SLA-Driven Business Process Management in SOA

Vinod Muthusamy and Hans-Arno Jacobsen
University of Toronto

Tony Chau, Allen Chan, and Phil Coulthard
IBM Canada*

Abstract

The management of non-functional goals, or Service Level Agreements (SLA), in the development of business processes in a Service Oriented Architecture (SOA) often requires much manual and error-prone effort by all parties throughout the entire lifecycle of the processes. The formal specification of SLAs into development tools can simplify some of this effort. In particular, the runtime provisioning and monitoring of processes can be achieved by an autonomic system that adapts to changing conditions to maintain the SLA's goals. SOA supports partitioning a system into services that are running in a distributed execution environment. When coupled with an associated cost model, a process can be both executed and monitored in an optimal manner, based on a declarative, user-specified optimality function.

1 Introduction

The Service Oriented Architecture (SOA) advocates building distributed applications by orchestrating reusable services using high level workflows or business processes. The complexity of developing and maintaining these processes is addressed by systematic development cycles that identify the roles and skills required by participants at each stage of development. To assist this development process, sophisticated tools have been developed, such as IBM's WebSphere suite of SOA products. However, the development, administration and maintenance of a business process still requires much manual effort that can be automated. In particular, non-functional goals, often expressed as Service Level Agreements (SLA), that are specified during the modelling stage need to be manually considered at each stage of the development process. For example, the requirement that process instances complete within some specified time may influence decisions in the development, deployment, resource provisioning, and monitoring of the process.

This paper presents a vision to achieve end-to-end SLA management by facilitating the various stages of business process development. By formally encoding the SLA goals within the development tools, each stage of the development cycle can be simplified to varying degrees. Specifically, the autonomic runtime execution and monitoring stages are addressed in this paper. We argue for a distributed architecture for the execution of business processes, and develop a model to control the provisioning of business processes in this architecture based on high-level goals that can be specified independently of the processes' implementation details.

An important phase of the business process development cycle is the runtime monitoring of processes to observe various metrics, such as the average execution time or throughput of a process. Monitoring is often achieved by collecting all instrumentation data in the system in a centralized database and querying this data. This can be expensive computationally and in terms of storage and network resource requirements. An alternative method is to automatically infer the measurements of interest based on the formally specified SLAs, and only monitor those metrics. This paper takes this approach and goes a step further by modelling the monitoring of a process as a process itself and reusing our autonomic execution architec-

*The views and opinions expressed in this paper solely reflect the personal views of the authors and do not necessarily represent the views, positions, strategies or opinions of IBM.

Copyright © 2009 Middleware Systems Research Group, University of Toronto and IBM Canada Ltd. Permission to copy is hereby granted provided the original copyright notice is reproduced in copies made.

Paper appears in *CASCON 2009*, Richmond Hill, Ontario, Canada, November, 2009.

ture to also perform monitoring.

The main contributions of this paper are (i) a vision of how formally specified SLAs can simplify the end-to-end SOA development cycle, (ii) the design and development of a cost model and distributed architecture to execute business processes, and to autonomically provision processes based on high-level goals, (iii) the systematic mapping of SLAs to a process that realizes the monitoring requirements inferred by the SLA, and (iv) an evaluation of the distributed execution engine.

Section 2 begins with an outline of the SOA development cycle, and points out the possible benefits of formally specified SLAs in each stage. In Section 3 the key components of an SLA language are presented, along with a short introduction to the WSLA [7] language, and some extensions to WSLA required for this work. Section 4 motivates the need for distributed process execution and describes a cost model and architecture that achieves autonomic process execution. Section 5 illustrates how the monitoring of an SLA can be modelled as a process and optimized by the autonomic execution engine above, and Section 6 follows with an evaluation of this engine. Section 7 puts this paper in context of related work, and Section 8 presents some concluding remarks and future research directions.

2 SLAs and the SOA Development Cycle

This section outlines a typical SOA development cycle, and follows with a vision of the benefits to this process of managing SLAs within the development tools.

2.1 SOA Development Cycle

The SOA development cycle illustrated in Figure 1 consists of modelling, development, execution, and monitoring stages. Each stage differs in the level of abstraction and is performed by the indicated roles, each of whom have varying expertise and concerns.

Consider the simple loan application business process fragment in Figure 2. The process first calls an external credit check service to

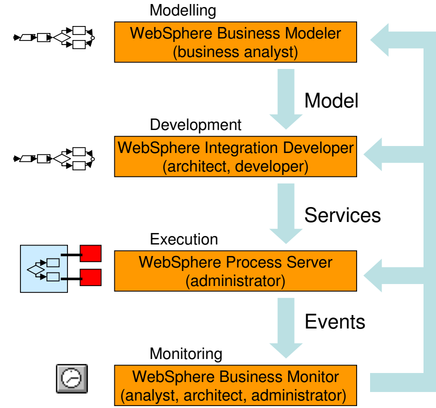


Figure 1: SOA development cycle

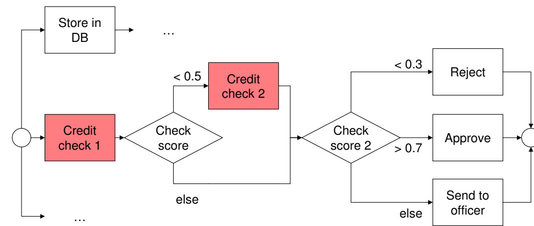


Figure 2: Loan application process

determine the applicant's credit rating, with a second more refined credit check service used in case of a low rating. This rating is used to determine if the application should be approved, denied, or processed further by an officer.

In the modelling stage of the development cycle, the business analyst would define the above process, abstracting from technology, infrastructure, and implementation details. The result of the modelling stage is an abstract model of the process represented in BPEL or a proprietary process language. This representation is imported into the development stage, where architects and developers break the model into development artifacts such as services along with their interfaces, and implement the required business logic. The result of this stage is a set of deployable components that are represented in a standard specification such as the Service Component Architecture (SCA) [12] model in WebSphere Integration Developer (WID). These services are deployed in a runtime environment that is managed by an administrator who is responsible for ensuring resource allocations and physical resource

provisioning sufficient for the goals of the deployed services and processes. Often it is desirable to monitor the execution of the processes by tracking metrics on the state of the executing system. These metrics are computed based on observations in the runtime system and can be captured using standard event frameworks such as the Common Event Infrastructure. The metrics gathered can be aggregated and presented to the stakeholders in the preceding development stages. For example, the business analyst may be interested in high level metrics such as the number of times the second credit check is required. On the other hand, the system architect may be interested in lower level metrics such as the processing delays of the individual credit checking services, while the administrator would be concerned with system performance bottlenecks such as network congestion or processor utilization.

