SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2014; **00**:1–18
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.2254

Generic Event-Based Monitoring and Adaptation Methodology for Heterogeneous Distributed Systems

Christian Inzinger*[1], Waldemar Hummer[1], Benjamin Satzger[2], Philipp Leitner[3], and Schahram Dustdar[1]

[1]*Distributed Systems Group, Vienna University of Technology, Austria*
[2]*Microsoft Corporation, Redmond, WA, USA*
[3]*University of Zurich, Switzerland*

SUMMARY

The Cloud computing paradigm provides the basis for a class of platforms and applications that face novel challenges related to multi-tenancy, adaptivity, and elasticity. To account for service delivery guarantees in the face of ever increasing levels of heterogeneity, scale and dynamism, service provisioning in the Cloud has raised the demand for systematic and flexible approaches to monitoring and adaptation of applications. In this paper, we tackle this issue and present a framework for efficient runtime management of Cloud environments, and distributed heterogeneous systems in general. A novel domain-specific language (DSL) termed MONINA is introduced that allows to define integrated monitoring and adaptation functionality for controlling such systems. We propose a mechanism for optimal deployment of the defined control operators onto available computing resources. Deployment is based on solving a quadratic programming problem, which aims at achieving minimized reaction times, low overhead, as well as scalable monitoring and adaptation. The monitoring infrastructure is based on a distributed messaging middleware, providing high level of decoupling and allowing new monitoring nodes to join the system dynamically. We provide a detailed formalization of the problem domain, discuss architectural details, highlight the implementation of the developed prototype, and put our work into perspective with existing work in the field.

Received . . .

KEY WORDS: Monitoring; Adaptation; Complex Systems;Domain-Specific Language; Deployment; Operator Placement

1. INTRODUCTION

Efficient monitoring and adaptation of large-scale heterogeneous systems is challenging, as they integrate a multitude of components, often from different vendors. Huge amounts of monitoring data and sophisticated adaptation mechanisms in complex systems render centralized processing of control logic impractical, as the significant network overhead could interfere with production traffic, requiring the use of intricate monitoring strategies selecting only necessary status information in order to minimize communication overhead. Moreover, complex interactions and interdependencies of system components call for advanced adaptation mechanisms, allowing for simple and clear specification of overall system behavior goals, as well as fine-grained control over individual components. In distributed systems it is desirable to keep relevant monitoring and adaptation functionality as local as possible, to reduce traffic and allow for timely reaction to changes.

*Correspondence to: inzinger@dsg.tuwien.ac.at

In this paper we propose an architecture and methodology for managing complex heterogeneous systems using a combination of Complex Event Processing (CEP) [1, 2] techniques to manage and enrich monitoring data, and production rule systems for defining system and component behavior goals to perform necessary adaptations. Furthermore, we introduce a domain-specific language (DSL) to easily and succinctly specify system components and their monitoring and adaptation relevant behavior. It allows to define integrated monitoring and adaptation functionality to realize applications on top of heterogeneous, distributed components. Using the introduced DSL we then outline the process of deploying the integration infrastructure, focussing on the efficient placement of monitoring and adaptation functionality onto available resources. The presented approach is especially suited for deployments in cloud computing environments, as efficient deployment strategies are suitable to reduce infrastructure costs and increase application performance.

A preliminary version of our approach has been presented in [3] where we introduce an outline of the basic concepts and the language constructs of the MONINA language. In this extended work, we deliver a more comprehensive picture of the approach, provide a detailed discussion of our prototype implementation, and put our approach into perspective with related work in greater detail.

The remainder of this paper is structured as follows: In Section 2 we outline a motivating scenario for the following discussion of our contribution. We present an architecture for managing complex heterogeneous distributed systems in Section 3. Section 4 introduces a DSL for concise definition of complex service-oriented systems along with their monitoring and adaptation goals, followed by a discussion of the necessary deployment procedure in Section 5. Relevant previous research is put into perspective with our approach in Section 7. We conclude the paper in Section 8 and provide an outlook for future research directions.

2. SCENARIO

In this section we introduce a motivating scenario based on the problems tackled in the Indenica[†] FP7 EU project. The project aims at providing methods and tools for describing, deploying and managing disparate platforms based on the concept of virtual service platforms (VSPs), which integrate and unify their services.

Complex service-based business applications consist of a multitude of components, both developed in-house, as well as from third parties. Often, multiple alternative products from different vendors exist that offer similar functionalities but exhibit significant fragmentation regarding technology, cost, or quality. A flight booking service from vendor A might, for instance, be implemented to offer SOAP [4] web service endpoints for communication, charge for every request to the system, and offer flights at competitive rates. A competing service from vendor B on the other hand might provide an AMQP interface [5], charge only for booked flights, and offer comparatively expensive flight rates. Depending on the application to be created, either of the offers may be more suitable, and even a combination of multiple services might be appropriate. The problem of deciding on suitable components gets exacerbated when implementing complex applications, as a large number of similar alternatives by different vendors will be available to use, each with different properties regarding dimensions such as technology, cost, or quality. It is therefore increasingly important to design applications to allow for easy and controlled migration of functionality between different components and providers.

Due to different fragmentation aspects, coordination and control of involved services must adapt to changes introduced by switching providers. Service access must be mediated to accommodate for technology differences, whereas coordination and control must be designed to easily compensate for fragmentation of aspects such as cost or quality, i.e., differences in provided functionality as well as different control policies.

Furthermore, deployment mechanisms for complex applications and their control infrastructure must be able to account for available processing capacity on involved hosts, as well as network

[†] http://indenica.eu

DOI: 10.1002/spe.2254

connection properties such as cost and capacity. This is especially important for cloud applications, as efficient deployment of components results in minimized infrastructure costs and maximized application performance.

In the following, we present an architecture and framework to ease the creation, deployment, and management of applications as described above.

3. ARCHITECTURE

In this section, we present an architecture for VSPs to tackle the problems outlined in the scenario above, allowing for integration of heterogenous service platforms, unified management of orchestrated behavior, as well as the addition of domain-specific functionality to be consumed by client applications.

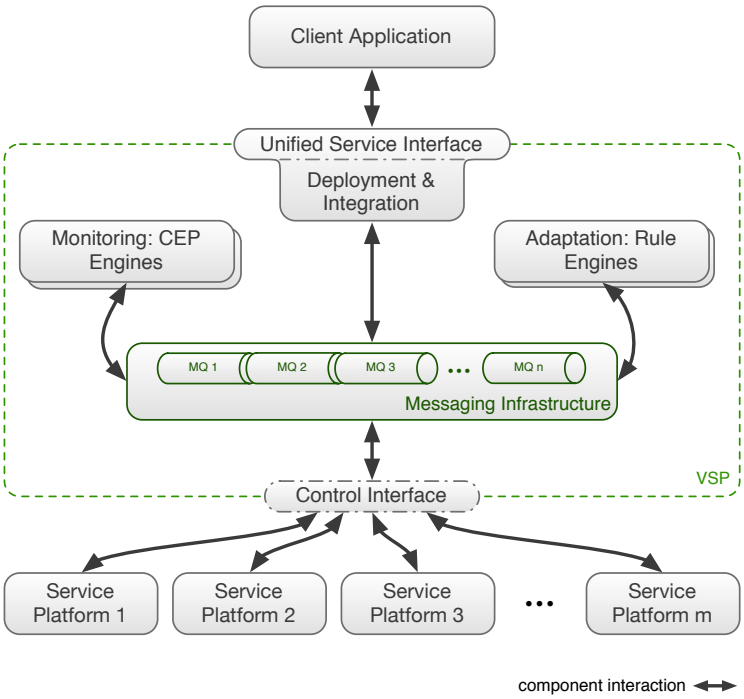


Figure 1. VSP Runtime Architecture

The VSP runtime architecture is presented in Figure 1. A VSP provides a unified view on the functionality of the integrated service platforms that are connected by control interfaces. Monitoring and adaptation are performed by complex event processing (CEP) engines and production rule engines, respectively. Communication within the VSP is based on a distributed messaging infrastructure.

