Specification and Deployment of Distributed Monitoring and Adaptation Infrastructures

Christian Inzinger, Benjamin Satzger, Waldemar Hummer, and Schahram Dustdar

Distributed Systems Group, Vienna University of Technology
Argentinierstrasse 8/184-1, A-1040 Vienna, Austria
{lastname}@dsg.tuwien.ac.at, http://dsg.tuwien.ac.at

Abstract. This paper presents a new domain-specific language that allows to define integrated monitoring and adaptation functionality for controlling heterogeneous systems. We propose a mechanism for optimal deployment of the defined control operators onto available resources. Deployment is based on solving a quadratic programming problem, and helps to achieve minimized reaction times, low overhead, as well as scalable monitoring and adaptation.

Keywords: Monitoring, Adaptation, Complex Systems, Domain-Specific Language, Deployment, Operator Placement

1 Introduction

Efficient monitoring and adapation of large-scale heterogeneous systems, integrating a multitude of components, possibly from different vendors, is challenging. Huge amounts of monitoring data and sophisticated adaptation mechanisms in complex systems render centralized processing of control logic impractical. In highly distributed systems it is desirable to keep relevant monitoring and adaptation functionality as local as possible, to reduce traffic and to allow for timely reaction to changes.

In this paper 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 based on top of heterogeneous, distributed components. Using the introduced DSL we then outline the process of deploying the integration infrastructure, focusing on the efficient placement of monitoring and adaptation functionality onto available resources.

The remainder of this paper is structured as follows: In Section 2 we outline a motivating scenario that is used throughout the discussion of our contribution. Section 3 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 4. Relevant previous research is presented in Section 5. We conclude the paper in Section 6 and provide an outlook for future research directions.

2 Scenario

In this section we introduce a motivating scenario based on the Indenica[1] FP7 EU project. The project aims at providing methods for describing, deploying and managing disparate platforms based on a virtual service platform (VSP), which integrates and unifies their services. As the focus of this paper is on the deployment and runtime aspects of the developed approach, the reader is referred to the project website for further information.

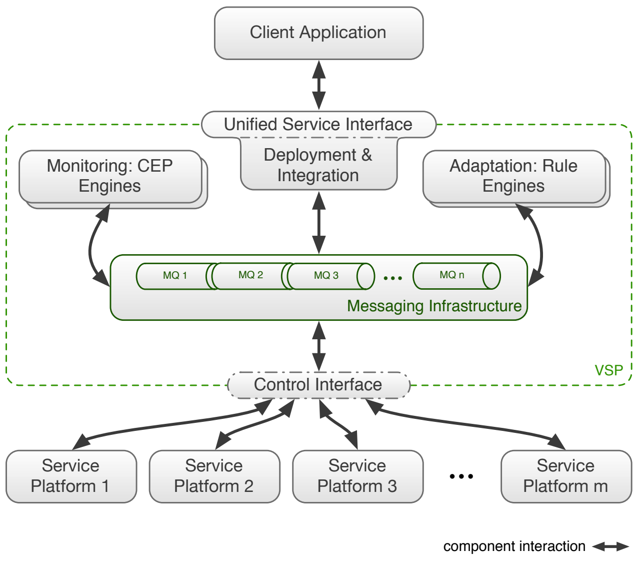


Fig. 1: INDENICA Runtime Architecture

The Indenica runtime architecture is presented in Fig. 1. The VSP provides a unified view on the functionality of the integrated service platforms, which are connected by control interfaces. Monitoring and adaptation are performed by complex event processing (CEP) engines and production rule engines, respectively. The former allows to implement monitoring by aggregating events emitted by service platforms, and the latter supports the definition of complex rules based on the gathered knowledge, which is stored as facts in a knowledge base. The execution of monitoring and adaptation on top of multiple engines allows for scalable control using distributed resources. Communication within the VSP is based on a distributed messaging infrastructure.