In the modelling stage, high level, declarative goals are specified by the analyst, such as the throughput requirements of the process, or the cost constraints on the process. These goals are formalized into Service Level Agreements that specify a contract between the service provider and consumer. SLAs can be represented at different levels of abstractions, and may simply be a document at the modelling stage. These SLA documents are passed down the chain of the development process, with each stakeholder being responsible for ensuring conformance to the SLAs. The architects and developers interpret and ensure that the services developed conform to the SLAs. During execution, SLA conformance is often achieved by over-provisioning resources and manually tuning the system. Finally, monitoring subsystems are instantiated to verify that the SLA goals are met, and that violations are reported to the appropriate parties. These violations are manually addressed by changes to the process, redevelopment of services, or provisioning of resources.

2.2 Integration of SLAs

If SLAs are integrated into the tools and translated into execution and monitoring models, violations can be tracked more quickly and resources can be provisioned on demand in response to or in anticipation of violations. For example, SLAs at the modelling stage can be mapped to lower level requirements on the services developed and resources provisioned, and translated to metrics that need to be monitored to observe SLAs violations. Furthermore, adaptations on the process itself or its resource provisioning can be performed automatically at runtime to maintain the SLA goals.

Several runtime adaptations can be performed to maintain SLAs. *Dynamic service selection* can occur whereby the most appropriate services are chosen from a catalogue. For example, a fast or cheap credit check service can be used based on the SLA requirements. As elaborated in Section 5, *monitoring* can be optimized by only observing those metrics that are relevant to the SLA. As outline in Section 4.3, the *distributed execution* of business processes becomes feasible by automatically assigning portions of a process to strategic locations in the system. Furthermore, dynamic *resource allocation* can take place to ensure the process has sufficient resources (CPU, bandwidth, etc.) to maintain the SLA despite changes in the process load. Finally, the Enterprise Service Bus (ESB) [11] that underlies the SOA can be reconfigured to satisfy the SLA. It is important to emphasize that the encoding of the SLAs into the SOA tools in a machine understandable format makes it possible for these runtime adaptations to take place dynamically and without human intervention.

Incorporating SLAs into SOA development tools *simplifies* the specification of SLAs since the analyst can declaratively specify high level business process goals without detailed knowledge of the underlying implementation technologies. Additional *flexibility* is achieved by allowing the developers and administrators to make design decisions without having to be as concerned about SLA violations since the tools can perform some of these tasks. As well, the analyst can change the SLAs and be confident that these revisions will be propagated and enforced throughout the development stages.

3 SLA Language

This section identifies the key components of an SLA language needed to specify goals on busi-

Component	Description
Purpose	High level statement
Parties	Service consumers, providers, etc.
Validity period	When SLA is active or expires
Scope/exclusions	Conditions under which to evaluate SLA
Metrics	Instrumented or computed measurements
SLOs	Commitment to a certain behaviour
Penalties	Cost of violating SLO

Table 1: SLA components

```
<SLA name="LoanAppServiceAgreement1" ...>
  <Parties>
    <ServiceProvider name="provider">
      <Contact>
        <POBox>P.O.Box 218</POBox>
        <City>Yorktown, NY 10598, USA</City>
      </Contact>
    </ServiceProvider>

    <ServiceCustomer name="customer">
      <Contact> ... </Contact>
      <Action> ... </Action>
    </ServiceCustomer>

    <SupportingParty name="ms"
     role="MeasurementServiceProvider" sponsor="provider">
      <Contact> ... </Contact>
      <Action xsi:type="WSDLSOAPActionDescriptionType"
       name="Notification" partyName="ms">
        <WSDLFile>Notification.wsdl</WSDLFile>
        <SOAPBindingName>
          SOAPNotificationBinding
        </SOAPBindingName>
        <SOAPOperationName>Notify</SOAPOperationName>
      </Action>
    </SupportingParty>
  </Parties>

  <ServiceDefinition name="LoanAppService">
    <Operation> ... </Operation>
  </ServiceDefinition>

  <Obligations> ... </Obligations>
</SLA>
```

Figure 3: WSLA example

ness processes, followed by a brief introduction to the WSLA language, and a description of our extensions to this language.

3.1 Language Components

Several SLA languages have been proposed with varying expressiveness. This section describes some of the common components required in an SLA, as outlined in Table 1.

The *purpose* is a high-level description of the SLA, and the *parties* specify contractual partners in the SLA such as the service consumers,

```
<Operation name="WSDLSOAPSubmitLoan" ...>
  <SLAParameter name="CreditCheckCostPerDay" type="float"
   unit="dollars">
    <Metric>creditCheckCostPerDay</Metric>
  </SLAParameter>

  <Metric name="creditCheckCostPerDay" type="float"
   unit="dollars">
    <Source>provider</Source>
    <Function xsi:type="wsla:Plus" resultType="float">
      <Operand>
        <Metric>creditCheck1CostPerDay</Metric>
      </Operand>
      <Operand>
        <Metric>creditCheck2CostPerDay</Metric>
      </Operand>
    </Function>
  </Metric>

  <Metric name="creditCheck1CostPerDay" type="float"
   unit="dollars">
    <Source>provider</Source>
    <Function xsi:type="wsla:Multiply" resultType="float">
      <Operand>
        <Metric>creditCheck1InvocationsPerDay</Metric>
      </Operand>
      <Operand>
        <LongScalar>0.01</LongScalar>
      </Operand>
    </Function>
  </Metric>

  <Metric name="creditCheck2CostPerDay"type="float"
   unit="dollars"> ...  </Metric>

  <Metric name="creditCheck1InvocationsPerDay"
   type="integer" unit="invocations">
    <Source>provider</Source>
    <Function xsi:type="wsla:SumPerDay"
     resultType="integer">
      <Metric>creditCheck1Invocation</Metric>
    </Function>
  </Metric>

  <Metric name="creditCheck1Invocation"
   type="integer" unit="invocation">
    <Source>ms</Source>
    <MeasurementDirective xsi:type="wsla:Invocation"
     resultType="integer">
      <MeasurementURI>
        http://ms.com/invocation
      </MeasurementURI>
    </MeasurementDirective>
  </Metric>

  <WSDLFile>LoanAppService.wsdl</WSDLFile>
  <SOAPBindingName>SOAPLoanAppBinding</SOAPBindingName>
  <SOAPOperationName>submitLoan</SOAPOperationName>
</Operation>
```