The control interface allows for integration of external services using a wire format transformation layer to accommodate various technologies, such as SOAP, REST, RMI or messaging based solutions such as JMS or AMQP. Furthermore, this interface allows for the specification of emitted monitoring events, as well as supported adaptation actions of connected service platforms.

Monitoring events emitted by integrated services are used within the monitoring infrastructure to derive composite events by aggregating and enriching data emitted by multiple sources (such as the integrated platforms and VSP components) using CEP techniques. The monitoring engines allow for the specification of monitoring queries to derive complex events in order to model the system state in domain-specific terms relevant to stakeholders, abstracting from low-level metrics and system details.

DOI: 10.1002/spe.2254

The modeled system state is then used in the adaptation infrastructure by transforming state change events to so-called facts in the adaptation knowledge base. The adaptation infrastructure utilizes production rule systems to enable sophisticated reasoning on the modeled application state to control VSP behavior. It allows for the specification of adaptation rules that can influence the integrated systems using actions specified in the corresponding control interface.

This clear separation of monitoring and adaptation concerns allows for independent evolution of data derived from the system state and control logic to facilitate the creation of monitoring and adaptation hierarchies. Business rule experts can specify high-level goals for the modeled application's behavior that are evaluated based on domain-specific system state indicators derived from composite monitoring events specified by system experts. The architecture is furthermore designed to allow operators to focus on specifying control logic and let the framework handle decisions about where and how the specified control infrastructure is physically deployed.

Communication between components is realized using a distributed messaging fabric that enables to minimize unnecessary network traffic (compared to a centralized message bus deployment) and further allows components to move freely within the network without changing connection bindings or losing connectivity to the system. The execution of monitoring and adaptation on top of multiple engines further allows for scalable control using distributed resources.

To enable the simplified specification of an application based on the presented architecture we introduce a new DSL called MONINA, which allows the user to specify service platform capabilities, monitoring queries, and adaptation rules. The MONINA language is presented in Section 4. While the presented architecture allows for simple distributed deployment of complex runtime environments, efficient and effective distributed deployment based on optimized component placement poses several challenges. In a deployed system, operating cost and network overhead should be minimal, but the provisioned compute resources must be able to handle the processing load of all deployed components. When external services are integrated, involved components should furthermore be placed "close to" their communication peers to reduce network latency and possibly transmission cost. Strategies to tackle the presented problems and deploy the specified functionality onto available resources will be discussed in Section 5. The prototype implementation based on the presented concepts is discussed in Section 6.

4. MONINA LANGUAGE

In this section we introduce MONINA (Monitoring, Integration, Adaptation) – a DSL allowing for concise, easy, and reusable specification of platforms integrated into a VSP, along with monitoring and adaptation rules governing their behavior.

The language is developed using the Xtext [6] language development framework, allowing for tight integration in the Eclipse platform. The plugin offers syntax highlighting, as well as several automated sanity checks to ease system specification. The language plugin is furthermore integrated into the overall Indenica tool suite, allowing for the usage of existing system models stored in the infrastructure repository. Future versions of the plugin will offer a graphical abstraction in addition to the textual DSL for increased simplicity and ease of use.

Listing 1 shows a simple definition for a service platform to be integrated into a VSP. The 'ApplicationServer' component emits 'RequestFinished' events after processing requests and supports a 'DecreaseQuality' action, which can be triggered by adaptation rules. Emitted events are processed by the 'AggregateResponseTime' query, which aggregates them over five minutes, creating an 'AverageProcessingTime' event. This event is converted to a fact, which might trigger the 'DecreaseQualityWhenSlow' adaptation rule. The physical infrastructure consists of hosts 'vm1' and 'vm2'. Runtime elements without defined costs are assigned default values, which are refined at runtime. In the following we discuss and illustrate the most important language constructs of MONINA in more detail.

event Monitoring events are described by attributes that are contained in emitted messages. Events are then used in component definitions, monitoring query declarations, as well as facts.

DOI: 10.1002/spe.2254

```
event RequestFinished {
  request_id : Integer
  processing_time_ms : Integer
}

event AverageProcessingTime {
  processing_time_ms : Integer
}

action DecreaseQuality {
  amount : Double
}

component ApplicationServer {
  endpoint {
    at "/app_server"
    emit RequestFinished
    action AdjustQuality
  }
  host vm1
  cost 32
}

host vm1 { capacity 128 }
host vm2 { capacity 256 }

query AggregateResponseTimes {
  from ApplicationServer
  event RequestFinished as e
  emit AverageProcessingTime(
    avg(e.processing_time_ms))
  window 5 minutes
}

fact {
  from AverageProcessingTime
}

rule DecreaseQualityWhenSlow {
  from AverageProcessingTime as f
  when f.processing_time_ms > 2000
  execute ApplicationServer
          .DecreaseQuality(5)
}
```

Listing 1: Sample System Definition

fact Facts constitute the knowledge base for adaptation actions. Fact definitions reference an event type and a partition key.

action Similar to events, adaptation actions list all their valid parameters. Actions are used in component definitions as well as adaptation rules.

component A component definition references all monitoring events the platform can emit (including their frequency), all adaptation actions that can be performed, as well as its processing requirements. Furthermore, it is correlated with a concrete instance of the component in question at deployment.

query Monitoring queries are used to define the aggregation, filtering and enrichment of emitted monitoring data in a CEP fashion. Monitoring rules will either emit complex aggregated events to be consumed by other monitoring rules, directly issue adaptation actions, or emit facts to be used in adaptation rules.

rule Adaptation rules allow for the usage of complex business management rules to govern system behavior. Monitoring rules emit facts to be used for reasoning over the current system state. Adaptation rules can either publish new facts or issue adaptation actions.

host Hosts represent possible deployment locations for components, monitoring queries and adaptation rules. A host description contains its processing capacity.

4.1. Event

In our work, we follow the event-based interaction paradigm [7]. Events are emitted by components to signal important information. Furthermore, events can be emitted by monitoring queries as a result of the aggregation or enrichment of one or more source events.

Figure 2 shows a simplified grammar of the event construct in Extended Backus-Naur Form (EBNF). Event declarations start with the **event** keyword and an event type identifier. As shown in the figure, an event can contain multiple attributes, defined by specifying name and type separated by a colon. Currently, supported event types are a variety of Java types such as `String`, `Integer`, `Decimal`, and `Map<?,?>`.

DOI: 10.1002/spe.2254

⟨*event*⟩ ::= 'event' ⟨*ID*⟩ '{' ⟨*attr*⟩* '}'
⟨*attr*⟩ ::= ⟨*attr-name*⟩ ':' ⟨*type*⟩
⟨*attr-name*⟩ ::= ⟨*ID*⟩

Figure 2. Simplified Event Grammar in EBNF

Since listing all available event types for every application would be a tedious and error-prone task, we automatically gather emitted event types from known components to improve reusability and ease of use. This procedure is described in more detail in Section 4.4.

More formally, we assume that E is the set of all event types, T is the set of all data types, and each event type $E' \in E$ is composed of event attribute types $E' = (\alpha_1, \ldots, \alpha_k)$, $\alpha_i \in T \; \forall i \in \{1, \ldots, k\}$. $\mathcal{I}_E$ denotes the set of monitoring event instances (or simply events), and each event $e \in \mathcal{I}_E$ has an event type, denoted $t(e) \in E$. The attribute values contained in event e are represented as a tuple $e = (\pi_{\alpha_1}(e), \ldots, \pi_{\alpha_k}(e))$, where $\pi_{\alpha_x}(e)$ is the projection operator (from relational algebra), which extracts the value of some attribute α_x from the tuple e.