In this paper we introduce a novel DSL called MONINA, which allows the user to specify service platform capabilities, monitoring queries, and adaptation rules. In addition to that, we propose an algorithm to deploy the specified functionality onto available resources. Deployment aims at optimal usage of available resources considering locality, minimizing network overhead and taking load distribution into account.

[1] http://indenica.eu

3 MONINA Language

In this section we introduce MONINA[2] (Monitoring, Integration, Adaptation), a DSL allowing for concise and reusable specification of platforms integrated into a VSP, along with monitoring and adaptation rules governing their behavior.

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

event AverageProcessingTime {
  processing_time_ms : Integer
}

action DecreaseQuality {
  amount : Double
}

component ApplicationServer {
  emit RequestFinished
  action DecreaseQuality
  host vm1
  cost 32
}

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)
}

host vm1 { capacity 128 }
host vm2 { capacity 384 }
```

Listing 1: Sample system definition

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 'AggregateResponseTimes' query, which aggregates them over five minutes, creating an 'AverageProcessingTime' event. This event is converted to a fact, which might trigger '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 will discuss the most important language constructs of MONINA in more detail.

[2] Eclipse plugin available at `http://dsgvienna.bitbucket.org/indenica`

3.1 Event

Indenica follows an event-based approach. 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. 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`, and `Decimal`, and `Map<?,?>`.

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 3.4.

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.

3.2 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. Action declarations start with the **action** keyword followed by the action type identifier. Furthermore, actions can take parameters, modeled analogously to event attributes.

Similar to events, adaptation actions offered by known components do not need to be specified manually, but are automatically gathered from component specifications, which is further discussed in Section 3.4.

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 (analogously to event attributes): $a = (\pi_{\alpha_1}(a), \ldots, \pi_{\alpha_k}(a))$.

3.3 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. 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.

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 $e \in \mathcal{I}_E$ 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.

3.4 Component

A component declaration incorporates all information necessary to integrate third-party platforms into the Indenica infrastructure. 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 the component is deployed to.

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 Fig. 1.

Formally, components $c \in C$ are represented with the signature function[3] $sig : C \to \mathcal{P}(A) \times \mathcal{P}(\{(e_j, \nu_j) | e_j \in E, \nu_j \in \mathbb{R}_0^+\}) \times \mathbb{R}_0^+ \times H$ and the signature for a component c_i is $sig(c_i) \mapsto (I_i^A, \Omega_i^E, \psi_i, h_i)$. 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 the component is deployed to.

3.5 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.

[3] For clarity, we use the same symbol sig for signatures of components (Section 3.4), monitoring queries (Section 3.5), adaptation rules (3.6), and hosts (Section 3.7).

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 the 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. In addition to the query construct presented above, the language infrastructure allows for the integration of other CEP query languages, such as EQL[4] if necessary.

The set of queries $q_i \in Q$ is represented using the signature $sig : Q \to \mathcal{P}(E) \times \mathcal{P}(E) \times \mathbb{R}_0^+ \times \mathbb{R}_0^+$ and the signature for a query q_i is $sig(q_i) \mapsto (I_i^E, O_i^E, \rho_i, \psi_i)$. Input and output event stream types are denoted by I_i^E and O_i^E respectively, while ρ_i represents the ratio of input events processed to output events emitted, and ψ_i represents the processing cost of the query.

3.6 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. 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, which will be adjusted at runtime.

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[5] rule language, if more complex language constructs are required.