Figure 4: Example: metrics and parameters

providers or third parties. The *validity period* defines the time frame during which the SLA is to be honoured. The *scope* and *exclusions* describe the conditions under which the SLA should be applied. For example, an SLA may only apply to gold customers or during weekends. *Metrics* are observable properties of the

```
<Obligations>
  <ServiceLevelObjective name="costSLO"
   serviceObject="WSDLSOAPSubmitLoan">
    <Obliged>provider</Obliged>
    <Validity>
      <StartDate>2008-04-01:1400</StartDate>
      <EndDate>2009-04-01:1400</EndDate>
    </Validity>
    <Expression>
      <Predicate xsi:type="wsla:Less">
        <SLAParameter>
          CreditCheckCostPerDay
        </SLAParameter>
        <Value>100</Value>
      </Predicate>
    </Expression>
    <EvaluationEvent>NewValue</EvaluationEvent>
  </ServiceLevelObjective>

  <ActionGuarantee name="CostGuarantee">
    <Obliged>provider</Obliged>
    <Expression>
      <Predicate xsi:type="wsla:Violation">
        <ServiceLevelObjective>
          costSLO
        </ServiceLevelObjective>
      </Predicate>
    </Expression>
    <EvaluationEvent>NewValue</EvaluationEvent>

    <QualifiedAction>
      <Party>ms</Party>
      <Action actionName="CostNotification"
       xsi:type="Notification">
        <NotificationType>Violation</NotificationType>
        <CausingGuarantee>costSLO</CausingGuarantee>
        <SLAParameter>
          CreditCheckCostPerDay
        </SLAParameter>
      </Action>
    </QualifiedAction>
    <ExecutionModality>OnceADay</ExecutionModality>
  </ActionGuarantee>
</Obligations>
```

Figure 5: Example: SLOs and violation actions

system or process. Atomic metrics are those that are directly measured and include processor utilization and number of process invocations. Composite metrics are derived from atomic or other composite metrics and include throughput (which is computed based on the number of invocations over time), and response time (which is the difference between the invocation and completion of a service). Composite metrics can be represented as functions of other metrics, such as the maximum, or average value of these metrics. *Service Level Objectives* express a commitment to a certain behaviour by one or more of the parties. It is a logical expression based on the defined metrics, and may include when to evaluate the metrics. For example, an SLO may state that the service provider must ensure that the system is available for at least 99% of requests for each calendar day. A final component in an SLA is the *penalty* of violating the SLOs. These costs may be fixed or a function of the degree of the violation. The costs may also be payable to other parties or simply to be an indicator to a party. For example, the cost of taking too long to process a loan application may be a refund to the applicant. In this work, the most important components are the metrics and SLOs. In the future, we plan to consider the severity of the penalties to prioritize conflicting SLA goals.

3.2 WSLA

Most of the components described above map directly to the WSLA [7] language that we adopt and extend in this work. Rather than repeat the formal description of WSLA in [7], we present an example SLA expressed in WSLA in Figures 3, 4 and 5. The SLA corresponds to the simple loan application business process in Figure 2. As shown in Figure 3, the three main sections of an SLA specified in WSLA are the *parties* involved in the SLA, including the service consumer, provider and any third parties; the *service definition* which specify the services the SLA applies to, and the metrics of interest on these services; and the *obligations* which outline the guarantees expected of the service including the action to take when an obligation is violated.

Figure 4 defines metrics to compute the daily cost of invoking the external credit check services in the process in Figure 2. Notice that composite metrics use custom `Function` types that operate on other metrics, and atomic metrics specify a custom `MeasurementDirective` that describes how the metric is to be retrieved. Also, any metrics that are to be referred to in an SLO obligation (see Figure 5) must be put in an `SLAParameter` element.

The obligations section of the SLA is presented in Figure 5. Here, an SLO is defined that requires the service provider to ensure that the daily cost of invoking the external credit checking services does not exceed $100, and to notify the third party measurement service if this SLO is violated.

3.3 WSLA Extensions

As stated earlier in Section 2, we wish to automatically monitor the SLAs. This raises the questions of how this monitoring should be done. For example, should the monitoring subsystem minimize its impact on the system it is monitoring or should it attempt to be responsive in quickly reporting the metrics it is observing. Such goals can be specified by defining another SLA on the SLA, but to avoid issues with infinite recursion of SLAs, we extend the WSLA language to express an SLA *optimization criteria*.

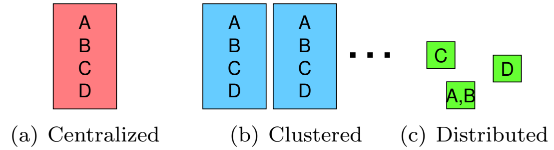
For example, given the SLA in Figure 3 that ensures that the cost of the services used by the load application process is within a threshold, we can specify an optimization criteria on the SLA that requires that the monitoring of this SLA incurs as little bandwidth as possible. Alternatively, it also possible to select an optimization criteria that minimizes the latency of this monitoring so that violations are reported as quickly as possible. These optimization criteria can be associated with the SLO, the `SLAParameter`, or `Metric`, with each criteria being inherited down a hierarchy from SLOs, to their constituent `SLAParameters`, through to their associated `Metrics`. For example, an optimization criteria associated with an SLO is implicitly applied to the `SLAParameters` used by the SLO, and the metrics that compose the `SLAParameter`. The optimization criteria is specified in an `Optimize` element within the `ServiceLevelObjective`, `SLAParameter`, or `Metric` elements. For example, the fragment in Figure 6 indicates that the monitoring of the SLO should minimize bandwidth.

