www.ascens-ist.eu

ASCENS

Autonomic Service-Component Ensembles

D4.5: Methods for Performance Monitoring and Prediction of SCs and SCEs

Grant agreement number: **257414**
Funding Scheme: **FET Proactive**
Project Type: **Integrated Project**
Latest version of Annex I: **Version 2.2 (30.7.2011)**

Lead contractor for deliverable: **CUNI**
Author(s): **Lubomír Bulej (CUNI), Tomáš Bureš (CUNI), Ilias Gerostathopoulos (CUNI), Vojtěch Horký (CUNI), Jaroslav Keznikl (CUNI), Michele Loreti (UNIFI), Lukáš Marek (CUNI), Petr Tůma (CUNI)**

Reporting Period: **3**
Period covered: **Oct 1, 2012 to Jun 30, 2013**
Submission date: **Oct 18, 2013**
Revision: **Final**
Classification: **PU**

Project coordinator: **Martin Wirsing (LMU)**
Tel: **+49 89 2180 9154**
Fax: **+49 89 2180 9175**
E-mail: **wirsing@lmu.de**

Partners: **LMU, UNIPI, UDF, Fraunhofer, UJF-Verimag, UNIMORE, ULB, EPFL, VW, Zimory, UL, IMT, Mobsya, CUNI**

SEVENTH FRAMEWORK PROGRAMME

Executive Summary

In this report we summarize the key results of the work performed in WP4 with respect to performance awareness of service components (SCs) and service-component ensembles (SCEs). First, we frame the adopted research approach, which focuses on smooth integration of performance awareness in the general project context by representing awareness as knowledge. Then we present the framework we have developed for convenient, non-intrusive monitoring of Java applications, based on a domain specific instrumentation language, DiSL, which allows monitoring through instrumentation with minimum overhead and no limitation on the nature of the instrumented code. Following, we detail the Stochastic Performance Logic (SPL), a powerful formalism that allows reasoning about performance while abstracting away the technicalities related to measurement and noise reduction. We also illustrate our approach towards performance awareness (evaluation and prediction) on an ad-hoc cloud scenario with mobile nodes forming collaboration ensembles. Finally, we describe the developed approach to link high-level performance requirements to implementation concerns and adaptation/reconfiguration strategies, embodied in the Invariant Refinement Method (IRM). We conclude by summarizing and sketching the items of future work.

Contents

1 Introduction

The ASCENS project deals with adaptive systems formed as ensembles of components that can possess and exchange knowledge. An important aspect of such systems is awareness, which requires that components are collecting knowledge about themselves and their environment. Such knowledge is ultimately used to guide autonomous adaptation of components and ensembles.

The many types of knowledge potentially collected by the components include information about performance. This may include information on the performance of the component itself (for example mean response time for components that react to external requests), information on the performance of other components (for example mean response time of external service provider components), or information on the performance of the environment (for example mean utilization of the execution platform).

This deliverable informs on the ongoing project effort to support performance awareness in components and ensembles.

1.1 Project Context

The effort to support performance awareness is centered in Task 4.4 of Work Package 4, the focus of the task is on dealing with the issues of performance monitoring and prediction. Relationship with the five research oriented work packages is as follows:

- Work Package 1: Language for modeling components and ensembles. The project relies on the SCEL language to describe components and ensembles for modeling purposes. Where performance awareness is concerned, there is a large conceptual distance between the predicates on observed performance that would be used in SCEL and the measurements that would be collected by monitoring executing systems – our work serves to reduce this distance so that the implementation of reasoning about performance can be close to the models that incorporate performance awareness.

- Work Package 2: Foundational models of autonomous ensembles. Due to their fundamental nature, the foundational models of autonomous ensembles also apply to performance awareness. As listed above, we help this inclusion by aligning the tools available for implementing performance awareness with the project modeling languages.

- Work Package 3: Knowledge representation. We structure the support for performance monitoring and prediction such that information about performance is presented as a particular kind of knowledge. Again, this helps apply the fundamental knowledge reasoning mechanisms to performance.

- Work Package 4: Adaptation mechanisms. The effort to support performance awareness belongs to this work package.

- Work Package 5: Behavior correctness. This work package is concerned mostly with behavior in the functional sense, in contrast with performance which is usually listed as a non-functional or extra-functional factor. If performance requirements are specified, our performance monitoring supports checking these requirements during execution.

Besides the five research oriented work packages, the project concentrates the effort on tool integration in Work Package 6. The tools related to supporting performance awareness are part of Work Package 6 and are being integrated with other project tools as technologically appropriate. As far as the case studies of Work Package 7 are concerned, experiments with performance awareness focus especially on the science cloud case study. This is in line with the project proposal, which seeks to focus the effort to deliver deeper results. The work on performance awareness also provides natural input to Work Package 8 on engineering best practices.

1.2 General Approach

An ensemble achieves awareness by observing both the state of its components and the state of its environment, deriving knowledge from thus collected information, and deciding how to act on this knowledge through reasoning. In each of these steps, many research activities of the ASCENS project deal with awareness in general, rather than focusing on a particular aspect such as location awareness or performance awareness - good examples of this generality are SOTA and SCEL, whose neutral character makes the project results more broadly applicable.

Rather than needlessly duplicating the general research activities with extra focus on performance awareness, our approach is to facilitate a smooth integration of performance awareness in the general awareness context. The integration starts with the need for observing performance – while many tools for performance monitoring exist, the dynamic nature of ensembles requires that we are able to start and stop monitoring performance of any component on demand, with managed overhead. Towards this goal, we work on dynamic instrumentation support. The next requirement of integration concerns the output of monitoring – typically, this output takes the form of a series of measurements listing function durations or event times, complete with noise and outliers due to interfering activities. Such output is difficult to use, we therefore work on a formalism that allows reasoning about performance while abstracting away the technical measurement details. The formalism is designed in a way that does not require differentiating between measurements of a real system and predictions of a high level model, which allows us to connect to the modeling activities of the ASCENS project whenever such models provide performance related output. We also integrate performance awareness with the high level ensemble concept – as a work in progress, we prepare support for ensembles whose members are selected based on observed performance.

The activities related to performance awareness are also well integrated with the project work on development lifecycle. In particular, an approach to derive design alternatives from high level performance requirements is introduced.

1.3 Report Structure

In Section 2, we briefly describe the instrumentation support based on the DiSL [MZA$^+$12] domain specific aspect language and framework. In Section 3, we focus on SPL [BBK$^+$12], a many sorted logic for reasoning about performance observed through measurements. Section 4 explains how the technology from the previous two sections gets integrated into a complete solution and Section 5 outlines the solution on an executing system. Finally, Section 6 illustrates the IRM method for ensemble-based component systems design based on invariant refinement.

2 Instrumentation for Performance Monitoring

The ability of ensembles to reason about the performance of the constituent components or the surrounding environment requires support for performance monitoring with particular dynamic properties. To avoid limiting the reasoning process, we must be able to monitor performance at any location that the reasoning process can consider. At the same time, we must avoid continuous monitoring of many locations, which would induce high overhead and therefore unduly influence the ensemble behavior.

To meet both requirements, we have developed performance monitoring with dynamic instrumentation that can be inserted and removed on demand. Due to a highly technical nature of the various issues the instrumentation has to solve, we limit our implementation to the Java environment – this means we can support not only the science cloud case study, but also the simulation prototypes in the other case studies that use the jRESP or jDEECo frameworks.

At a glance, Java provides several technologies for performance monitoring, each with a particular set of advantages and limitations.

The ***JVM Tool Interface (JVMTI)*** is a powerful native interface used for monitoring, debugging, profiling and similar application analyses. The interface provides access to predefined JVM events related to field accesses, method invocations, class loading and synchronization. It also provides a variety of functions to introspect running Java process together with the rest of JVM. Even though the interface is powerful, the necessity to develop a specialized native agent limits its applicability.