4.2. Action

Complementary to monitoring events described above, adaptation actions are another basic language element of MONINA. Adaptation actions are invoked by adaptation rules and executed by corresponding components to modify their behavior. Figure 3 shows a simplified grammar of the action construct in EBNF. Action declarations start with the **action** keyword followed by the action type identifier. Furthermore, actions can take parameters, modeled analogously to event attributes shown in Figure 2.

⟨*action*⟩ ::= 'action' ⟨*ID*⟩ '{' ⟨*attr*⟩* '}'

Figure 3. Simplified Action Grammar in EBNF

Similar to events, adaptation actions offered by known components do not need to be specified manually, but are automatically derived from component specifications, as discussed in Section 4.4.

The symbol A denotes the set of all types of adaptation actions, and each type $A' \in A$ contains attribute types: $A' = (\alpha_1, \ldots, \alpha_k)$, $\alpha_i \in T \; \forall i \in \{1, \ldots, k\}$. The set $\mathcal{I}_A$ stores all action instances (or simply actions) that are issued in the system. The values of an action $a \in \mathcal{I}_A$ are evaluated using the projection operator: $a = (\pi_{\alpha_1}(a), \ldots, \pi_{\alpha_k}(a))$.

4.3. Fact

Facts constitute the knowledge base for adaptation rules and are derived from monitoring events. A fact incorporates all attributes of the specified source event for use by adaptation rules. Figure 4 shows a simplified grammar of the fact construct in EBNF. Fact declarations start with the **fact** keyword and an optional fact name. A fact must specify a source event type that is used to derive the fact from. Furthermore, an optional partition key can be supplied. If the fact name is omitted, the fact will be named after its source event.

⟨*fact*⟩ ::= 'fact' ⟨*ID*⟩? '{' ⟨*ID*⟩ ⟨*partition-key*⟩? '}'
⟨*partition-key*⟩ ::= 'by' ⟨*ID*⟩

Figure 4. Simplified Fact Grammar in EBNF

DOI: 10.1002/spe.2254

The partition key construct is used to enable the creation of facts depending on certain event attributes, allowing for the concise declaration of multiple similar facts for different system aspects. For instance, a fact declaration for the event type `ProcessingTimeEvent` that is partitioned by the `component_id` attribute will create appropriate facts for all encountered components, such as `ProcessingTime(Component1)`, ..., `ProcessingTime(ComponentN)`. In contrast, a fact declaration for the `MeanProcessingTimeEvent` without partition key will result in the creation of a single fact representing the system state according to the attribute values of incoming events.

Formally, a fact $f \in F$ is represented as a tuple $f = (\kappa, e)$, for event type $e \in E$ and partition key κ. The optional partition key κ allows for the simplified creation of facts concerning specified attributes, to model facts relating to single system components, using $\pi_\kappa(e)$, the projection of attribute κ from event e. Alternatively, the type of event e itself acts as the partition key, aggregating all events of the same type to a single fact.

4.4. Component

A component declaration incorporates all information necessary to integrate third-party system into the Indenica infrastructure. Figure 5 shows a simplified grammar of the component construct in EBNF. Component declarations start with the **component** keyword and a component identifier. A component specifies all monitoring events it will emit with an optional occurrence frequency, supported adaptation actions, as well as a reference to the host on which the component is deployed.

⟨*component*⟩ ::= '`component`' ⟨*ID*⟩ '`{`' ⟨*metadata*⟩? ⟨*c-elements*⟩* ⟨*host-ref*⟩ '`}`'

⟨*metadata*⟩ ::= ('`vendor`' ⟨*STRING*⟩)? ('`version`' ⟨*STRING*⟩)? ...

⟨*c-elements*⟩ ::= ⟨*endpoint*⟩ | ⟨*refs*⟩

⟨*refs*⟩ ::= ⟨*event-ref*⟩ | ⟨*action-ref*⟩

⟨*action-ref*⟩ ::= '`action`' ⟨*ID*⟩

⟨*event-ref*⟩ ::= '`event`' ⟨*ID*⟩ ⟨*frequency*⟩?

⟨*endpoint*⟩ ::= '`endpoint`' ⟨*ID*⟩? '`{`' ⟨*e-addr*⟩ ⟨*refs*⟩* '`}`'

⟨*frequency*⟩ ::= '`every`' ⟨*Decimal*⟩ '`seconds`' | ⟨*Decimal*⟩ '`Hz`'

⟨*host-ref*⟩ ::= '`host`' ⟨*ID*⟩

Figure 5. Simplified Component Grammar in EBNF

For brevity, further elements such as endpoint addresses, are omitted in the presented grammar snippet but are included in the implementation.

As mentioned before, it is usually not necessary to manually specify component, action, and event declarations. The Indenica infrastructure provides for means to automatically gather relevant information from known components through the control interface shown in Figure 1.

Formally, components C are represented with the signature function[‡]

$$sig : C \rightarrow \mathcal{P}(A) \times \mathcal{P}(\{(e_j, \nu_j) | e_j \in E, \nu_j \in \mathbb{R}_0^+\}) \times \mathbb{R}_0^+ \times H$$

and the signature for a component $c_i \in C$ is

$$sig : c_i \mapsto (I_i^A, \Omega_i^E, \psi_i, h_i)$$

[‡]For clarity, we use the same symbol sig for signatures of components (Section 4.4), monitoring queries (Section 4.5), adaptation rules (4.6), and hosts (Section 4.7).

DOI: 10.1002/spe.2254

The signature function sig extracts relevant information from the according language construct for later use by the deployment infrastructure. Monitoring events emitted by the component are represented by Ω_i^E, and for each emitted event type e_j an according frequency of occurrence ν_j is supplied. Adaptation actions supported by the component are denoted by I_i^A, its processing cost is represented by ψ_i, and h_i identifies the host on which the component is deployed.

4.5. Monitoring Query

Monitoring queries allow for the analysis, processing, aggregation and enrichment of monitoring events using CEP techniques. In the context of the Indenica project we provide a simple query language tailored to the needs of the specific solution.

A simplified EBNF grammar of the monitoring query construct is shown in Figure 6. A query declaration starts with the **query** keyword and a query identifier. Afterwards, an arbitrary number of event sources for the query is specified using the **from** and **event** keywords to specify source components and event types. A query then specifies any number of event emission declarations, denoted by the **emit** keyword followed by the event type and a list of expressions evaluating the attribute assignments of the event to be emitted. For brevity we omit the specification of ⟨*cond-expression*⟩ clause that represents a SQL-style conditional expression. Queries can be furthermore designed to operate on event stream windows using the **window** keyword, specifying either a number of events to create a batch window or a time span to create a time window. Conditions expressed using the **where** keyword are used to limit query processing to events satisfying certain conditions, using the conditional expression construct mentioned above. Finally, queries can optionally indicate the rate of incoming vs. emitted events, as well as an indication of required processing power. These values are user-defined estimations in the initial setup, and are adjusted continuously during runtime to accommodate changes in the environment.

⟨*query*⟩ ::= ‘`query`’ ⟨*ID*⟩ ‘`{`’ (⟨*sources*⟩ | ⟨*emits*⟩)*
⟨*window*⟩? ⟨*condition*⟩? ⟨*io-ratio*⟩? ⟨*cost*⟩? ‘`}`’

⟨*sources*⟩ ::= ‘`source`’ ⟨*ID*⟩ (‘`,`’ ⟨*ID*⟩)*
‘`event`’ ⟨*ID*⟩ (‘`,`’ ⟨*ID*⟩)*

⟨*emits*⟩ ::= ‘`emit`’ ⟨*ID*⟩ (⟨*attr-emit*⟩*)*

⟨*attr-emit*⟩ ::= ⟨*cond-expression*⟩ (‘`as`’ ⟨*ID*⟩)?