Another extension we make to WSLA is the addition of a new `MeasurementDirective` type. `MeasurementDirectives` are an abstract type in WSLA used to define various methods to retrieve the measurements used in a metric. We define a `wsla:CostComponent` type to access the components in the distributed execution engine's cost model described in Section 4.2. An SLO based on metrics of this type is implicitly used to optimize the process the SLO refers to. Presently, we support a set of predefined optimization criteria that map to the cost model presented in Section 4.2 and we plan to generalize this in future work.

```
<Obligations>
  <ServiceLevelObjective name="costSLO"
   serviceObject="WSDLSOAPSubmitLoan">
    <Obliged> ... </Obliged>
    <Validity> ... </Validity>
    <Expression> ... </Expression>
    <EvaluationEvent> ... </EvaluationEvent>
    <Optimize>Bandwidth</Optimize> <!-- WSLA extension -->
  </ServiceLevelObjective>

  <ActionGuarantee name="CostGuarantee">
    ...
  </ActionGuarantee>
</Obligations>
```

Figure 6: WSLA extension: optim. criteria



(a) Centralized (b) Clustered (c) Distributed

Figure 7: Execution engine architectures

4 System Architecture

Business process execution engines are typically centralized systems in which one node is provisioned to execute and manage all instances of one or more business processes. To address scalability, the centralized engine can be replicated and the process instances balanced among the replicas. In this work, we take a fundamentally different architectural approach whereby even individual process instances are executed in a distributed manner. The benefits of this architecture include scalability, in-network processing, and fine-grained use of IT resources. We further describe and compare the various business process execution architectures below.

4.1 Execution Architectures

Centralized The simplest business process engine consists of a single execution engine as shown in Figure 7(a). This centralized engine is responsible for executing and managing all concurrent instances of the processes deployed on it. The advantage of such an architecture is

its simplicity in terms of deployment and management. However, as the resources in such an architecture are fixed, the system may not scale with the complexity of processes and the number of process instances. Also, as the single execution engine represents a single point of failure and may not be appropriate for the execution of mission critical processes. Furthermore, inter-organization business processes may have no obvious choice for a central coordinator.

Clustered To address scalability and fault-tolerant, a cluster of execution engines can be deployed. In this architecture, illustrated in Figure 7(b), each engine in the cluster is essentially a replica of the others, and can execute a complete business process. A call to a business process P is first sent to a load balancing component (not shown in the figure), which forwards the call to one of the engines E in the cluster, based on some criteria such as ensuring a balance of load across the cluster. At this point, engine E creates an instance of process P and is responsible for executing the instance until completion. Some systems support the ability to add and remove engines to the cluster as the load varies. A clustered execution architecture can be scalable and does not suffer from a single point of failure. However, process *instances* are still executed in a centralized manner, and control and data is still concentrated in the cluster. Consider the case of a data-intensive process such as a scientific workflow that transfers and operates on large volumes of data. In a clustered architecture, the data needs to be transfered to the cluster before it can be operated on by the process. In a more flexible deployment it would be possible to move the portions of the process that operate on the data closer to the data source thereby reducing the time and network costs incurred in having to transfer the data.

Distributed This paper proposes an execution engine in which processes themselves can be distributed. As shown in Figure 7(c), a process is first decomposed into tasks. In a BPEL process, the tasks can be the individual BPEL activities. These tasks are then assigned to various execution engines in the system. To emphasize the fact that these execution engines can be light-weight as they only execute fine-grained tasks, as opposed to complete processes, we refer to the entity that execute tasks as an *agent*. A key benefit of such an architecture is the ability to deploy *portions* or processes close to the data they operate on, thereby minimizing bandwidth and latency costs of a process. For example, for data intensive business processes, such as rendering farms or large simulations, it would be possible to deploy only those portions of the process that require access to large data sets close to their respective data sources. Different parts of the process that operate on different data sets can be independently deployed near their respective data sources. This is not possible in a clustered architecture since the entire process instance must be executed by a single engine.

The benefits of the agent-based execution engine architecture are only achieved if the agents are deployed in a strategic manner. This can be a labour intensive procedure that requires knowledge of the system resources, and process characteristics. It may even be a futile exercise if either of these variables changes frequently. It is desirable for the system itself to determine an optimal placement of agents. To achieve this, we develop a cost model below to model the cost of a particular placement of agents. This model is used to compare different placement possibilities.

Component	Notation
Distribution cost	C_{dist} (distribution overhead)
Message rate	C_{d1}
Message size	C_{d2}
Message latency	C_{d3}
Engine cost	C_{eng} (execution overhead)
Load	C_{e1}
Resources	C_{e2}
Task complexity	C_{e3}
Service cost	C_{serv} (service overhead)
Service latency	C_{s1}
Service execution	C_{s2}
Marshalling	C_{s3}

Table 2: Cost model components

4.2 Cost Model

The cost model is a framework that consists of various factors that can influence the agent placement decisions. Some cost factors are shown in Table 2 grouped into *cost components*.

Criteria	Cost function mapping
3s response time	$C_{d1} + C_{d3} + C_{e3} + C_{serv} < 3$
Optimize bandwidth	$min(C_{d2})$

Table 3: Examples of cost functions

The first component is the *distribution cost* which represents the overhead of distributing a process into small, fine-grained agents. This overhead can be expressed in terms of the bandwidth or latency of the inter-agent communication depending on the desired goal. .

Another important cost component captures the resource usage of an agent on the engine it is executing on. Factors here include the number of concurrent instances an agent is executing, the resource utilization (in terms of processor or memory) of an agent, and the complexity of the task the agent is executing.

The third cost component in Table 2 is the *service cost* which represents the cost of calling external services. This includes the time to call the service (which is a function of the network conditions between the agent and service), and the execution time of the service (which depends on the particular service provider used to execute the desired service).

A *cost function* based on various cost components can flexibly express different goals. The cost function specifies that an arbitrary weighting of the various cost components either meet a *threshold* or should be *minimized*. In the former case, the process is adapted only when the threshold is violated, while in the latter, process adaptation occurs whenever a more optimal placement is found. For example, Table 3 shows cost functions that constrain process response times to three seconds, and that minimize the network overhead of a process.

4.3 Distributed Execution

As discussed earlier, we take a distributed approach to the execution of business processes, whereby individual tasks in a process are executed by autonomous agents which collaborate to execute the original process. The agents execute within a distributed execution engine whose architecture is shown in Figure 8.

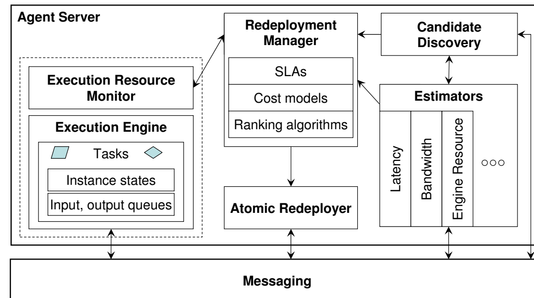


Figure 8: Distributed execution engine

In the spirit of the distributed nature of the system, each agent is autonomous in deciding which engine it should execute on and when it should move to another engine. These decisions are based on the cost function associated with the process, which is known to each agent in the process. Notably, there is no centralized component that is used to gather statistics, or to make agent placement decisions.