Another option for application monitoring is offered by the ***java.lang.instrument*** API, a Java bytecode instrumentation interface. The instrumentation interface provides class-loading hooks that allow instrumenting an application using a custom Java agent. The code to be instrumented is delivered to the instrumentation hook as an array of bytes ; the agent is responsible for parsing the class and modifying the bytecode. As a limitation, the approach is not able to instrument the entire Java class library.

Both instrumentation interfaces provide hooks for class loading and reinstrumentation of already loaded classes, however, Java itself does not contain any additional instrumentation support, such as a bytecode manipulation library or a domain specific language for describing instrumentation.

Java also provides a standard interface for delivering performance data to applications, based on the ***Java Management Extensions (JMX)***. In the JMX framework, applications are monitored and controlled through specialized application objects called managed beans, MBeans for short. Each MBean controls a component (resource) of an application, providing functions to read the component state, set and get the component configuration, and register event notification listeners. MBeans are suitable for continuous remote monitoring where only a small amount of data is transferred, however, the JMX framework provides no support for data gathering that needs to occur inside MBeans.

To combine the listed technologies in a robust instrumentation solution, we develop and utilize DiSL [MZA$^+$12], a domain specific language and framework allowing to conveniently monitor an application using instrumentation. DiSL is inspired by the Aspect Oriented Programing paradigm. We use DiSL to specify and execute the performance monitoring code, whose output events are processed by the custom performance logic framework described in Section 3.

Listing 1 illustrates a simple method invocation profiling code written in DiSL. The responsibility of the profiling code is to sample the time before and after a method invocation and print the method duration after the invocation (in real monitoring code, the duration is recorded rather than printed).

Method entry time sampling is done in the *onMethodEntry* method. DiSL is guided by the *@Before* annotation to insert the entire body of *onMethodEntry* at the beginning of each monitored method. Similarly, the method exit time sampling and method duration output in *onMethodExit* is inserted at the end of each monitored method. The use of DiSL removes the need for manual bytecode instrumentation, as well as complex constructs that would otherwise be necessary to handle for example exceptional method exits.

DiSL is capable of monitoring not only invocations of methods, but also execution of arbitrary blocks of code. The monitored block is defined by a construct called *Marker*. A predefined library

Listing 1 Simple method invocation profiling in DiSL

```
public class SimpleProfiler {

  @SyntheticLocal
  static long entryTime;

  @Before(marker=BodyMarker.class)
  static void onMethodEntry() {
    entryTime = System.nanoTime();
  }

  @After(marker=BodyMarker.class)
  static void onMethodExit(MethodStaticContext msc) {
    long exitTime = System.nanoTime();
    System.out.println(msc.thisMethodFullName()
      + " duration is "
      + (exitTime − entryTime));
  }
}
```

of *Markers* allows to monitor basic block executions, invocations of try-catch handlers, array or field assignments and others. DiSL further allows extending *Markers* to capture arbitrary code patterns.

DiSL contains specialized *SyntheticLocal* and *ThreadLocal* variables to allow efficient communication between related events. A *SyntheticLocal* variable is used in Listing 1 for passing sampled time data between the monitoring code within one method.

To access additional event context information, DiSL introduces constructs called *StaticContext* and *DynamicContext*. *StaticContext* exposes information about location like method or class name. *DynamicContext* allows to access dynamic information like field or variable values. In the listed example, *MethodStaticContext* is used to retrieve a name of the profiled method.

Insertion of monitoring code can be restricted using two mechanisms, *Scope* and *Guards*. *Scope* is a simple language construct allowing to define patterns restricting methods and classes to be instrumented. *Guards* are standard Java classes allowing to evaluate complex instrumentation conditions.

As a vital property from the performance monitoring perspective, DiSL does not insert any additional code besides snippets into the observed application, and therefore does not create any hidden overhead. The code of events is prevented from modifying the control flow of the observed application and the instrumentation does not violate the JVM hotswapping rules. As a result, the monitoring code can be dynamically inserted and removed during application execution. Finally, DiSL has very few limitations on which code can be instrumented, making it possible to monitor any arbitrary location in both the application components and the Java class library.

3 Expressing Performance Properties

In its raw form, the monitoring output contains records of performance relevant events, such as times when particular requests or responses were observed, or execution durations of particular methods. Further processing of the monitoring output depends on the context. For example, an application that needs to be aware of SLA violations would count those request processing times that exceed a given threshold, including potential outliers. In contrast, an application that needs to adapt an algorithm for the current processor would look for minimum or median algorithm execution times, removing outliers.

To provide a suitable level of abstraction for processing the monitoring output, we introduce a formalism where the performance measurements are represented as observations of random variables and operators allow comparing measurements in a statistically rigorous manner, depending on the adopted interpretation. The formalism is called Stochastic Performance Logic (SPL) [BBK$^+$12].

3.1 Stochastic Performance Logic

We illustrate the SPL concepts on an example of two methods whose performance needs to be related to each other – this example finds an application in systems that adapt by choosing the faster of two method implementations or the faster of two execution platforms.

We formally define the performance of a method as a random variable representing the time it takes to execute the method with random input parameters drawn from a particular distribution. The nature of the random input is formally represented by *workload class* and *method workload*. The workload is parametrized by *workload parameters*, which capture the dimensions along which the workload can be varied, e.g. array size, matrix sparsity, number of vertices in a graph, etc.

Definition 3.1 Workload class *is a function $\mathfrak{L} : P^n \to (\Omega \to I)$, where for a given $\mathfrak{L}$, P is a set of* workload parameter *values, n is the number of parameters, Ω is a sample space, and I is a set of objects (method input arguments) in a chosen programming language.*

For later definitions we also require that there is a total ordering over P.

Definition 3.2 Method workload *is a random variable $L^{p_1,\dots,p_n}$ such that $L^{p_1,\dots,p_n} = \mathfrak{L}(p_1, \dots, p_n)$ for a given workload class $\mathfrak{L}$ and parameters $p_1, \dots, p_n$.*

Unlike conventional random variables that map observations to a real number, method workload is a random variable that maps observations to object instances, which serve as random input parameters for the measured method. If necessary, the developer may adjust the underlying stochastic process to obtain random input parameters representing domain-specific workloads, for example partially sorted arrays.

To demonstrate the above concepts, let us assume we want to measure the performance of a method S, which sorts an array of integers. The input parameters for the sort method S are characterized by workload class $\mathfrak{L}_S : \mathbb{N}^+ \to (\Omega_S \to I_S)$. Let us assume that the workload class $\mathfrak{L}_S$ represents an array of random integers, with a single parameter determining the size of the array. The method workload returned by the workload class is a random variable, whose observations are instances of random arrays of given size. For example, method inputs in form of random arrays of size 1000 will be obtained from observations of random variable $L_S^{1000} : \Omega_S \to I_S = \mathfrak{L}_S(1000)$.

Note that without loss of generality, we assume in the formalization that there is exactly one $\mathfrak{L}_M$ for a particular method M and that M has just one input argument.

With the formal representation of a workload in place, we now proceed to define the method performance.

Definition 3.3 *Let $M(in)$ be a method in a chosen programming language and $in \in I$ its input argument. Then method performance $P_M : P^n \to (\Omega \to \mathbb{R})$ is a function that for given workload parameters $p_1, \dots, p_n$ returns a random variable, whose observations correspond to execution duration of method M with input parameters obtained from observations of $L_M^{p_1,\dots,p_n} = \mathfrak{L}_M(p_1, \dots, p_n)$, where $\mathfrak{L}_M$ is the workload class for method M.*

To facilitate comparison of method performance, SPL is based on regular arithmetics, in particular on axioms of equality and inequality adapted for the method performance domain.