⟨*window*⟩ ::= ‘`window`’ (⟨*batch-window*⟩ | ⟨*time-window*⟩)

⟨*batch-window*⟩ ::= ⟨*Integer*⟩ ‘`events`’

⟨*time-window*⟩ ::= ⟨*Integer*⟩ (‘`seconds`’ | ‘`minutes`’ | ‘`days`’ | ...)

⟨*condition*⟩ ::= ‘`where`’ ⟨*cond-expression*⟩

⟨*io-ratio*⟩ ::= ‘`ratio`’ ⟨*Decimal*⟩

⟨*cost*⟩ ::= ‘`cost`’ ⟨*Decimal*⟩

Figure 6. Simplified Monitoring Query Grammar in EBNF

In addition to the query construct presented above, the language infrastructure allows for the integration of other CEP query languages, such as EQL [8] if necessary.

The set of queries Q is represented using the signature

$$sig : Q \rightarrow \mathcal{P}(E) \times \mathcal{P}(E) \times \mathbb{R}_0^+ \times \mathbb{R}_0^+$$

and the signature for a query $q_i \in Q$ is

$$sig : q_i \mapsto (I_i^E, O_i^E, \rho_i, \psi_i)$$

DOI: 10.1002/spe.2254

Input and output event streams are denoted by I_i^E and O_i^E respectively, where ρ_i represents the ratio of input events processed to output events emitted, and ψ_i represents the processing cost of the query.

4.6. Adaptation Rule

Adaptation rules employ a knowledge base consisting of facts to reason on the current state of the system and modify its behavior when necessary using a production rule system. Figure 7 shows a simplified grammar of the adaptation rule construct in EBNF. A rule declaration starts with the **rule** keyword and a rule identifier. After importing all necessary facts using the **from** keyword, a rule contains a number of **when**-statements where the condition evaluates a ⟨*cond-expression*⟩ as described above, referencing imported facts, and the **then** block specifies a number of adaptation action invocations including any necessary parameter assignments. Optionally, a rule can indicate processing requirements (cf. Figure 6) that will be adjusted at runtime.

⟨*rule*⟩ ::= ‘`rule`’ ⟨*ID*⟩ ‘`{`’ (⟨*r-sources*⟩)+ ⟨*stmt*⟩+ ⟨*cost*⟩?‘`}`’

⟨*r-sources*⟩ ::= ‘`from`’ ⟨*ID*⟩ (‘`as`’ ⟨*ID*⟩)?

⟨*stmt*⟩ ::= ‘`when`’ ⟨*cond-expression*⟩ ‘`then`’ ⟨*action-expr*⟩+

⟨*action-expr*⟩ ::= ⟨*ID*⟩ ‘`(`’ ⟨*action-attr*⟩ (‘`,`’ ⟨*action-attr*⟩)* ‘`)`’

⟨*action-attr*⟩ ::= ⟨*cond-expression*⟩ (‘`as`’ ⟨*ID*⟩)?

Figure 7. Simplified Adaptation Rule Grammar in EBNF

As with monitoring queries, the adaptation rule module is tailored to the requirements of the Indenica infrastructure but also allows for the usage of different production rule languages, such as the Drools [9] rule language, if more complex language constructs are required.

More formally, the set of rules R is represented with the signature function

$$sig : R \rightarrow \mathcal{P}(F) \times \mathcal{P}(A) \times \mathbb{R}_0^+$$

and the signature for a rule $r_i \in R$ is

$$sig : r_i \mapsto (I_i^F, O_i^A, \psi_i)$$

The set of facts from the knowledge base used by the adaptation rule are denoted by F_i, while A_j representes the adaptation actions performed, and ψ_i represents te processing cost of the rule.

4.7. Host

Hosts represent the physical infrastructure available for deployment of infrastructure components. Figure 8 shows a simplified grammar of the host construct in EBNF. A host declaration starts with the **host** keyword and a host name. An address in the form of a fully qualified domain name (FQDN) or an IP address can be supplied. If no address is given, the host name will be used instead. Furthermore, a **capacity** indicator is provided that will be used for deployment decisions.

⟨*host*⟩ ::= ‘`host`’ ⟨*ID*⟩ ‘`{`’ ⟨*address*⟩? ⟨*capacity*⟩ ‘`}`’

⟨*address*⟩ ::= ⟨*fqdn*⟩ | ⟨*ip-address*⟩

⟨*capacity*⟩ ::= ‘`capacity`’ ⟨*Decimal*⟩

Figure 8. Simplified Host Grammar in EBNF

DOI: 10.1002/spe.2254

The set of hosts H is represented with the signature function

$$sig : H \rightarrow \mathbb{R}_0^+$$

and the signature for a host $h_i \in H$ is

$$sig : h_i \mapsto (\psi_i)$$

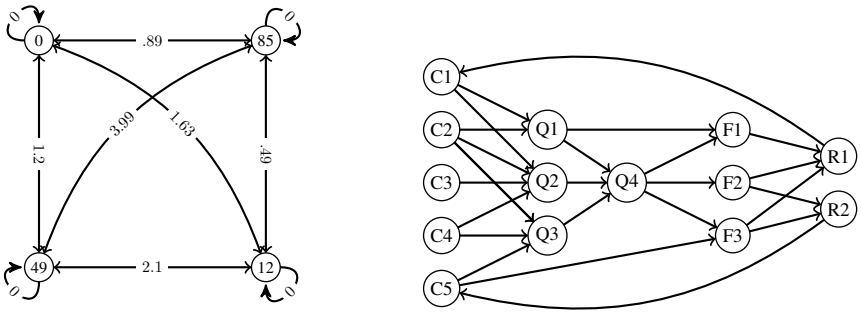
with the capacity of a host represented by ψ_i.

5. DEPLOYMENT OF MONITORING QUERIES AND ADAPTATION RULES

In this section, we propose a methodology for efficiently deploying runtime elements, based on the system description defined in a MONINA file. The deployment strategy attempts to find an optimal placement with regard to locality of information producers and consumers, resource usage, network load, and minimal reaction times. Our deployment procedure consists of three main stages. First, an infrastructure graph is generated from the host declarations in the MONINA definition to create a model of the physical infrastructure. Then, a dependency graph is derived from component, query, fact, and rule definitions. Finally, a mathematical optimization problem is formulated based on both graphs, which is utilized to find an optimal deployment scheme.

5.1. Infrastructure Graph

The infrastructure graph $G_I = (V_I, E_I)$ is a directed graph which models the available infrastructure. The nodes of the graph represent execution environments. We will refer to execution environments as hosts, even though they might not only represent single machines, but more complex execution platforms. The graph's edges represent the connection between hosts. The capacity function $c_I : V_I \rightarrow \mathbb{R}_0^+$ assigns to each host its capacity for hosting runtime elements, e.g., monitoring queries or adaptation rules. A capacity of zero prohibits any runtime elements on the host. Edge weight function $w_I : E_I \rightarrow \mathbb{R}_0^+$ models the delay between two hosts. Values close to zero represent good connection. For the sake of convenience we assume that each vertex has a zero weighted edge to itself. Figure 9a shows an exemplary infrastructure graph.



(a) Infrastructure Graph

(b) Dependency Graph

Figure 9. Graphs generated from a MONINA description

The infrastructure graph is generated based on a MONINA description, i.e., its node set V_I is taken from the description file, which also contains the hosts' physical addresses. The next step is the exploration of the edges based on the *traceroute* utility, which is available for all major operating systems. It allows, amongst others, measuring transit delays. Furthermore, node capacities can be read by operating system tools to complement missing MONINA values. In Unix-like operating systems, for instance, the `/proc` pseudo-filesystem folder provides information about hardware and its utilization.