The distributed execution engine shown in Figure 8 consists of a core Execution Engine that provides support services for agents to collaborate among one another to execute a particular business process. A Candidate Discovery component is used to find other execution engines in the system. In the current implementation, the discovery component returns all immediate neighbours of an engine and uses a random walk to find a random set of distant engines. The discovered candidates are periodically probed by the Estimator components to gather various statistics. The Redeployment Manager computes the cost function for each agent executing in the engine, and determines if a more optimal placement of the agent is available among the known candidate engines. Finally, agents that are to be moved are redeployed using the Atomic Redeployer component which is responsible for ensuring that the movement of the agent does not affect the execution of the process. Briefly, the redeployer pauses the triggering of new instances of the agent, transfers the agent state to the new engine, rebinds the agent to its successors and predecessors in the process, and resumes the execution of the agent.

4.3.1 Redeployment Manager

The Redeployment Manager maintains for each agent A_i the agent server is currently hosting, the cost function $F(A_i)$ associated with the agent, a running average of the cost $C(A_i, S_j)$ imposed by the agent were it hosted by agent server S_j, and the agent servers where agent A_i's predecessors and successors are hosted. For convenience, the cost of deploying A_i at the current server is denoted as $C(A_i)$. The cost $C(A_i)$ has two different interpretations depending on the type of cost function. For threshold functions, $C(A_i)$ is the *accumulated* cost by all agents from the beginning of the process to the current agent, whereas for minimum or maximum cost functions, $C(A_i)$ is the *local* cost of the agent. An example should make the reason for the difference clear. Consider a cost function to minimize the message latency of a process: $min(C_{d3})$. In this case, the local cost of an agent is the latency of communicating with its predecessors and successors. To minimize the overall latency, each agent should attempt to minimize its local latency cost. On the other hand, for a cost function that requires the message latency to stay below a threshold, such as $C_{d3} < 10$, it is necessary to keep track of how much each agent contributes to the overall latency of the process. The local latency cost of each agent must be accumulated as the process flow executes. To achieve this, for agents associated with threshold cost functions, messages between agents are annotated with the accumulated cost.

The running average of the cost $C(A_i, S_j)$ of an agent is computed and maintained by the Redeployment Manager based on information from various Estimators or the Execution Resource Monitor. For example, consider an agent A hosted on server S_j, with predecessor agent A_p and successor agent A_s, hosted on servers $S(A_p)$ and $S(A_s)$, respectively. Agent A's local message latency cost is

$$Latency(S(A_p), S_j) + Latency(S_j, S(A_s)),$$

where $Latency(S_p, S_q)$ is the latency between agent servers S_p and S_q as determined by the Latency Estimator.

The Redeployment Manager recomputes the costs $C(A_i, S_j)$ when one of two conditions occurs. When the agent A_i executes (including when it sends and receives messages with its successors or predecessors), the cost for the current server $C(A_i, S_{current})$ is updated. Likewise, when an estimator updates a metric that is included in the cost function associated with A_i, the cost of hosting the agent at the candidate server whose metric was just updated is recomputed. To facilitate the latter case, each estimator given a list of agents which are relevant to the metric the estimator is computing. Therefore, the estimator will only notify the Redeployment Manager when necessary.

An update of the cost $C(A_i, S_j)$ may reveal a better placement for agent A_i. Every update to the cost of an agent initiates a call to the $CheckDeployment(A_i)$ function to find a more optimal deployment. The $CheckDeployment(A_i)$ algorithm differs based on the cost function associated with agent A_i.

If the cost function is to be minimized, the algorithm finds the server $S_{min} \in S$ such that $C(A_i, S_i)$ is minimized across all $S_i \in S$ where S is the set of known candidate agent servers. Agent A_i is then moved to agent server S_{min}. To avoid frequent redeployment, an agent is redeployed only if the improvement in the cost exceeds a given threshold $T_{benefit}$ and if the agent has not been redeployed for some time duration $T_{duration}$. The values $T_{benefit}$ and $T_{duration}$ have system wide defaults that may be overridden by each cost function.

If the cost function associated with agent A_i is a threshold function, a check is made to see if the accumulated cost $C(A_i)$ exceeds the threshold. If the cost is still within the threshold, nothing further is done. Otherwise, the $CheckDeployment()$ function finds the agent server S_{min} that results in $\min_{S_i \in S} C(A_i, S_i)$, and redeploys agent A_i to agent server S_{min}. Now it may be that $C(A_i, S_{min})$ still exceeds the cost function threshold, in which case a message is sent to the predecessor servers of agent A_i to force them to redeploy. Notice that these predecessors would not have normally chosen to redeploy because their accumulated cost is still within the threshold. This "back pressure" by agents to force a redeployment of their predecessors will occur repeatedly as long as the optimal placement of the agent is not sufficient to satisfy the cost function threshold.

4.3.2 Estimators

Estimators compute metrics used to rank possible placements of agents and determine the optimal placement. To avoid unnecessary work, an estimator is enabled only if there is a locally hosted agent whose cost function depends on the metric computed by the estimator. For example, if there is an agent whose cost function is to minimize message latency, the latency estimator would be enabled but not the bandwidth estimator. Each estimator is also provided with additional information necessary to perform the estimations. For example, the latency estimator is given a set of the predecessor and successor servers S_N associated with relevant agents. The redeployment manager then estimates the latency between nodes in S_N with the nodes S_C discovered by the Candidate Discovery component. The Service Latency Estimator, on the other hand, is provided a list of services invoked by relevant agents and the estimator computes the time to invoke the services from each node in S_C.

4.3.3 Atomic Redeployer

The actual movement of an agent A_i from agent server S_1 to agent server S_2 as determined by the Redeployment Manager is carried out by the Atomic Redeployer. The challenge is to move an agent without disrupting the process and to ensure that failures during movement do not leave the system in an inconsistent state. The movement is modelled as a transaction consisting of a single $move(A_i, S_1, S_2)$ operation. If the transaction aborts—perhaps because S_2 refused to accept agent A_i—the agent must remain at server S_1, otherwise if the transaction commits, the agent must be instantiated at server S_2 and deallocated from server S_1. In either case, the predecessors and successors of agent A_i should be unaware of the movement, no messages must be lost, and each message should be delivered to either the agent instance at S_1 or at S_2 but not both. These requirements and other have been formalized in detail, and the algorithms to achieve atomic movement satisfying these requirements have been developed, their correctness proven, and their performance quantified [6].