Definition 3.4 *SPL is a many-sorted first-order logic defined as follows:*

- *There is a set $FunPe$ of function symbols for method performances with arities $P^n \to (\Omega \to \mathbb{R})$ for $n \in \mathbb{N}^+$.*
- *There is a set $FunT$ of function symbols for performance observation transformation functions with arity $\mathbb{R} \to \mathbb{R}$.*
- *The logic has equality and inequality relations $=, \leq$ for arity $P \times P$.*
- *The logic has equality and inequality relations $\leq_{p(tl,tr)}$, $=_{p(tl,tr)}$ with arity $(\Omega \to \mathbb{R}) \times (\Omega \to \mathbb{R})$, where $tl, tr \in FunT$.*
- *Quantifiers (both universal and existential) are allowed only over finite subsets of P.*
- *For $x, y, z \in P$ and $P_M, P_N \in FunPe$, the logic has the following axioms:*

$$x \leq x \tag{1}$$
$$(x \leq y \wedge y \leq x) \leftrightarrow x = y \tag{2}$$
$$(x \leq y \wedge y \leq z) \to x \leq z \tag{3}$$
$$\begin{gathered}\textit{For each pair } tl, tr \in FunT \textit{ such that}\\ \forall o \in \mathbb{R} : tl(o) \leq tr(o), \textit{there is an axiom}\\ P_M(x_1, \ldots, x_m) \leq_{p(tl,tr)} P_M(x_1, \ldots, x_m)\end{gathered} \tag{4}$$
$$\begin{gathered}(P_M(x_1, \ldots, x_m) \leq_{p(tm,tn)} P_N(y_1, \ldots, y_n) \wedge\\ P_N(y_1, \ldots, y_n) \leq_{p(tn,tm)} P_M(x_1, \ldots, x_m)) \leftrightarrow\\ P_M(x_1, \ldots, x_m) =_{p(tm,tn)} P_N(y_1, \ldots, y_n)\end{gathered} \tag{5}$$

Axioms (1)–(3) come from arithmetics, since workload parameters (P) are essentially real or integer numbers. In analogy to (1)–(2), axiom (4) may be regarded as generalised reflexivity, and axiom (5) shows the correspondence between $=_p$ and $\leq_p$. An analogy of (3), i.e. transitivity, cannot be introduced for $=_p$ and $\leq_p$, because it does not hold for all interpretations of SPL.

Note that even though we currently do not make use of the axioms in our approach, they make the properties of the logic more obvious (in particular the performance relations $=_p$ and $\leq_p$). Specifically, the lack of transitivity for performance relations ensures that SPL formulas can only express statements that are consistent with hypothesis testing approaches used in the SPL interpretation.

Using the logic defined above, we would like to express assumptions about method performance in the spirit of the following examples:

Example 3.1 *"On arrays of 100, 500, 1000, 5000, and 10000 elements, the sorting algorithm A is at most 5% faster and at most 5% slower than sorting algorithm B."*

$$\begin{gathered}\forall n \in \{100, 500, 1000, 5000, 10000\} :\\ P_A(n) \leq_{p(id,\lambda x.1.05x)} P_B(n) \wedge P_B(n) \leq_{p(id,\lambda x.0.95x)} P_A(n)\end{gathered}$$

Example 3.2 *"On buffers of 256, 1024, 4096, 16384, and 65536 bytes, the Rijndael encryption algorithm is at least 10% faster than the Blowfish encryption algorithm and at most 200 times slower than* *array copy.”*

$$\begin{aligned}&\forall n \in \{256, 1024, 4096, 16384, 65536\} :\\ &P_{Rijndael}(n) \leq_{p(id,\lambda x.0.9x)} P_{Blowfish}(n) \land\\ &P_{Rijndael}(n) \leq_{p(id,\lambda x.200x)} P_{ArrayCopy}(n)\end{aligned}$$

For compact in-place representation of performance observation transformation functions, we use the lambda notation [Bar84], with id as a shortcut for identity, $id = \lambda x.x$.

3.2 Logic Interpretations

To ensure correspondence between the SPL formulas in Examples 3.1 and 3.2 and their textual description, we need to introduce the appropriate semantic that provides the intended SPL interpretation. The interpretation described in [BBK$^+$12] is a *sample-based* interpretation – the basic idea is to use statistical hypothesis testing to evaluate the $\leq_p$ and $=_p$ relations in the formulas.

To formulate the sample-based interpretation, we first need to fix the set of observations for which the relations will be interpreted. We therefore define an *experiment*, denoted $\mathcal{E}$, as a finite set of observations of method performances under a particular method workload.

Definition 3.5 *Experiment $\mathcal{E}$ is a collection of $\mathcal{O}_{P_M(p_1,\ldots,p_m)}$, where $\mathcal{O}_{P_M(p_1,\ldots,p_m)} = \{P^1_M(p_1,\ldots,p_m), \ldots, P^V_M(p_1,\ldots,p_m)\}$ is a set of V observations of method performance P_M subjected to workload $L^{p_1,\ldots,p_m}_M$, and where $P^i_M(p_1,\ldots,p_m)$ denotes i-th observation of performance of method M.*

For a particular experiment, we define the sample-based interpretation of SPL.

Definition 3.6 *Let $tm, tn : \mathbb{R} \to \mathbb{R}$ be performance observation transformation functions, P_M and P_N be method performances, $x_1,\ldots,x_m, y_1,\ldots,y_n$ be workload parameters, and $\alpha \in \langle 0, 0.5\rangle$ be a fixed significance level.*

For a given experiment $\mathcal{E}$, the relations $\leq_{p(tm,tn)}$ and $=_{p(tm,tn)}$ are interpreted as follows:

- $P_M(x_1,\ldots,x_m) \leq_{p(tm,tn)} P_N(y_1,\ldots,y_n)$ *iff the null hypothesis*

$$\mathrm{H}_0 : E(tm(P^i_M(x_1,\ldots,x_m))) \leq E(tn(P^j_N(y_1,\ldots,y_n)))$$

  cannot be rejected by one-sided Welch's t-test [Wel47] at significance level α based on the observations gathered in the experiment $\mathcal{E}$;

- $P_M(x_1,\ldots,x_m) =_{p(tm,tn)} P_N(y_1,\ldots,y_n)$ *iff the null hypothesis*

$$\mathrm{H}_0 : E(tm(P^i_M(x_1,\ldots,x_m))) = E(tn(P^j_N(y_1,\ldots,y_n)))$$

  cannot be rejected by two-sided Welch's t-test at significance level 2α based on the observations gathered in the experiment $\mathcal{E}$;

where $E(tm(P^i_M(\ldots)))$ and $E(tn(P^j_N(\ldots)))$ denote the mean value of performance observations transformed by function tm or tn, respectively.

Briefly, the Welch's t-test rejects with significance level α the null hypothesis $\overline{X} = \overline{Y}$ against the alternative hypothesis $\overline{X} \neq \overline{Y}$ if

$$\left| \frac{\overline{X} - \overline{Y}}{\sqrt{\frac{S^2_X}{V_X} + \frac{S^2_Y}{V_Y}}} \right| > t_{\nu,\alpha/2}$$

and rejects with significance level α the null hypothesis $\overline{X} \leq \overline{Y}$ against the alternative hypothesis $\overline{X} > \overline{Y}$ if

$$\frac{\overline{X} - \overline{Y}}{\sqrt{\frac{S_X^2}{V_X} + \frac{S_Y^2}{V_Y}}} > t_{\nu,\alpha}$$

where V_i is the sample size, S_i^2 is the sample variance, $t_{\nu,\alpha}$ is the $(1-\alpha)$-quantile of the Student's distribution with ν levels of freedom, with ν computed as follows:

$$\nu = \frac{\left(\frac{S_X^2}{V_X} + \frac{S_Y^2}{V_Y}\right)^2}{\frac{S_X^4}{V_X^2(V_X-1)} + \frac{S_Y^4}{V_Y^2(V_Y-1)}}$$

Although Welch's t-test formally requires normal distribution of X and Y, it is robust to violations of normality due to the Central Limit Theorem.