DOI: 10.1002/spe.2254

5.2. Dependency Graph

Dependency graphs model the dependencies between components, monitoring queries, facts, and adaptation rules. A dependency graph $G_D = (V_D, E_D)$ is a directed, weighted graph, whose node set $V_D = C \cup Q \cup F \cup R$ is composed of pairwise disjoint sets C, Q, F, and R. These represent components, queries, facts, and rules, respectively. Edges represent dependencies between these entities, i.e., exchange of events, and weight function $w_D : E_D \to \mathbb{R}_0^+$ quantifies the relative number of events. Another function $e_D : E_D \to E$ maps edges to events they are based on, where E is the set of event types. Components are event emitters, which may be consumed by queries or may be converted into a fact in a knowledge base. Queries consume events from components or other queries producing new events. Knowledge bases convert certain events into facts. Rule engines work upon knowledge bases, and the rules trigger actions if respective conditions become true. Edges link event emitters (components or queries) to respective event consumers (queries or knowledge bases). They also connect knowledge bases to rules relying on facts they are managing. Finally, rules are linked to the components they are adapting, i.e., components in which they trigger adaptation actions. Thus, the edge set is limited to the following subset $E_D \subseteq (C \times Q) \cup (C \times F) \cup (Q \times Q) \cup (Q \times F) \cup (F \times R) \cup (R \times C)$. Figure 9b shows an exemplary dependency graph. Event types and edges weights are omitted for readability.

The generation of a dependency graph is based on a MONINA description. Initially, the dependency graph $G_D = (V_D, E_D)$ is created as a graph without any edges, i.e., $V_D = C \cup Q \cup F \cup R$ and $E_D = \emptyset$, where C, Q, F, R are taken from the MONINA description. Then, edges are added according to the following edge production rules.

Component → Query. An edge $c \xrightarrow{\psi} q$ is added to E_D for every component $c \in C$, query $q \in Q$, and event $e \in (O^E \cap I^E)$, where $sig(c) = (\bullet, O^E, \psi, \bullet)$ and $sig(q) = (I^E, \bullet, \bullet, \bullet)$. In case an edge $c \xrightarrow{\psi_2} q$ is supposed to be added to E_D, but E_D already contains $c \xrightarrow{\psi_1} q$, then the latter is replaced by $c \xrightarrow{\psi_1+\psi_2} q$. For all following edge production rules we assume that edges that already exist are merged by adding weights, like here.

Component → Fact. An edge $c \xrightarrow{\psi} f$ is added to E_D for every component $c \in C$, fact $f \in F$, and event $e \in O^E$, where $sig(c) = (\bullet, O^E, \psi, \bullet)$ and $f = (\bullet, e)$.

Query → Query. An edge $q_1 \xrightarrow{\rho} q_2$ is added to E_D for all queries $q_1, q_2 \in Q$ and event $e \in (O_1^E \cap I_2^E)$, where $q_1 \neq q_2$, $sig(q_1) = (\bullet, O_1^E, \rho, \bullet)$ and $sig(q_2) = (I_2^E, \bullet, \bullet, \bullet)$.

Monitoring Query → Fact. An edge $q \xrightarrow{\rho} f$ is added to E_D for every query $q \in Q$, fact $f \in F$, and event $e \in O^E$, where $sig(q) = (\bullet, O^E, \rho, \bullet)$ and $f = (\bullet, e)$.

Fact → Adaptation Rule. An edge $f \to r$ is added to E_D for every fact $f \in F$ and adaptation rule $r \in R$, where $f \in I^F$ and $sig(r) = (I^F, \bullet, \bullet)$.

Adaptation Rule → Component. An edge $r \to c$ is added to E_D for every adaptation rule $r \in R$ and component $c \in C$, where $a \in (O^A \cap I^A)$, $sig(r) = (\bullet, O^A, \bullet)$ and $sig(c) = (I^A, \bullet, \bullet, \bullet)$.

5.3. Quadratic Programming Problem Formulation

Quadratic programming [10] is a mathematical optimization approach, which allows to minimize/maximize a quadratic function subject to constraints. Assume that $\mathbf{x}, \mathbf{b}, \mathbf{c}, \mathbf{d} \in \mathbb{R}^n$ are column vectors, and $Q \in \mathbb{R}^{n \times n}$ is a symmetric matrix. Then, a quadratic programming problem can be defined as follows.

DOI: 10.1002/spe.2254

$$\min_{\mathbf{x}} \ f(\mathbf{x}) = \tfrac{1}{2}\mathbf{x}^T Q\mathbf{x} + \mathbf{c}^T\mathbf{x}$$

Subject to

$$E\mathbf{x} = \mathbf{d} \qquad \text{(Equality constraint)}$$
$$A\mathbf{x} \leq \mathbf{b} \qquad \text{(Inequality constraint)}$$

We want to achieve an optimal mapping (i.e., physical deployment) of the dependency graph onto the infrastructure graph. Runtime entities described in the dependency graph that depend on each other should be as close as possible, in the best case running on the same host. This results in fast reactions, timely adaptations, and low network overhead. On the other hand, hosts have capacity restrictions, which have to be considered. Adding more hosts (scaling out) is often the only possibility to cope with growing load. Our mapping approach is able to find the *optimal* tradeoff between the suboptimal strategies (1) putting everything on the same host and (2) evenly/randomly distributing runtime elements among the available hosts.

Since we want to get a mapping from the optimization process, we introduce placement variables p_{v_I,v_D} for each host $v_I \in V_I$ in the dependency graph and each runtime element $v_D \in V_D$ in the dependency graph. Each of these variables has a binary domain, i.e., $p_{v_I,v_D} \in \{0,1\}$. The assignment $p_{v_I,v_D} = 1$ decodes that runtime element v_D is hosted on v_I, $p_{v_I,v_D} = 0$ stands for v_D is not running on host v_I. This results in $|V_I| \cdot |V_D|$ binary variables, whose aggregation can be represented as a single vector $\mathbf{p} \in \{0,1\}^{|V_I|\cdot|V_D|}$.

To find out the optimal mapping of the dependency graph onto the infrastructure, we solve the following optimization problem, which can be classified as binary integer quadratic programming problem, based on the form of variable $\mathbf{p}$ and the function to minimize.

$$\min_{p} \sum_{e_I \in E_I} w_I(e_I) \cdot \sum_{e_D \in E_D} w_D(e_D) \cdot p_{v_I^1,v_D^1} \cdot p_{v_I^2,v_D^2} \tag{1}$$

Subject to

$$\forall c \in C: \ p_{h(c),c} = 1 \tag{2}$$
$$\forall v_D \in V_D: \sum_{v_I \in V_I} p_{v_I,v_D} = 1 \tag{3}$$
$$\forall v_I \in V_I: \sum_{v_D \in V_D} p_{v_I,v_D} \cdot c_D(v_D) \ \leq c_I(v_I) \tag{4}$$

The function to minimize (1) calculates for each edge $e_I = (v_I^1, v_I^2)$ in the infrastructure graph and each edge $e_D = (v_D^1, v_D^2)$ the weight that incurs if this particular dependency edge is mapped to this particular infrastructure edge. If both runtime elements (v_D^1 and v_D^2) are mapped to the same node no weight is added to the function, because all self-links have weight zero. The first equality constraint (2) fixes the mapping for every component $c \in C \subseteq V_D$ to the hosts they are statically assigned to, as defined in MONINA and represented by $h(c)$, where $sig(c) = (\bullet, \bullet, \bullet, h)$. We assume that components are bound to hosts. If there exist components that can be deployed on any host and do not have an assignment in MONINA, then this can be handled by simply omitting the respective constraint for this component. The second equality constraint (3) defines that each node from the dependency graph is mapped to exactly one node in the infrastructure graph. Finally, the inequality constraint (4) requires that for all hosts the summarized costs of all elements they are hosting is less than the respective capacity. The function $c_D : V_D \to \mathbb{R}_0^+$ represents the costs of executing a runtime element v_D, as defined in the MONINA description.

