL: Yet another workflow language. *Information Systems*, 30:245–275, June 2005.

[20] 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.