Given a set of observations of method performances (i. e. random variables), the interpretation determines whether the mean values of the observed method performances are in a particular relation (i. e., $\leq_p$ or $=_p$). The sample-based interpretation is reasonable for situations where it is possible to collect a relatively large number of samples to be used for the statistical testing (experience suggests tens of thousands of samples suffice). When SPL is used to make adaptation decisions at runtime, the number of collected samples might be smaller by several orders of magnitude. In such situations, other interpretations can be more suitable (e.g. based on different statistical tests or even based on comparing simple statistics such as floating average).

So far, we have limited our discussion of SPL use to comparing method execution times. SPL, however, provides considerable freedom as far as the input data is concerned. In particular, we can compare metrics such as system load – the system load is typically represented as the number of ready threads; if this number is normalized to the number of processors, it can be used as a criterion in a distributed environment for finding the least loaded machine. The formula for deciding whether machine A is less loaded than machine B then remains rather simple, $L_A < L_B$.

When comparing system load values, we can do with a simple SPL interpretation, because both L_A and L_B are, in fact, scalars and plain comparison can therefore be used. On the other hand, if multiple observations of load are available, this comparison can be seen as an SPL formula with sample-based interpretation and evaluated as such. We note that the two cases differ – one is concerned with the current system load, one evaluates the mean system load over a longer time period. There are other practical differences – for example, when a new observation arrives, evaluating the formula with sample-based interpretation is more resource intensive than evaluating with plain comparison. For environments with restricted resources, this can be a key factor.

Our current experiments investigate whether other interpretations are beneficial for particular adaptation scenarios. Given that most statistical tests have very specific requirements on the nature of the data they use, one of the possibilities is to introduce freedom to use a different statistical test for evaluating each individual relation. If the user knows that the samples conform to particular constraints, a more powerful specialized test can be used to achieve greater test sensitivity or reliability. And vice versa – if there is little knowledge of the sample properties, one may choose a less powerful nonparametric test. Naturally, such changes to SPL require proving that the basic SPL axioms hold for the chosen interpretation.

Besides investigating different interpretations, we also work on extending our SPL framework to exploit the freedom of using various data as formula inputs. In an SPL formula, the data is abstracted as a random variable and it is up to the user to provide specific values for such variable – these can be the method execution times or system load values already used in the examples, but also workload related values such as request frequencies (for systems that seek to adapt to workload changes) or platform related values such as power consumption measurements (for systems that seek to optimize battery lifetime). In the following, we use the term *data source* to refer to the provider of specific data.

Introducing data sources as a new concept connected to the random variables provides an important level of abstraction in a system designed and implemented using SPL formulas. The same SPL formula can be used through multiple software development phases from modeling to implementation – the only difference is what data sources would be *bound* to the actual random variables in the formula.

We illustrate the concept on an example of an adaptive application. Consider a problem that can be solved using two different algorithms, A_1 and A_2. One of the algorithms performs better on large input sizes, the other on smaller ones. Theoretically, the boundary between the input sizes where one or the other algorithm performs better is quite precise, but in reality it depends on factors such as the technical details of the execution platform. The decision which algorithm to use therefore cannot be hardcoded into the application. Instead, a trivial SPL formula can be used to desribe the condition for selecting particular algorithm for given input size – we use A_1 if:

$$A_1(n) \leq A_2(n) \tag{6}$$

where n denotes the input size, A_2 is used otherwise.

In the early application design phase, modeling would be used to assess the application behavior – and because data on actual performance is not available at that stage, the model would bind data sources that rely on the algorithm complexity analysis to formula (6). In the testing phases of the application development process, the same formula would be bound to data sources that measure the performance of the (already implemented) algorithms in the potentially restricted testing environment. The formula would be used as a base indicator that the implementation works as expected. Where needed, artificial data injection (similar to fault injection) through the same data sources could be used to test corner cases. Finally, at runtime, the same formula would be bound to data sources collecting runtime measurements, allowing the application to adapt itself to the hardware it is running on.

Extending SPL with the concept of pluggable data sources allows us to reuse the same formulas in a broad range of software development phases, from the early design phases where reasoning about performance is done at the model level, to the testing and runtime phases where actual data is used. At this stage of the project, we are including the pluggable data sources in the prototype implementation of our SPL framework and preparing experiments with the science cloud case study.

4 Integrating Monitoring and Evaluation

As two major building blocks of the support for performance awarness, we have described the use of dynamic instrumentation for performance monitoring in Section 2 and the formalism for expressing and evaluating conditions over measurements in Section 3. This section briefly outlines the steps necessary to integrate the two building blocks.

4.1 Language Integration Support

Writing applications with performance awareness requires that the concepts presented earlier are integrated into the particular programming languages used to develop the application. In the ASCENS project context, Java is an obvious choice – it is used in the case studies and in the prototype tools, it is also a well-known multi-platform language – in particular, it is the language used by the jRESPand jDEECoframeworks, as well as the Science Cloud Platform. We note, however, that the choice of Java is without loss of generality – in principle, most methods developed in the ASCENS project are implementation-language-agnostic. Developing the language integration support poses novel challenges, however, replicating the effort for multiple implementation environments would not be beneficial.

To indicate requirements on performance, SPL formulas are attached in the form of annotations to one of the measured methods, as outlined in the example in Listing 2. The annotations are useful

Listing 2 Java annotations expressing performance requirements

```
@SPL(
  methods = "javaSort=java.util.Arrays#sort(long[])",
  generators = "data=SPL:LongUniform('0;1000')",
  formula = "for ( i {100, 1000, 10000} ) SELF[data](i) <=(2, 1) javaSort[data](i)"
)
public void fasterSort(long[] data) {
  // Measured method ...
}
```

The formula states that *fasterSort* should be at least two times faster than *javaSort*, a library implementation. The generator provides the workload that is being measured, that is, the objects used as arguments when calling the measured methods.

if the performance requirements are to be evaluated at development time or deployment time using external tools; it is also possible to include the formula evaluation in the component system runtime, where it can direct mechanisms related to component lifecycle or connector binding, as outlined in [BBH+12]. For measurement at development time, we have developed an Eclipse plugin that can display an overview of performance tests, such as the one in Figure 1, complete with an interactive visualization. For now, the plugin does not support runtime data collection and evaluation.