We use the Gurobi optimizer [11] for solving the optimization problem as described above. Runtime aspects of the currently implemented deployment module are discussed in Section 6.

DOI: 10.1002/spe.2254

5.4. *Deployment in Cloud Computing Environments*

The presented approach is suitable for continuous deployments in order to react to changes in the runtime environment. If new rules are added or communication characteristics change significantly we derive a new deployment strategy based on the existing structure. The goal of continuous (re-)deployment is to maintain a near-optimal component distribution while minimizing the changes to be performed. By moving as little components as possible, we minimize the cost of transferring component state information between machines.

This model of continuous optimization and re-deployment integrates perfectly with the concept of Cloud computing [12], which allows to dynamically allocate and release computing resources to implement elastically scaling applications. Cloud environments fulfill two prerequisites which are central to our approach. First, the Cloud provides the possibility to acquire a practically unbounded number of virtual machine (VM) instances. In the optimization procedure of our approach, application components and monitoring queries are placed on hosts, and Cloud computing effectively removes any potential limits of the optimization procedure with regards to the number of hosts. Our approach considers this by dynamically adjusting the number of hosts available for deployment planning. Hosts are added to the solution space until a solution can be found that does not violate any placement constraints. Second, cost aspects are typically an integral part of (commercial) Cloud offerings, hence we can directly incorporate the computation and communication costs into our optimization model. In addition, many Cloud providers offer a convenient set of pre-configured software tools which simplify the implementation of our approach, including distributed messaging fabrics for de-centralized event transmission, host and network monitoring tools for obtaining the decision basis of our optimization, data storage services for persisting (event) data, and more.

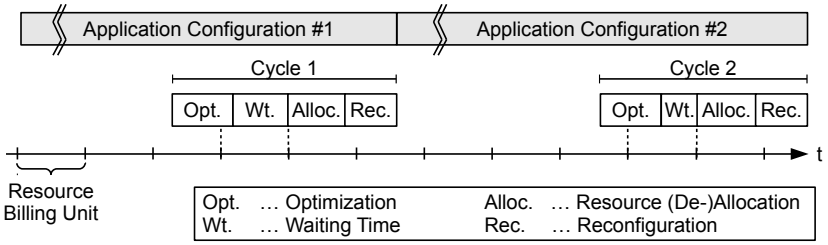


Figure 10. Cost-Efficient Optimization and Re-Configuration in Cloud Environments

Note, however, that the commercial nature of Cloud computing entails certain peculiarities, which should be taken into account for our approach. In particular, Cloud resources are typically subject to a billing cycle (e.g., VMs are billed in units of one hour), which requires that the optimization approach be adjusted in order to achieve optimal results. To address this issue, we suggest to perform adaptations in cycles, as illustrated in Figure 10. The figure shows a timeline which is split up into the billing units of computing resources, e.g., one hour (for simplification we assume that all resources are stopped/started simultaneously in each cycle). The grey bars at the top illustrate the current configuration (first #1, then #2) of the application whose deployment we strive to optimize. We assume that two adaptations are triggered over the duration of the timeline, consisting of four main parts each: optimization procedure, waiting time, allocating and de-allocating of resources, and reconfiguration of the application based on the new resource allocations. The essential part is that the change in resource allocation should be aligned with the expiry time of the resource billing unit. If this alignment were not implemented, the unused resource utilization corresponding to the "waiting time" would be wasted from a cost perspective. Depending on the duration of the optimization algorithm (in relation to the resource billing unit), the duration of the waiting time should be minimized, in order to avoid changes in the environmental conditions which could potentially result in a different optimum at the time the adaptations get applied.

DOI: 10.1002/spe.2254

6. IMPLEMENTATION

In this section we discuss the implementation of the concepts presented in this paper. The developed prototype is available for download from the prototype web site[§]. As mentioned above, application specification using the MONINA language is implemented as Eclipse plugin. We use the Xtext language development framework to model MONINA. The plugin offers convenience functions such as syntax highlighting, code completion and static analysis of system specifications to detect definition errors. Fig. 11 shows a screen shot of the editor, illustrating some of the implemented features. After specifying the relevant system structure, the MONINA plugin generates a set of configuration directives to be used with the runtime infrastructure[¶].

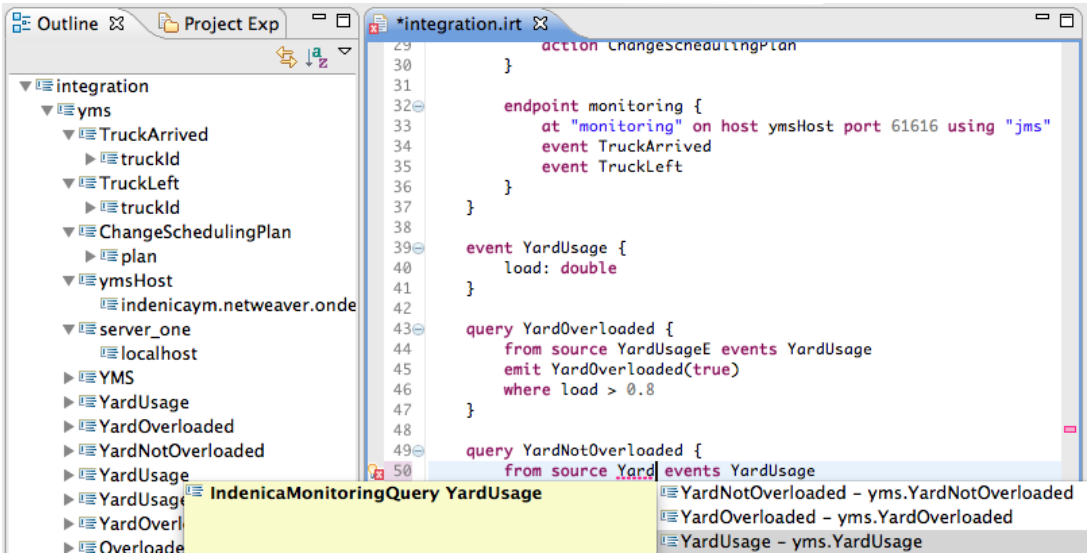


Figure 11. Sample Screen Shot of MONINA Editor

The runtime infrastructure implements the architecture presented in Section 3. At the core of the runtime, a distributed messaging fabric (currently based on embedded ActiveMQ [13] brokers) allows for extensible realization of distributed applications. The messaging fabric automatically establishes a communication mesh between application components deployed in the same network using multicast discovery. Components deployed in different networks need to know a single address per external network to establish connections to all relevant modules of the deployed app. When using a MONINA description for deployment, this information can be gathered from the contained `host` declarations and the deployment strategy. The messaging fabric furthermore establishes conventions for component discovery and management, such as common topic names and management endpoint addresses for infrastructure components to subscribe to. Furthermore, the messaging fabric allows components to register for communication paths or event streams they are interested in to create the runtime interaction structure. Messages are then delivered to components via the best available path (considering latency and bandwidth) to significantly reduce the introduced traffic overhead compared to solutions using centralized messaging middleware. Future versions of the framework will take additional factors, such as communication cost into account to enable more fine-grained control over the communication behavior of deployed applications.

The control interface is realized using the Apache Tuscany [14] SCA container to allow for easy integration of different interface technologies, such as SOAP and REST web services, Java Messaging Service (JMS), Remote Method Invocation (RMI) or the Common Object

[§]http://indenicatuv.github.io/releases/
[¶]https://github.com/inz/indenica-runtime-core

DOI: 10.1002/spe.2254

Request Broker Architecture (CORBA). Configuration directives from the MONINA application specification are used to establish connections to external components by registering monitoring event sinks and adaptation endpoints.