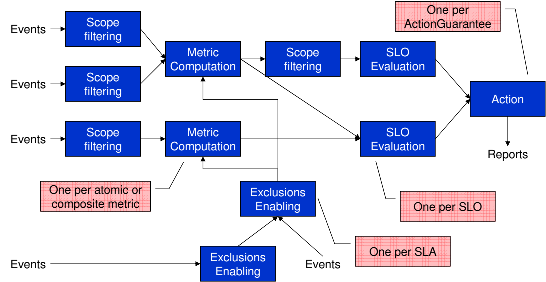


Figure 9: Generated monitoring agents

5 Monitoring Case Study

This section illustrates an end-to-end example that demonstrates the methodology developed in this paper. In this case study we express system monitoring as a process.

We show how an SLA is automatically mapped to a set of monitoring agents that collaborate to observe violations of the SLAs, and how the SLA's associated optimization criteria is mapped to a cost function in the cost model. We also present an example of how these monitoring agents may be deployed on a distributed set of execution engines, and show how changes in the execution state of the process or the characteristics of the executing environment are reflected in the cost model, in turn initiating reconfiguration of the monitoring agents to comply with the optimization criteria.

SLAs are systematically mapped to a set of monitoring agents as shown Figure 9. Each `Metric` and `SLAParameter` is associated with an agent, and there is one agent responsible for each SLO and `ActionGuarantee`. For the SLA in Figure 3, the generated agents and their relationships are shown in Figure 10. As noted in Table 3, the bandwidth optimization criteria specified in Figure 6 is mapped to the cost function $cost(agent) = C_{d2} = C_{messagesize}$. In the current system, this optimization criteria is updated every time an agent is invoked, and the costs averaged over a sliding window. We plan in the future to extend this model to allow arbitrary function evaluation along the lines of what is expressible in WSLA.

For the example SLA in Figure 3, the generated agents shown in Figure 10 are deployed into the distributed execution environment. A possible deployment of some of the agents is

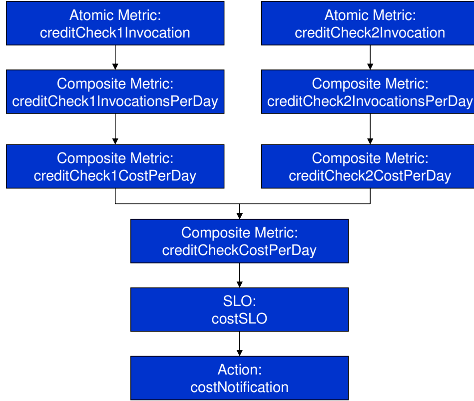


Figure 10: Agents generated from SLA in Fig. 3

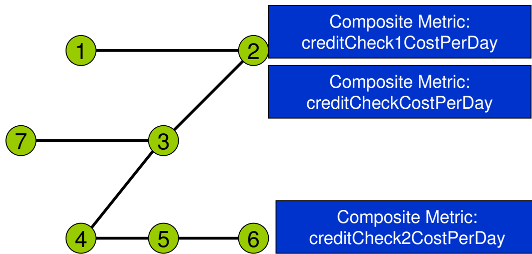


Figure 11: Initial deployment of agents

shown in Figure 11, where the graph represents the various execution engines deployed on the ESB. Suppose that the number of calls to the second credit check service is increases greatly. This will results in more cost updates being sent from the $creditCheck2CostPerDay$ agent to the $creditCheckCostPerDay$ agent in Figure 11. In response to this, the ranking algorithms' estimate of the cost of the $creditCheckCostPerDay$ agent while deployed at Node 2 is greater than if it were deployed at Node 3, and will move this agent to Node 3.

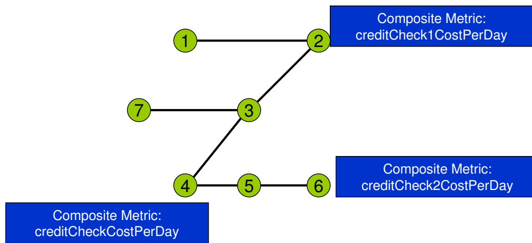


Figure 12: Reconfigured deployment of agents

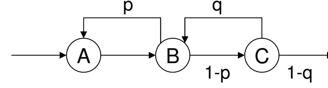


Figure 13: Business process with loops

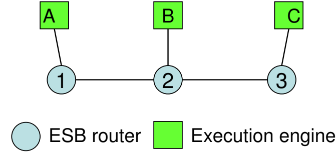


Figure 14: Experimental topology

The ranking algorithms may then again determine that the cost at Node 4 is less than that at Node 3, and move the agent again, resulting in the deployment shown in Figure 12.

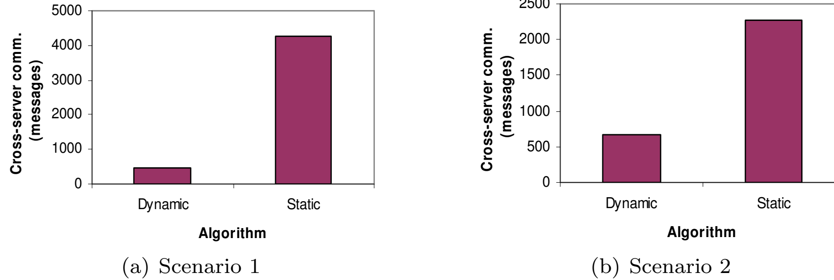
In this section we have shown how a high level SLA and optimization criteria can be automatically mapped to a set of monitoring agents, whose deployment will automatically reconfigure in response to changing business or infrastructure conditions to meet the stated optimization criteria.

Notice that we applied the distributed execution engine to a monitoring "process". The same principles are used to optimize the deployment of a business process according to some SLA goals.

6 Evaluation

In this experiment, we use the business process shown in Figure 13 deployed to a set of execution engines in Figure 14. The process is designed to model a business process with time varying branch probabilities. Also, while the process only consists of three tasks, the looping constructs in the process results in many tasks being executed in the course of a process instance. The agents associated with the tasks in the process are initially deployed as shown in Figure 14.