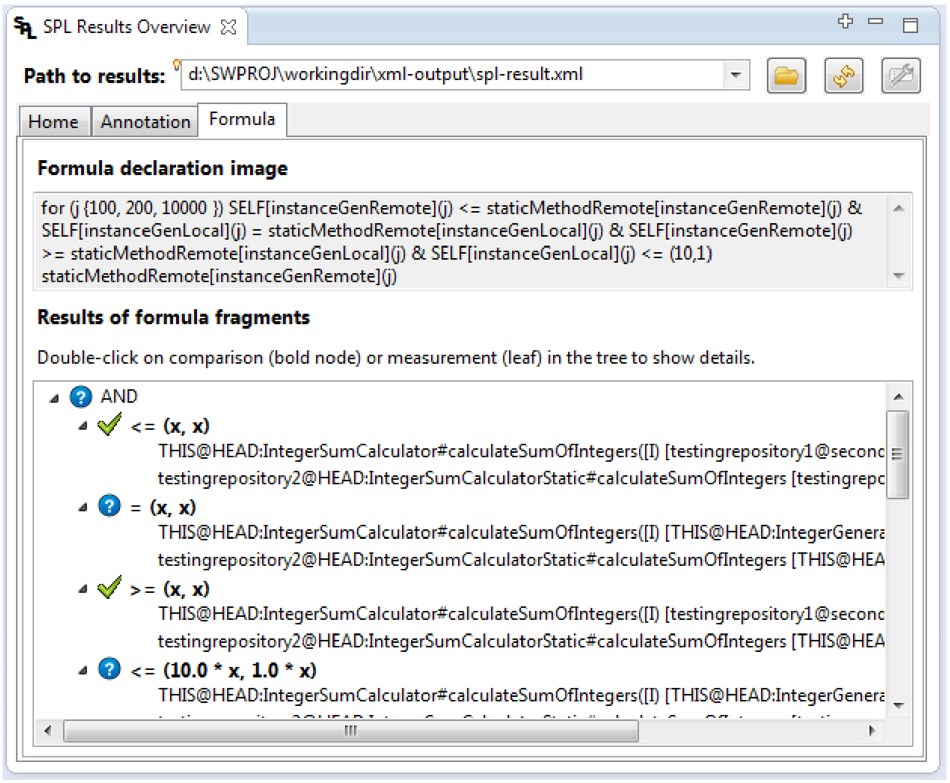
Besides the static language integration based on annotations, we have also designed an API for evaluating SPL formulas directly from application code at runtime. Central to the API are the *DataSource* and *Formula* interfaces that represent the pluggable data source and the SPL formula, respectively. The interfaces are displayed in Listing 3. The *DataSource* interface offers the *getStatisticSnapshot()* method that returns a consistent snapshot of the samples provided by the data source. The snapshot is independent of the data source, whose performance can change at any time. The individual data sources are attached to the *Formula* instance using the *bind()* method. Once all the variables are bound, the formula can be evaluated. For the evaluation, the snapshot is used.

Listing 3 Main SPL runtime evaluation API interfaces

```
public interface DataSource {
    StatisticSnapshot getStatisticSnapshot();
}

public interface Formula {
    void setInterpretation(Interpretation interpretation);
    void bind(String variable, DataSource data);
    Result evaluate();
}
```

The *Formula* interface can be used to specify the interpretation of the SPL formula used for the evaluation. This allows the developer to combine different interpretations inside a single application, choosing the most appropriate one as needed.



Path to results: d:\SWPROJ\workingdir\xml-output\spl-result.xml

Formula declaration image

```
for (j {100, 200, 10000 }) SELF[instanceGenRemote](j) <= staticMethodRemote[instanceGenRemote](j) &
SELF[instanceGenLocal](j) = staticMethodRemote[instanceGenLocal](j) & SELF[instanceGenRemote](j)
>= staticMethodRemote[instanceGenLocal](j) & SELF[instanceGenLocal](j) <= (10,1)
staticMethodRemote[instanceGenRemote](j)
```

Results of formula fragments

Double-click on comparison (bold node) or measurement (leaf) in the tree to show details.

```
AND
    <= (x, x)
        THIS@HEAD:IntegerSumCalculator#calculateSumOfIntegers([I) [testingrepository1@seconc
        testingrepository2@HEAD:IntegerSumCalculatorStatic#calculateSumOfIntegers [testingrepc
    = (x, x)
        THIS@HEAD:IntegerSumCalculator#calculateSumOfIntegers([I) [THIS@HEAD:IntegerGenera
        testingrepository2@HEAD:IntegerSumCalculatorStatic#calculateSumOfIntegers [THIS@HEA
    >= (x, x)
        THIS@HEAD:IntegerSumCalculator#calculateSumOfIntegers([I) [testingrepository1@seconc
        testingrepository2@HEAD:IntegerSumCalculatorStatic#calculateSumOfIntegers [THIS@HEA
    <= (10.0 * x, 1.0 * x)
        THIS@HEAD:IntegerSumCalculator#calculateSumOfIntegers([I) [THIS@HEAD:IntegerGenera
```

Figure 1: Eclipse plugin with performance test overview

In addition to the methods described above, the API contains support to prepare the data sources. These take the form of helper classes for manual measurement as well as the access to the automated instrumentation features. Listing 4 depicts a code fragment that checks whether a method execution time does not exceed the given threshold – where the example simply prints a warning, an adaptive application would take an action to remedy the problem.

Listing 4 Checking method execution time

```
/*
 * Preparation.
 * The SPL.instrument() creates the data source and also
 * adds the measuring code automatically to the measured
 * method.
 */
SourceData data = SPL.instrument("pkg.MyClass#myMethod");
Formula formula = SPL.createFormula("A < 100");
formula.bind("A", data);

/*
 * Check the formula (once it is clear that enough samples
 * were collected).
 */
if (formula.evaluate() == Result.FALSE) {
    logger.warn("myMethod is way too slow!");
}
```

4.2 Instrumenting Running Application

As explained, the *SPL.instrument()* method uses DiSL to instrument the running Java applications with code that measures the method execution time. The instrumentation is also performed on demand, whenever the need to measure a particular method arises – this happens when an annotation uses an SPL formula to refer to the method performance. In addition to plain Java, the prototype implementation includes support for OSGi applications, where the use of class loaders for component isolation poses additional technical challenges.

When an SPL formula refers to a frequently executed method, the number of measurements collected over time can grow rapidly. To avoid exhausting the available memory, we only store enough data to evaluate the particular formula. In some cases, this can be an aggregate statistic such as the sample average. Other situations require storing a limited number of measurements, but the number is always limited through configuration.

Depending on the need of the case studies, we are also considering a solution where each SPL formula is evaluated as close to the measurement locations as possible – in a distributed system, this may minimize the cost of transporting data from the measurement locations to the SPL formula evaluation engine.

4.3 Integrating Predictive Models

Obviously, a degree of performance awareness can be achieved entirely based on the knowledge of current and past performance. A simple example of this is a server that increases the number of threads in reaction to the observed response time using a simple rule – when the response time grows, more threads are added. The SPL formulas used to express this rule only need to rely on current and past measurements, provided by the appropriate data sources.

In some situations, performance awareness can augment the information about current and past events by using predictive models – following the example, it may be possible to use trend estimation methods to predict a rise in request frequency and adjust the number of service threads accordingly. This option is handled by the support for performance awareness by presenting predictive models as data sources – that way, the same SPL formulas that were used to react to current events can react to predicted behavior.

The integration of predictive models on an application migration example is exemplified in the next section, where the Planner may rely on modeling to pick a suitable deployment alternative. Relying on the modular solution with pluggable data sources simplifies the technical integration of such models in the SPL framework.

5 Performance Awareness Example

In this section we demonstrate how we imagine the performance awareness of an application. We demonstrate this on an ad-hoc cloud case study, where a smart phone would offload computationally intensive applications to the nearby available computational nodes to improve battery lifetime [BBHK13]. Along that we show how the models and techniques from previous sections are put into practice. To elaborate the application migration example, we use the jDEECo component model [BGH+13], which realizes the concepts of the SCEL formalism for developing adaptive ensembles. As a major feature, the ensembles consist of components that communicate exclusively through shared knowledge – we therefore include performance measurements (e.g., results of SPL formulas evaluated on data obtained with DiSL-based instrumentation) among the knowledge elements.