As described in the architecture in Section 3, the monitoring engines process event streams from integrated external components as well as queries running within the system to derive complex events representing relevant application state changes. In the current implementation, we use the Esper [8] CEP engine to perform event stream processing. Source events are received via the messaging fabric, processed using monitoring queries defined in the MONINA description, and derived events are handed back to the messaging infrastructure to make them available for further processing by other monitoring queries or adaptation rules.

The adaptation engines execute production rules on the current system state to influence its properties in order to maintain or achieve desired behavior. In the current implementation, we use the Drools Expert [9] rule engine to execute adaptation rules as specified in the MONINA system description. To establish a knowledge base from the available system state change events, a fact transformer component is used, translating a stream of state change events into a fact object representation that can be used in production rules. In the current implementation, fact transformation is handled using the Esper CEP engine, aggregating state changes into appropriate facts. Adaptation rules act on conditions about the state of facts in the knowledge base and can execute adaptation actions on integrated components. Actions to be executed are delivered to the according control interfaces by the messaging fabric. The control interface will then perform the actual execution of adaptation actions on the external component.

Control interface specifications, monitoring queries, fact transformation, and adaptation rules can be submitted to the system in multiple formats. MONINA descriptions are supported as a portable, technology-independent behavior specification, but component-native directives, such as raw Esper Event Processing Language (EPL) queries or Drools Rule Language (DRL) rules are supported by the currently implemented monitoring and adaptation engines respectively.

Application deployment is carried out using the deployment component. It analyzes the supplied system specification, starts components on the available nodes according to computed deployment strategy, and deploys all necessary configuration artifacts such as endpoint definitions for external communication, monitoring queries, fact transformation rules, and adaptation rules. The deployment strategy is currently realized using the Gurobi [11] optimizer to solve the mapping problem discussed in Section 5. In the current version, a MONINA specification will be deployed according to cost and communication traffic estimations provided with the system description and can be redeployed based on interaction information gathered during runtime. In the future, we will extend the deployment strategy module to allow incremental deployments considering the costs of migrating existing components, rules, and queries.

Extensible interface design throughout the implemented framework allows for easy extension or replacement of components if required to avoid potential vendor lock-in.

7. RELATED WORK

In this section we discuss important previous work related to event-based monitoring and adaptation, as well as optimized deployment of query operators in monitoring infrastructures. Although some of the seminal work dates back to the pre-Cloud era, we also emphasize the relevance of these approaches for Cloud-based monitoring.

Monitoring of QoS and SLAs. Previous work on monitoring and adaptation of distributed heterogeneous systems is mainly concerned with establishing and monitoring Service Level Agreements (SLAs) and Quality of Service (QoS) policies. SLAs are typically composed of a number of Service Level Objectives (SLOs) [15] which correspond to the monitoring metrics, denoted facts, in our approach. The work by Comuzzi et al. [16] discusses a wholistic SLA management approach. Whereas their work is strongly focused at the process for SLA establishment, we assume that the SLAs and the corresponding SLO metrics are known to the service provider. The MONINA language then facilitates the definition of raw facts (emitted by

DOI: 10.1002/spe.2254

monitoring agents) and complex or derived facts (resulting from monitoring queries) to monitor the values of these SLOs. One of the core issues in service computing (and more recently Cloud computing) is the efficient generation of adaptation policies. The approach by Jung et al. [17] generates adaptation policies for multi-tier applications in consolidated server environments. The authors argue that online adaptation approaches based on, both, control theory and rule-based expert systems, have disadvantages. Hence, a hybrid approach is proposed which combines the best of both worlds. Their approach builds on queuing theoretic models for predicting system behavior, in order to automatically generate optimal system configurations. Our approach, on the other hand, abstracts from the type of monitoring data (whether predicted or actual values are used), and focuses on efficient definition and deployment of monitoring infrastructures. The work by Cardellini et al. [18] targets QoS-driven runtime adaptation of service oriented architectures. The presented approach does not, however, consider the efficient placement of the monitoring and adaptation rules themselves, but relies on decent initial placement or intervention by the operator. Our work contributes an integrated approach which allows high-level definition of application topologies, which are then mapped to infrastructure graphs and deployed in Cloud environments.

Optimized Deployment of Monitoring Queries. The performance of monitoring infrastructures depends on the topology and data flow between query operators, hence efficient operator placement plays a key role. The work by Lakshmanan et al. [19] provides an overview of eight different operator placement algorithms, which are evaluated with respect to five core dimensions: node location (clustered/distributed), data rates (bursty/uniform), administrative domain (single/multiple), topology changes (dynamic/uniform), and queries (redundant/heterogeneous). Algorithms for efficient operator placement in widely-distributed systems are presented in [20]. Also the work by Pietzuch et al. [21] has influenced our work. Their approach performs operator placement using a stream-based overlay network for decentralized optimization decisions. A decentralized algorithm for near optimum operator placement in heterogeneous CEP systems is presented in [22]. The algorithm in [23] models the system load as a time series X and computes the load correlation coefficient ρ_{ij} for pairs of nodes i and j. The optimization goal is to maximize the overall correlation, which has the effect that the load variance of the system is minimized. A comprehensive and fine-grained model of CPU capacity and system load is provided in [24]. The *feasible set* of stream data rates under a certain placement plan is constructed. Mathematically, the feasible set corresponds to the (nonnegative) space under n *node hyperplanes*, where n is the number of nodes and the i-th hyperplane consists of all points that render node i fully loaded.

Adaptation Rules and Objectives. Machine learning approaches can be used to automatically generate or improve adaptation rules based on the feedback the system is providing following their execution [25, 26]. The approach in [27] achieves optimization and adaptation of service compositions, which can arguably also be applied to the monitoring topology deployed in our approach. In contrast to [27], which takes a cost-centric viewpoint, in this work we target fast reactions, timely adaptations, and low network overhead. Adaptation rules based on the event-condition-action (ECA) [28] scheme are a popular technique used to control systems. However, for some complex systems the enumeration of all conditions, e.g., all possible types of failures, is often impracticable. Also, the actions to recover the system can become too tedious to be specified manually. Automated planning allows to automatically compute plans on top of a knowledge base following predefined objectives, and helps to enable goal-driven management of computer systems [29, 30].

Dynamic Reconfiguration and Redeployment. Facilities for dynamic reconfiguration and redeployment of monitoring infrastructures is at the heart of our approach. Srivastava et al. [31] present an approach for minimizing network usage and managing resource consumption in data aquisition networks by moving query operators. An elastic approach for optimal CEP query operator placement using cloud computing techniques is presented in [7]. As part of an optimization algorithm, the approach achieves a tradeoff between load distribution, duplicate event buffering and inter-node data traffic, also taking into account the costs of migration. The work also tackles the technical challenge of migrating stateful operators between infrastructure nodes. On the implementation level, dynamic deployment of hosts is typically achieved using tailor-made

DOI: 10.1002/spe.2254

automation scripts, which are prone to errors that could potentially leave part of the system in an intermediate or undesired state. Since reliable (re-)configuration is imperative to our approach, we utilize upfront testing techniques to increase the repeatability and reliability of deployment automations [32].

8. CONCLUSION

In this paper we introduce an architecture and a domain-specific language that allow to integrate functionality provided by different components and to define monitoring and adaptation functionality. We assume that monitoring is carried out by complex-event processing queries, while adaptation is performed by condition action rules performed on top of a distributed knowledge base. However, our approach can be applied to other forms of control mechanisms with dependencies among functionality blocks. Furthermore, we discuss implementation characteristics of the currently realized prototype based on the proposed architecture.