In this process, the branch probabilities of tasks B and C are varied and the reactions of the redeployment algorithms are observed. In the experiment, the SLO associated with the process seeks to minimize the bandwidth used by the process (see Table 3). Consequently, the



(a) Scenario 1

(b) Scenario 2

Figure 15: Message cost for process in Fig. 13

metric we observe is the number of messages sent between the execution engines. Note that messages between agents deployed on the same engine are not counted in this measurement.

The metric to minimize bandwidth results in all agents being deployed on the same execution engine regardless of the workload. To make the experiment more interesting, we fix the agents associated with tasks A and C to their initially deployed engines in Figure 14, and allow agent B to move freely to any engine. This represents the case where certain tasks must be executed on engines owned by a particular department for administrative or security reasons.

Figure 15(a) shows the results of an experiment in which the branch probabilities of the process are fixed at $p = 0.9$ and $q = 0.1$ and 100 instances of the process in Figure 13 are invoked. Note that this workload is biased toward the loop involving tasks A and B. We observe that the system reconfigures itself from the initial deployment in Figure 14 to one where agent B runs on the same execution engine as agent A. In Figure 15(a), this results in the dynamic algorithm having about 10% of the message cost of the static algorithm in which reconfiguration is disabled and the agents remain in the initial deployment in Figure 14. The point here is that it is not necessary to manually deploy agents in a strategic manner — a possibly complex task — but to allow the system to configure itself.

In another experiment, the branch probabilities are varied, starting out with $p = 0.9$ and $q = 0.1$ for the first half of the experiment, and changing to $p = 0.1$ and $q = 0.9$ for the second half. Therefore, the workload initially results in a lot of communication between tasks A and B, and then becomes biased towards tasks B and C. This time, the deployment starts with agent B at the same engine as agent A, which is the optimal placement for the initial branch probabilities in this experiment. Again, we observe that the dynamic algorithms keeps agent B in the initial optimal engine, and only when the branch probabilities change, the system moves agent B first to the middle execution engine and then to the one with agent C. The results in Figure 15(b) confirm that the dynamic reconfiguration algorithms adapt to the changing workload to achieve the desired goal with less than 30% of the message cost of the static case. A notable point about this experiment is that because the conditions of the system change over time, there is no optimal static deployment. Dynamic reconfiguration is required to achieve the best results.

7 Related Work

Related work falls into the categories of distributed workflow processing, business process execution, and data stream management. We discuss these in turn, emphasizing how our envisioned approach extends and differs from them. We also briefly review the Business Process Execution Language and summarize the state-of-the-art in languages for specifying SLAs. Both are essential to our approach.

Distributed workflow and business process execution: Distributed workflow processing has been studied in the 1990s to address scalability, fault resilience, and enterprise-wide workflow management [2, 17, 10]. The term *business process* today is often used in lieu of the term *workflow*. In this paper we do not

distinguish between the two and use them synonymously. Alonso *et al.* [2] present a detailed design of a distributed workflow management system. The work bares similarity with our envisioned approach in that a business process is fully distributed among a set of computing nodes. However, the distribution architectures differ fundamentally. In our approach, a messaging substrate is built to naturally enable task decoupling, dynamic reconfiguration, system monitoring, and run-time control. Moreover, we aim at planning the assignment of resources to execute the flow based on specified SLAs and dynamic scheduling of resources to react to over or underutilization and thus prevent the violation of SLAs, while minimizing cost. This is not at all addressed in the earlier work. Moreover, our research methodology aims at delivering a proof-of-concept implementation including performance results and a comparison between distributed and centralized workflow management architectures.

A behaviour preserving transformation of a centralized activity chart, representing a workflow, into an equivalent partitioned one is described in [10] and realized in the MENTOR system [17]. The objective of the work is to enable the parallel execution of the partitioned flow, while minimizing synchronization messages, and analytically prove certain properties of the partitioned flow [10]. This is complementary to our work since we operate with the original business process model without analyzing the process.

Stream processing: In distributed stream processing engines [8, 1, 3, 13] a set of *operators* are installed in the network to execute SQL-like queries over the data streams. These operators input and output a set of streams and may filter, or change the data on these streams. Borealis [1] is a distributed stream processing engine in which streams are queried by a network of operators. In addition to using a proprietary query language, Borealis does not support loops in the query network, which makes it unsuitable for business processes. None of these approaches consider the planning or scheduling of resources based on SLAs determining higher-level business goals.

In the IFLOW [8] distributed stream processing engine, IFLOW nodes are organized in a cluster hierarchy, with nodes higher in the hierarchy assigned more responsibility. For example, the root node is responsible for deploying the entire operator network to its children, and for monitoring the summarized execution statistics of this network. This is different from our completely distributed architecture in which brokers have equal responsibility. IFLOW executes streams according to a utility function that allows the user to schedule streams based on multiple objectives. The concept of a utility function to drive the execution of streams is similar to the articulated objectives of this proposal for using SLAs. However, SLAs provide much finer-grained control, are more expressive, and are widely used in industry, as opposed to the notion of utility functions. The lack of expressiveness in IFLOW is due to the designers choice of associating utility with links, rather than the nodes representing computational services, which makes it difficult to express requirements such as balanced (or minimized) sever load for the executing process. IFLOW does not have a way to drive service selection by, for example, requiring a task to select a service with minimum latency.

Stream processing engines may bear some architectural resemblance to a set of agents executing a business process, but process execution issues are not always easily handled by stream processing engines. First, the stream processing work above is based on proprietary languages, not an industry standard such as BPEL. More significantly, a business process is conceptually not simply a data stream. There are notions of process instances and the accompanying state and isolation semantics that are not required in streams.

In addition to the semantic differences between processes and streams, process distribution in our approach differs from the above work by exploiting an underlying messaging substrate. Like [8], our agents are decoupled by communicating using content-based names instead of network identifiers. In addition, we plan to utilize the advanced features of our messaging substrate, developed in earlier research, to offload some of the agent processing to the network [5, 9]. This simplifies the agents, and lets the network optimize this processing logic.

BPEL: The Business Process Execution

Language (BPEL) standard supports writing distributed applications by composing, or *orchestrating*, Web services. A BPEL process consists of a set of predefined *activities* that allow calling other Web services, assigning variables, or conditionally executing other activities. BPEL programs have properties of traditional programming languages (with concepts of scope, variables, and loops) and workflows (with concepts of parallel and sequential flows). BPEL processes are often authored in a proprietary graphical tool that serializes the process into a standard BPEL XML file.