As a side note, including performance in knowledge is not entirely straightforward. In SCEL, the knowledge elements are exchanged between components throughout the lifetime of the ensemble – knowledge that is requested is assumed to exist and communicating it does not bring any additional cost. This does not apply to measurements expressed as performance knowledge – measurements are not collected before being requested, because measuring just to populate knowledge that may never be used would incur an excessive overhead. Once demanded, the instrumentation and measurement take certain time, and incur overhead on the observed rather than the observing component. Further penalty may be associated with communicating the measurements.

The example is also described in [BBHK13].

5.1 Scenario Description

We illustrate the performance awareness here on a restricted version of the ASCENS cloud case study. In particular, the scenario we consider is that of a user travelling in a train or a bus, who wants to do productive work using a tablet computer or review travel plans and accommodation. The tablet notes the presence of an offload server machine located in the bus itself, and to save battery, it offloads most computationally intensive tasks to that machine. Later, when the bus approaches its destination, the offload server notifies the tablet that its service will soon become unavailable and tasks will start moving back to the tablet. When the bus enters the terminal, the tablet will discover another offload server, provided by the terminal authority, and move some of its tasks to the newly found machine. The challenge is in predicting which deployment scenario will deliver the expected performance – that is, when is it worth offloading parts of the application to a different computer.

For our example, we assume that the application has a frontend that cannot be migrated (such as the user interface which obviously has to stay with the user, Af in our example) and a backend that can be offloaded (typically the computationally intensive tasks, Ab in our example). Figure 2 depicts the adaptation architecture (the used notation is that of component systems, except for interfaces which

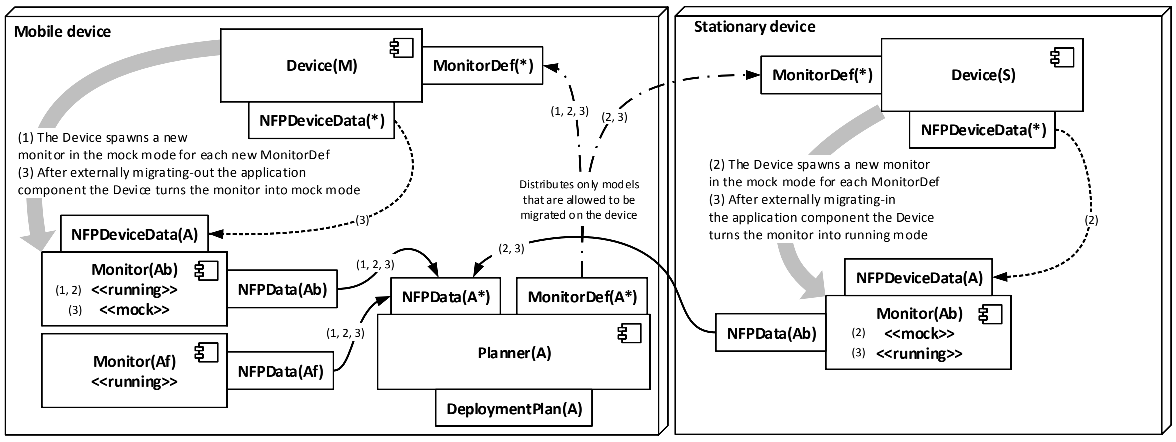

Mobile device
Device(M)
MonitorDef(*)
NFPDeviceData(*)
(1) The Device spawns a new monitor in the mock mode for each new MonitorDef
(3) After externally migrating-out the application component the Device turns the monitor into mock mode
NFPDeviceData(A)
Monitor(Ab)
(1, 2) <<running>>
(3) <<mock>>
NFPData(Ab)
Monitor(Af)
<<running>>
NFPData(Af)
NFPData(A*)
MonitorDef(A*)
Planner(A)
DeploymentPlan(A)
(1, 2, 3)
(2, 3)
Distributes only models that are allowed to be migrated on the device
Stationary device
MonitorDef(*)
Device(S)
NFPDeviceData(*)
(2) The Device spawns a new monitor in the mock mode for each MonitorDef
(3) After externally migrating-in the application component the Device turns the monitor into running mode
NFPDeviceData(A)
NFPData(Ab)
Monitor(Ab)
(2) <<mock>>
(3) <<running>>

Figure 2: Adaptation architecture. Following phases, denoted as (1), (2), (3), are depicted in the diagram: phase 1 (M isolated), phase 2 (S discovered) and phase 3 (Ab migrated to S).

are based on exchanging knowledge rather than invoking methods, various types of arrows denote various instances of interaction through knowledge described next). The adaptation logic forms a separate overlay architecture mirroring the architecture of the adapted application, based on emergent component ensembles.

5.2 Adaptation Architecture Components

The adaptation architecture on Figure 2 is formed by the following components:

Planner. Each adapted application, and particularly its NFP[1] preferences, are represented by the Planner component. Specifically, the Planner selects a (potentially optimal) deployment of the application, given the alternatives for deploying each of the applications components. We assume an external mechanism to interpret the deployment plan provided by the Planner and perform the adaptation (e.g. by migrating a component). The alternatives comprise important NFP-related data (NFPData) indicating the (potential) performance of the corresponding application component in that particular deployment (e.g. FPS[2], energy, etc.). The Planner also advertises definitions of Monitors for individual application components (MonitorDef); see Device.

Monitor. Each application component, particularly each of its deployment alternatives, is reflected by the Monitor component, which is responsible for obtaining the NFPData for that particular alternative. Monitor operates in one of two modes, depending on the actual deployment of the corresponding application component.

- Monitor is in the *running mode* if it resides on the same computation node as the corresponding application component. In this mode, it reflects the actual deployment and NFPData is obtained by performance measurement and analysis of the running application component (e.g., using the techniques and formalisms described in sections 2-4).

- Monitor is in the *mock mode* if it resides on a different computation node, it therefore represents a potential deployment alternative. NFPData is obtained from the included performance dependency model of the corresponding application component (for example a function from

[1]Non Functional Properties
[2]Frames Per Second

node configuration to estimated performance, $CPU \times GPU \rightarrow FPS$). In other words, Monitor roughly predicts the performance that the application component would exhibit if it were deployed on a particular computation node. The model might depend on machine-specific performance data (NFPDeviceData, for example the available CPU capacity, etc.); see Device.

Important is to point out that the ability of SPL to reason on data obtained by different sources (in this case, data acquired either from real measurements of from predictive performance models), by binding variables to different data sources (see section 3.2), enables the reuse of the same SPL formulas in both modes.

Device. Each computation node is reflected by the Device component. Specifically, a Device component ensures management of the Monitors (e.g., it instantiates Monitors advertised by newly discovered Planners) and it provides NFPDeviceData for Monitors in the mock mode.

5.3 Adaptation Architecture Ensembles

The expectation is that the number of available computation nodes, as well as the number of Monitors, changes dynamically. Therefore, the communication among the components exploits the concept of emergent component ensembles. The architecture involves the following ensembles (Figure 2):

Planner and Device(s). Each Planner is a coordinator of an ensemble that distributes MonitorDefs (including the performance dependency model) of application components to Devices representing currently available computation nodes (including the one the Planner is running on). The Planner is able to constraint which MonitorDefs should be distributed to which Devices (effectively constraining the potential migration destinations for a particular application component). A simplified example of a definition of this ensemble is in Listing 5. It specifies that only reachable devices within 2 network hops are to be considered and that this check is to be performed every 15 seconds. The distribution of the MonitorDefs is performed by adding the MonitorDef to the target components knowledge.

Listing 5 Example of an ensemble definition.

```
ensemble PlannerToDevice:
    coordinator:
        Planner
    member:
        Device
    membership:
        HopDistance(Planner.device, Device) ≤ 2
    knowledge_exchange:
        Device.monitorDef[Planner.app] := Planner.monitorDef
    scheduling:
        periodic(15s)
```

Planner and Monitor(s). Each Planner is a coordinator of an ensemble that aggregates NFPData from all Monitors corresponding to the components of the application reflected by the Planner. Thus, this ensemble aggregates all the deployment alternatives for the application.