In future work we will present extensive experiments in order to quantify the characteristics of the implemented approach, and assess deployment performance relative to the size of infrastructure and elements to deploy. We also plan to integrate continuous deployment techniques, i.e., the capability to migrate elements at runtime to adapt according to more precise knowledge and changing environments. Furthermore, we aim to integrate the presented framework with current cloud management tools, such as OpenStack Heat [33] or a middleware-based meta cloud [34] approach.

ACKNOWLEDGEMENTS

This research has received funding from the European Commission's Seventh Framework Program [FP7/2007-2013] under grant agreement 257483 (Indenica), as well as from the Austrian Science Fund (FWF) under grant P23313-N23 (Audit 4 SOAs).

REFERENCES

1. Zang C, Fan Y. Complex event processing in enterprise information systems based on RFID. *Enterprise Information Systems* 2007; **1**(1):3–23, doi:10.1080/17517570601092127.
2. Mühl G, Fiege L, Pietzuch PR. *Distributed event-based systems*. Springer, 2006.
3. Inzinger C, Satzger B, Hummer W, Dustdar S. Specification and deployment of distributed monitoring and adaptation infrastructures. *Service-Oriented Computing - ICSOC 2012 Workshops*, *LNCS*, vol. 7759. Springer Berlin Heidelberg, 2012; 167–178, doi:10.1007/978-3-642-37804-1_18.
4. (W3C) WWWC. Soap version 1.2 part 1: Messaging framework (second edition). http://www.w3.org/TR/soap12-part1 2007. Accessed: 2013-10-01.
5. Vinoski S. Advanced message queuing protocol. *IEEE Internet Computing* 2006; **10**(6):8789.
6. Eclipse Foundation. Xtext Documentation. URL http://www.eclipse.org/Xtext/documentation.html, [22 October 2013].
7. Hummer W, Leitner P, Satzger B, Dustdar S. Dynamic migration of processing elements for optimized query execution in event-based systems. *On the Move to Meaningful Internet Systems*, Springer Berlin Heidelberg, 2011; 451–468, doi:10.1007/978-3-642-25106-1_3.
8. EsperTech. Esper Reference Documentation. URL http://esper.codehaus.org/esper/documentation/documentation.html, [22 October 2013].
9. JBoss Drools team. Drools Expert User Guide. URL http://docs.jboss.org/drools/release/5.5.0.Final/drools-expert-docs/html_single/index.html, [22 October 2013].
10. Bazaraa MS, Sherali HD, Shetty CM. *Nonlinear Programming: Theory and Algorithms*. 2nd edn., Wiley, 2006.
11. Gurobi Optimization, Inc. Gurobi Optimizer Reference Manual. URL http://www.gurobi.com, [22 October 2013].
12. Armbrust M, Fox A, Griffith R, Joseph AD, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I, *et al.*. A view of cloud computing. *Communications of the ACM* 2010; **53**(4):50–58.
13. Apache Software Foundation. ActiveMQ. URL http://activemq.apache.com, [22 October 2013].
14. Apache Software Foundation. Tuscany SCA. URL http://tuscany.apache.org, [22 October 2013].
15. Chen Y, Iyer S, Liu X, Milojicic D, Sahai A. SLA decomposition: Translating service level objectives to system level thresholds. *Proceedings of the Fourth International Conference on Autonomic Computing*, IEEE, 2007; 3, doi:10.1109/ICAC.2007.36.

16. Comuzzi M, Kotsokalis C, Spanoudakis G, Yahyapour R. Establishing and monitoring SLAs in complex service based systems. *IEEE International Conference on Web Services*, ICWS'09, 2009; 783–790, doi:10.1109/ICWS.2009.47.
17. Jung G, Joshi KR, Hiltunen MA, Schlichting RD, Pu C. Generating adaptation policies for multi-tier applications in consolidated server environments. *International Conference on Autonomic Computing*, ICAC'08, 2008; 23–32, doi:10.1109/ICAC.2008.21.
18. Cardellini V, Casalicchio E, Grassi V, Lo Presti F, Mirandola R. Qos-driven runtime adaptation of service oriented architectures. *European software engineering conference/ACM SIGSOFT symposium on the foundations of software engineering*, ESEC/FSE'09, ACM, 2009, doi:10.1145/1595696.1595718.
19. Lakshmanan G, Li Y, Strom R. Placement strategies for internet-scale data stream systems. *IEEE Internet Computing* 2008; **12**(6):50–60, doi:10.1109/MIC.2008.129.
20. Ahmad Y, Çetintemel U. Network-aware query processing for stream-based applications. *International conference on Very large data bases*, VLDB'04, VLDB Endowment, 2004; 456–467.
21. Pietzuch P, Ledlie J, Shneidman J, Roussopoulos M, Welsh M, Seltzer M. Network-aware operator placement for stream-processing systems. *International Conference on Data Engineering*, ICDE'06, IEEE, 2006; 49, doi:10.1109/ICDE.2006.105.
22. Schilling B, Koldehofe B, Rothermel K. Efficient and distributed rule placement in heavy constraint-driven event systems. *International Conference on High Performance Computing and Communications*, HPCC'11, IEEE, 2011; 355–364, doi:10.1109/HPCC.2011.53.
23. Xing Y, Zdonik S, Hwang J. Dynamic load distribution in the borealis stream processor. *International Conference on Data Engineering*, ICDE'05, IEEE, 2005; 791–802, doi:10.1109/ICDE.2005.53.
24. Xing Y, Hwang JH, Çetintemel U, Zdonik S. Providing resiliency to load variations in distributed stream processing. *International Conference on Very Large Data Bases*, VLDB'06, VLDB Endowment, 2006; 775–786.
25. Inzinger C, Satzger B, Hummer W, Leitner P, Dustdar S. Non-intrusive policy optimization for dependable and adaptive service-oriented systems. *Symposium on Applied Computing*, SAC'12, ACM, 2012; 504–510, doi:10.1145/2245276.2245373.
26. Inzinger C, Hummer W, Satzger B, Leitner P, Dustdar S. Towards identifying root causes of faults in service orchestrations. *International Symposium on Reliable Distributed Systems*, SRDS'12, IEEE, 2012; 404–405, doi:10.1109/SRDS.2012.78.
27. Leitner P, Hummer W, Dustdar S. Cost-based optimization of service compositions. *IEEE Transactions on Services Computing* 2013; **6**(2):239–251, doi:10.1109/TSC.2011.53.
28. Almeida EE, Luntz JE, Tilbury DM. Event-condition-action systems for reconfigurable logic control. *IEEE Transactions on Automation Science and Engineering* 2007; **4**(2):167–181, doi:10.1109/TASE.2006.880857.
29. Satzger B, Pietzowski A, Trumler W, Ungerer T. Using automated planning for trusted self-organising organic computing systems. *Autonomic and Trusted Computing*, *LNCS*, vol. 5060. Springer Berlin Heidelberg, 2008; 60–72, doi:10.1007/978-3-540-69295-9_7.
30. Satzger B, Kramer O. Goal distance estimation for automated planning using neural networks and support vector machines. *Natural Computing* 2012; **12**(1):87–100, doi:10.1007/s11047-012-9332-y.
31. Srivastava U, Munagala K, Widom J. Operator placement for in-network stream query processing. *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS'05, ACM, 2005; 250–258, doi:10.1145/1065167.1065199.
32. Hummer W, Rosenberg F, Oliveira F, Eilam T. Testing Idempotence for Infrastructure as Code. *ACM/IFIP/USENIX Middleware Conference*, 2013.
33. OpenStack Foundation. OpenStack Heat. URL `https://wiki.openstack.org/wiki/Heat`, [22 October 2013].
34. Satzger B, Hummer W, Inzinger C, Leitner P, Dustdar S. Winds of Change: From Vendor Lock-In to the Meta Cloud. *IEEE Internet Computing* 2013; **17**(1):69–73, doi:10.1109/MIC.2013.19.