Several vendors have implemented BPEL execution engines, including IBM, Microsoft, Oracle, and Sun Microsystems. These engines are centralized and must manage an entire BPEL process on one machine. Scalability is typically addressed by load balancing process instances to a cluster of engines, where each engine still executes the entire process. In our approach, however, the individual activities within a process are distributed among the available computing resources. The latter design also allows placing computational activities near the data they operate on, which is not possible in the cluster architecture.

Languages for specifying SLAs: No one single language for specifying SLAs has emerged as standard or de facto standard today. A variety of interesting proprietary attempts for defining languages have been made for several years [14, 7, 15].

WS-Policy [16] — a standard today — is a specification published by the W3C that allows Web service providers to specify policies, such as security and quality of service policies, and allows Web service consumers to specify policy requirements. WS-Policy is designed to express a wide range of policies, but does not exclusively target SLAs and consequently does not include a large set of attributes for specifying SLAs. The application designer has to define the exact SLA terms and attributes on her own. Moreover, WS-Policy is non-declarative and based on XML that can get challenging to read and understand by a human user.

Our research develops language extensions to declaratively specify SLAs. We will build on and extend existing language models [14, 7], and the work presented in this paper most closely resembles WSLA and makes several extensions to it [7].

This paper builds on our prior work on developing efficient messaging and publish/subscribe middleware systems [4, 5, 9]. Most notably, we leverage research on the PADRES project (Publish/Subscribe Applied to Distributed Resource Scheduling). The message routing in the PADRES [5] distributed content-based publish/subscribe system works as follows. Publishers and subscribers connect to one of a set of brokers in an overlay. Publishers specify a template of their event space by submitting an *advertisement* message that is flooded through the broker network and creates a spanning tree rooted at the publisher. Similarly, subscribers specify their interest by sending a *subscription* message that is forwarded along the reverse paths of *intersecting* advertisements, i.e., those with potentially interesting events. Now *publications* from publishers are forwarded along the reverse paths of *matching* subscriptions to interested subscribers.

PADRES extends the traditional publish/subscribe semantics with *composite subscriptions* that allow event correlations to be specified [9]. For example, a subscriber may only be interested in being notified of business processes with at least two failed instances within an hour. The correlation computations are performed at strategic points in the broker network. Another PADRES extension is *historic queries* [5] which allow subscriptions to query for events published in the past in addition to those in the future. This is useful in enterprise applications where past system events may be needed for auditing or analysis purposes. Overall PADRES provides a powerful messaging layer that permits the fine-grained filtering of monitoring events helping to track and monitor higher-level SLAs.

8 Conclusions

The development of business processes in an SOA involves several stages in which different actors concerned with different aspects of the process contribute to the realization of the final process. Each actor must be aware of and fulfill goals associated with the process. In this

paper we present a vision where these goals are formally represented in a WSLA specification in the development tools, and used to simplify each stage of the development cycle from the modelling of the process, through the development of it, to the deployment, execution and runtime monitoring of the process.

A distributed architecture is proposed for the execution of business processes in which lightweight agents collaborate to execute a larger process. This architecture affords scalability by allowing more fine-grained resource allocation, and the ability to strategically move computation close to the data it operates on. To simplify the management of this process execution architecture, a cost model is developed to allow goals, such as minimizing bandwidth resources and process response times, to be independently specified on the executing process, and algorithms are devised to redeploy the executing process to satisfy the specified goals.

This autonomic execution engine is used not only to optimize the execution of business process but also to optimize the monitoring of these processes. Based on the insight that monitoring a process can be modelled as a process itself, a systematic method to map SLAs to a set of monitoring agents is presented. This technique brings the same benefits of the distributed execution architecture to process monitoring as to the execution of processes, namely more efficient use of resources based on strategic deployment of monitoring components.

Evaluations support the ability of the system to repeatedly adapt to changing runtime conditions to achieve a declaratively specified goal. In one workload, the system was able to save about 70% of the bandwidth by adapting a process compared to an initially optimal, but static deployment.

There is much ongoing and future work planned on this research. We are evaluating the system further with more realistic and complex processes, workloads and SLAs. We also plan to compare the trade-offs between the non-optimal but distributed reconfiguration algorithms presented with centralized one with global knowledge of the system configuration. As well, we want to generalize the WSLA extensions to allow arbitrary optimization criteria, which may ultimately result in the ability to recursively apply SLAs on other SLAs.

References

[1] Daniel J Abadi et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[2] G. Alonso, D. Agrawal, et al. Exotica/FMQM: A persistent message-based architecture for distributed workflow managemen. In *IFIP*, 1995.

[3] Sirish Chandrasekaran, Owen Cooper, et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[4] Francoise Fabret, H.-Arno Jacobsen, et al. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD*, 2001.

[5] E. Fidler, Hans-Arno Jacobsen, Guoli Li, and Serge Mankovski. The PADRES distributed publish/subscribe system. In *ICFI*, 2005.

[6] Songlin Hu, V. Muthusamy, Guoli Li, and H.-A. Jacobsen. Transactional mobility in distributed content-based publish/subscribe systems. In *ICDCS*, 2009.

[7] IBM. Web service level agreements (WSLA) project. `http://www.research.ibm.com/wsla/`.

[8] Vibhore Kumar, Zhongtang Cai, et al. Implementing diverse messaging models with self-managing properties using IFLOW. In *ICAC*, 2006.

[9] Guoli Li and Hans-Arno Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *Middleware*, 2005.

[10] Peter Muth et al. From centralized workflow specification to distributed workflow execution. *JII*, 10(2):159–184, 1998.

[11] Chris Nott et al. Using message sets in websphere business integration message broker to implement an ESB in an SOA. `http://www.redbooks.ibm.com/abstracts/redp3978.html`, 2005.

[12] OSOA. Service component architecture. `http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications`.

[13] Peter R. Pietzuch, Jonathan Ledlie, et al. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.

[14] Akhil Sahai et al. Specifying and monitoring guarantees in commercial grids through SLA. *CCGrid*, 2003.

[15] Y. Tang et al. An analysis of web services QoS management infrastructures based on the C-MAPE framework,. In *CoALa*, 2005.

[16] W3C. Web services policy framework (WS-Policy) and Web services policy attachment (WS-PolicyAttachment), 2006. `http://schemas.xmlsoap.org/ws/2004/09/policy/`.

[17] Dirk Wodtke, Jeanine Weisenfels, et al. The Mentor project: Steps toward enterprise-wide workflow management. In *ICDE*, 1996.