Formally, the set of rules $r_i \in R$ is represented with the signature function $sig : R \to \mathcal{P}(F) \times \mathcal{P}(A) \times \mathbb{R}_0^+$ and the signature for a rule r_i 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 I_i^F, while O_i^A representes the adaptation actions performed, and ψ_i represents the processing cost of the rule.

[4] `http://esper.codehaus.org`
[5] `http://jboss.org/drools`

3.7 Host

Hosts represent the physical infrastructure available for deployment of runtime elements, i.e., components, queries, and rules. 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.

The set of hosts $h_i \in H$ is represented with the signature function $sig : H \mapsto \mathbb{R}_0^+$ and the signature for a host h_i is $sig(h_i) \mapsto (\psi_i)$ with the capacity of a host represented by ψ_i.

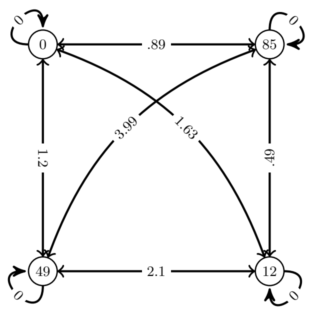
4 Deployment of Monitoring Queries and Adaptation Rules

In this section, we propose a methodology for efficiently deploying runtime elements for monitoring and adaptation, in line with the definitions provided in Section 3. Deployment is based on a MONINA definition. The deployment strategy attempts to find an optimal placement with regard to locality of information producers and consumers, resource usage, network load, and minimum 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 finds an optimal deployment scheme.

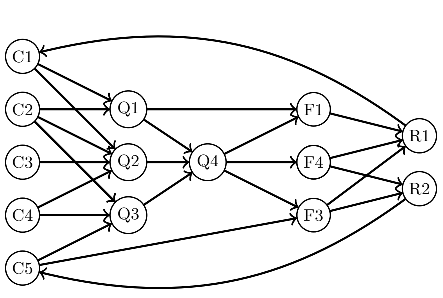
4.1 Infrastructure Graph

The infrastructure graph $G_I = (V_I, E_I)$ is a directed graph which models the available infrastructure. Its nodes (V_I) 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 ($E_I \subseteq V_I \times V_I$) represent the network connection between hosts. The node capacity function $c_I : V_I \to \mathbb{R}_0^+$ assigns 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 $w_I : E_I \to \mathbb{R}_0^+$ 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 2a shows an exemplary infrastructure graph.

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



(a) Infrastructure Graph



(b) Dependency Graph

Fig. 2: Graphs generated from a MONINA description

read by operating system tools to complement missing MONINA values. In Unix-based operating systems, for instance, the `/proc` pseudo-filesystem folder provides information about hardware and its utilization.

4.2 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 event transfer rate along an edge. 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 trigger rules 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 2b 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 $sig(c) = ((O^E, \bullet), \bullet, \psi)$ and $sig(q) = (I_i^E, \bullet, \bullet, \bullet)$. 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 $sig(c) = ((O^E, \bullet), \bullet, \psi)$ 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 $e \in (O^E \cap I^E)$, where $q_1 \neq q_2$, $sig(q_1) = (\bullet, O^E, \rho, \bullet)$ and $sig(q) = (I_i^E, \bullet, \bullet, \bullet)$.

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 $f \rightarrow r$ 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 $r \rightarrow c$ 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 $sig(c) = (I^A, \bullet, \bullet)$.

4.3 Quadratic Programming Problem Formulation

Quadratic programming [2] 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.

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

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 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 distribute 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 infrastructure 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 contraint 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 $c_D : V_D \rightarrow \mathbb{R}_0^+$ represents the costs of executing a runtime element v_D, as defined in the MONINA description.

We use Gurobi [5] for solving the optimization problem as described above. Due to space restrictions we cannot give results of runtime analyses in this paper. For problem sizes typically considered in INDENICA (5-15 hosts, 5-50 runtime elements) the optimization process usually takes less than 10 seconds on a regular laptop computer.

5 Related Work

The query operator placement problem in the context of complex event processing has received considerable attention in the past. To the best of our knowledge, none of these approaches explicitly consider adaptation and the according efficient placement of derived facts and adaptation rules. Also, operator networks usually are either trees or acyclic graphs, in contrast to our dependency graphs.