Device and Monitor(s). Each Device component is a coordinator of an ensemble that distributes NFPDeviceData to the Monitors in the mock mode residing on the corresponding computation node.

5.4 Adaptation Architecture in Action

In this section, we illustrate on the motivation example the adaptation architecture interaction at run-time.

At first (phase 1, Figure 2), the ensemble distributes the MonitorDefs of both Af and Ab from Planner of A to the Device component of the mobile device (M), which subsequently spawns Monitors for both components and sets them to the running mode. The Monitors start measuring NFPData of the running components which are then aggregated back to the Planner. So far no deployment alternatives are discovered.

After the stationary device (S) is discovered (phase 2, Figure 2), the ensemble propagates MonitorDefs of the components that could be (potentially) migrated (i.e., Ab) to its Device component, which spawns a new Monitor. Since Ab is deployed on a different Device this Monitor runs in the mock mode. Thus, the Device component of the stationary device feeds the Monitor with NFPDeviceData allocated for A. Based on this NFPDeviceData and the performance dependency model of Ab the Monitor produces NFPData reflecting the expected performance of Ab on S. Consequently, another ensemble aggregates all the currently produced NFPData for Af and Ab to the Planner. The Planner thus eventually discovers that there are two deployment alternatives for Ab (i.e., one actually running on M and one modeled on S) and finally decides to deploy Ab on the stationary device.

After Ab is migrated to the stationary device (phase 3, Figure 2), the Monitor on S is set to the running mode, while the Monitor on M is set to the mock mode and the whole monitoring and planning process repeats.

In the case of discovering further stationary devices, new Monitors in the mock mode are spawned which eventually results in new deployment alternatives aggregated in the Planner (similarly, if devices disappear).

6 Designing Performance-Based Adaptation

In order to fully leverage the advantages of performance awareness, performance-based adaptation strategies are needed. Such strategies stand as possible remedies in face of performance degradation and add to the robustness and fault-tolerance of the system under development. To this end, this section builds on the *Invariant Refinement Method (IRM)* [KBP$^+$13, BGeH$^+$12], which guides the design of an application from high-level strategic goals and (performance) requirements to their realization in terms of system architecture with design alternatives, corresponding to different adaptation strategies.

The main idea of IRM is to capture the high-level system goals and requirements in terms of interaction *invariants*. Invariants describe the desired state of the system-to-be at every time instant. In general, invariants are to be maintained by the coordination of the different system components. At the design stage, by *component* we refer to a participant or actor of the system-to-be. A special type of invariant, called *assumption*, describes a condition that is expected to hold about the environment; an assumption is not intended to be maintained explicitly by the system-to-be. As a design decision, the identified top-level invariants are decomposed into more concrete sub-invariants forming a decomposition graph. By this decomposition, we strive to get to the level of abstraction where the (leaf) invariants represent detailed design of the particular system constituents – components, component processes, and ensembles. Two special types of invariants, the *process* and *exchange* invariants, are used to model the low-level component computation (processes) and interaction (ensembles), respectively.

Originally, IRM featured a single decomposition type, being essentially a refinement in the traditional interpretation, where the composition of the children exhibits all the behavior expected from the parent and (potentially) some more. Formally, the *AND-decomposition* of a parent invariant I_p into a conjunction of sub-invariants I_{s1} ... I_{sn} is a refinement if the conjunction of the sub-invariants entails the parent invariant, i.e., if it holds that

1. $I_{s1} \wedge ... \wedge I_{sn} \Rightarrow I_p$ (entailment)
2. $I_{s1} \wedge ... \wedge I_{sn} \nRightarrow false$ (consistency)

In order to capture the design alternatives (for performance-based adaptation), a special decomposition type along with the concept of situations is introduced in this deliverable. A *situation* is essentially a state which the composite system (software system and its environment) can reside in. Situations should not be confused with system (operating) modes; whereas the former refer to the perceived environment, which is inherently impossible to control, the later describe different system configurations, whose application is under the control of the running software.

When an invariant can get decomposed into two or more possible sub-invariants, each of which corresponds to a situation that features different requirements to be handled by the system-to-be, the special OR-decomposition is used. The *OR-decomposition* of a parent invariant I_p into two or more sub-invariants I_{s1}, I_{s2}, ... is correct if in any situation (corresponding to the alternative sub-invariants) there is at least one branch that refines the sub-invariant. Important is to notice that the situations identified and elaborated at each OR-decomposition step can potentially overlap. Although overlapping of situations naturally complicates the design, it is useful in case we are designing our system to work under only a limited number of (possibly overlapping) situations, and not their counterparts. Overlapping of situations also adds to the overall robustness of the system, as it essentially means that more than one solution concept is applicable to the same situation. Of course, situations can also be nested, following the observation that certain situations arise only in the context of others.

Technically, each situation in the IRM graph is associated with one or more assumptions (see Figure 3). These assumptions describe the conditions that are expected to hold under a given situation in a declarative way. Although not necessary, assumptions can be viewed as evaluation conditions ("triggers") for a given situation. Depending on the nature of the assumptions (and specifically based on whether they specify a condition that can be observed/quantified or not), different formalisms for their description can be employed, ranging from plain English to SPL formulas.

6.1 Illustrating scenario

To illustrate the IRM method, extended with the above-described concepts, we elaborate on a scenario from the Science Cloud Case Study, which itself is an extension of the base scenario already described in [BGeH$^+$12]. In this scenario, several heterogeneous network nodes, forming an open-ended cloud platform, run a third-party service, such us a user's program which involves computational-intensive scientific calculations. This way, the client running e.g., on a smartphone, can take advantage of the nearby available computational power (laptops, tablets, servers, etc.) by offloading parts of the application into other nodes that handle the back-end application logic. These nodes can be part of a traditional cloud infrastructure, if available, and thus leased on demand while the system is running and its demand for resources grows. The general assumptions are that a) the application's back-end can be partitioned into several nodes (allows "scaling out") and b) the mechanism of effectively migrating application parts across different nodes exists. Given the above scenario, the goal of the system under development is twofold: i) guarantee an upper limit in the rendering delay observed by the user, ii) distribute the off-loaded computation according to the capacity and current load of every contributing node.

Figure 3 shows a possible IRM graph for the above scenario. The design starts with the identified top-level invariant stating that *"Load is balanced while expected QoS is kept"*. The "expected QoS" has been quantified by the SPL formula that specifies an upper bound on the application's response time (500 ms). This invariant can be decomposed into two possible sub-invariants, based on the situation the system resides in and specifically based on whether extra computational power from a cloud data center is needed.