Some of the previous research is discussed in the following. An approach for minimizing network usage and managing resource consumption in sensor networks by moving query operators is presented in [14]. The work in [1] discusses algorithms for distributed placement of operator trees in wide area networks based on a distributed hash table structure. Another distributed operator placement for wide area networks using an overlay network is presented in [10], focussing on continuous cost optimization by migrating operators. The approach in [13] utilizes a decentralized algorithm based on negotiation between nodes for operator placement in heterogeneous CEP at runtime, minimizing the number of migrations. In [6], dynamic migration of processing elements is used as the basis for optimized multi-query execution in stream processing platforms.

Previous work in monitoring and adaptation of distributed heterogeneous systems deals with various aspects, such as establishing and monitoring SLAs (e.g., [4]), efficient rule generation (e.g., [9]), and adaptations based on QoS-requirements (e.g., [3]). However, there are no approaches that consider the efficient placement of monitoring and adaptation rules themselves, but rely on manual initial placement or human intervention.

Machine learning approaches can be used to automatically generate or improve adaptation rules based on the feedback the system is providing following their execution [8, 7]. Adaptation rules based on the condition-action 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 [12, 11].

6 Conclusion

In this paper we introduce a domain-specific language that allows 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. In future work we will present experiments in order to quantify the deployment performance relative to the size of infrastructure and elements to deploy. We also plan to integrate the capability to migrate elements at runtime to adapt according to more precise knowledge and changing environments.

Acknowledgement

The research leading to these results has received funding from the European Commission's Seventh Framework Programme [FP7/2007-2013] under grant agreement 257483 (Indenica).

References

1. Ahmad, Y., Çetintemel, U.: Network-aware query processing for stream-based applications. International Conference on Very Large Data Bases (VLDB'04) pp. 456–467 (2004)
2. Bazaraa, M.S., Sherali, H.D., Shetty, C.M.: Nonlinear Programming: Theory and Algorithms. Wiley, 2 edn. (2006)
3. Cardellini, V., Casalicchio, E., Grassi, V., Lo Presti, F., Mirandola, R.: Qos-driven runtime adaptation of service oriented architectures. In: European Software Engineering Conference (ESEC'09). ACM (2009)
4. Comuzzi, M., Kotsokalis, C., Spanoudakis, G., Yahyapour, R.: Establishing and Monitoring SLAs in Complex Service Based Systems. In: International Conference on Web Services (ICWS'09). pp. 783–790. IEEE (2009)
5. Gurobi Optimization, Inc.: Gurobi optimizer reference manual (2012), http://www.gurobi.com
6. 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 (OTM'11) pp. 451–468 (2011)
7. Inzinger, C., Hummer, W., Satzger, B., Leitner, P., Dustdar, S.: Towards Identifying Root Causes of Faults in Service Orchestrations. In: International Symposium on Reliable Distributed Systems (SRDS'12). IEEE (2012)
8. Inzinger, C., Satzger, B., Hummer, W., Leitner, P., Dustdar, S.: Non-Intrusive Policy Optimization for Dependable and Adaptive Service-Oriented Systems. In: Symposium on Applied Computing (SAC'12). pp. 504–510. ACM (2012)
9. Jung, G., Joshi, K.R., Hiltunen, M.A., Schlichting, R.D., Pu, C.: Generating Adaptation Policies for Multi-tier Applications in Consolidated Server Environments. In: International Conference on Autonomic Computing (ICAC'08). pp. 23–32 (2008)
10. Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., Seltzer, M.: Network-Aware Operator Placement for Stream-Processing Systems. In: International Conference on Data Engineering (ICDE'06). pp. 49–49. IEEE (2006)
11. Satzger, B., Kramer, O.: Goal distance estimation for automated planning using neural networks and support vector machines. Natural Computing, Springer (2012)
12. Satzger, B., Pietzowski, A., Trumler, W., Ungerer, T.: Using automated planning for trusted self-organising organic computing systems. In: International Conference Autonomic and Trusted Computing (ATC'08), pp. 60–72. Springer (2008)
13. 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) pp. 355–364 (2011)
14. Srivastava, U., Munagala, K., Widom, J.: Operator placement for in-network stream query processing. Symposium on Principles of Database Systems (PODS'05) pp. 250–258 (2005)