In the first case (Cloud not needed situation) invariant (2) is decomposed into one assumption (4) and two invariants, (5) and (6), following Figure 3. Assumption (4) specifies that *"Mobile nodes*

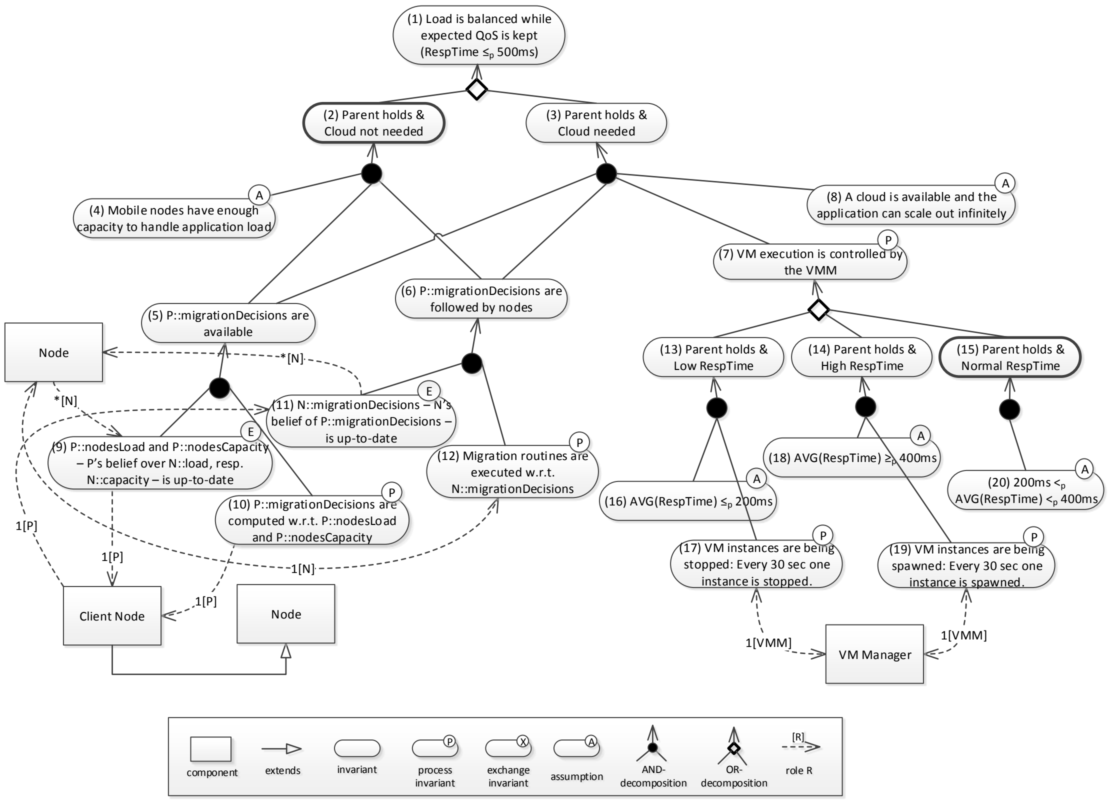

(1) Load is balanced while expected QoS is kept (RespTime ≤p 500ms)
(2) Parent holds & Cloud not needed
(3) Parent holds & Cloud needed
(4) Mobile nodes have enough capacity to handle application load
(5) P::migrationDecisions are available
(6) P::migrationDecisions are followed by nodes
(7) VM execution is controlled by the VMM
(8) A cloud is available and the application can scale out infinitely
(9) P::nodesLoad and P::nodesCapacity – P's belief over N::load, resp. N::capacity – is up-to-date
(10) P::migrationDecisions are computed w.r.t. P::nodesLoad and P::nodesCapacity
(11) N::migrationDecisions – N's belief of P::migrationDecisions – is up-to-date
(12) Migration routines are executed w.r.t. N::migrationDecisions
(13) Parent holds & Low RespTime
(14) Parent holds & High RespTime
(15) Parent holds & Normal RespTime
(16) AVG(RespTime) ≤p 200ms
(17) VM instances are being stopped: Every 30 sec one instance is stopped.
(18) AVG(RespTime) ≥p 400ms
(19) VM instances are being spawned: Every 30 sec one instance is spawned.
(20) 200ms <p AVG(RespTime) <p 400ms
A
P
E
Node
Client Node
Node
VM Manager
*[N]
*[N]
1[P]
1[P]
1[P]
1[N]
1[VMM]
1[VMM]
component
extends
invariant
process invariant
exchange invariant
assumption
AND-decomposition
OR-decomposition
role R
[R]

Figure 3: Cloud case study with situations – IRM graph.

have enough capacity to handle application load" – it is an example of an assumption specified in an informal way in natural language. Obviously, this kind of assumption cannot be formally specified or checked at execution time, but have to be included for design completeness and consistency. Invariants (5) and (6) are further refined into lower level semantics which specify the architecture of the load balancing mechanism. Specifically, the Client component (which is a specialization of the Node component, meaning that it contributes both to the invariants it takes a role in and to its parent's invariants) implements the load balancing logic by acquiring a view over every node's capacity and load (9), and devising a migration plan (10). In order for the migration plan to be followed, it has to be distributed to all nodes (11) and executed in every node separately (12).

In the second case (Cloud needed situation) invariant (3) is now decomposed into one assumption (8), and three invariants, (5) – (7). Whereas invariants (5) and (6), which describe the load balancing mechanism, are shared between the two situations, invariant (7) is local to the second situation. In particular, invariant (7) specifies that virtual machine execution should be controlled by the Virtual Machine Manager (VMM).

Here, three situations are distinguished, depending on the response time of the application: Low RespTime, High RespTime and Normal RespTime, each associated with a different assumption regarding the average response time over some period in time (assumptions (16), (18) and (20)). This kind of assumptions refer to a measurable system attribute and as such can be formally verified and checked at execution time. Indeed, all three assumptions are specified in SPL formulas (which would be evaluated at runtime using the SPL evaluator in conjunction with DiSL, as exemplified in Listing 4). The idea here is to use the concept of situations to specify a simple control logic: if the average response time is less than 200 ms (16), the VMM needs to react by stopping virtual machines (17); if it is more than 400 ms (18), the VMM has to start new virtual machines (19); otherwise (20), do nothing.

6.2 Integration with performance monitoring and SPL

Revisiting the goal of this section, IRM, extended by the concepts of situations and OR-decomposition, provides the means to model different adaptation strategies in the form of alternative low-level designs. Performance monitoring and metrics, as described in sections 2 and 3, can help to identify which situation the system is currently in, and react accordingly. Specifically, SPL can be employed in specifying the measurable performance-related assumptions, according to which situations are evaluated. SPL is also useful in specifying certain performance-related invariants (e.g. invariant (1), Figure 3) so that they are amenable to runtime checking. Finally, since performance awareness does not come for free, the advantage of the above-described method lies in the fact that it gives an early (requirements-time) indicator of the trade-off between monitoring and adaptation capability.

6.3 Mapping to DEECo concepts

IRM is tailored towards producing DEECo-based system designs [BGH$^+$13]. In particular, the leafs of the refinement graph can be either process invariants, exchange invariants or assumptions. The first two types are easily mappable to the DEECo concepts of component processes and ensembles, respectively. Additionally, IRM components are mapped into DEECo components in a straightforward way. For the assumptions, we distinguish between those that can be formally specified (e.g. in SPL formulas) and the more abstract ones, typically specified in natural language (English text). The first ones can be translated into instrumented code (see section 2) and evaluated at runtime. Technically, the assumption evaluation (and thus the situation evaluation) can be executed either in a process of a dedicated jDEECo component, acting as global system monitor, or as part of a separate plugin, attached to the jDEECo framework. The informal assumptions cannot be checked at runtime, so it is optimistically assumed that they hold true during execution.

7 Summary

In this deliverable we have reported on the advancements so far in the topic of performance awareness of service components and service-component ensembles. In our view, awareness, and performance awareness in particular, is achieved by an iterative process comprising monitoring, filtering and evaluation, and (optionally) prediction. As a natural next step, we regard adaptation actions as a remedy to identified violations of performance contracts.

Specifically, in this deliverable we have detailed the developed dynamic instrumentation platform based on DiSL domain specific language. This implements the "monitoring" part of the process. The platform is complemented by a performance requirements formalism, SPL, which implements the "filtering and evaluation" part of the envisioned process. We have sketched how to use predictive performance models to reason not only on past and current values, but also on future ones, thus implementing the "prediction" part. The applicability of our approach has been tested in a prototype adaptive cloud application, which integrates the above concepts and techniques. Lastly, we have reported on a method to derive performance requirements and bind them to specific components and adaptation actions, by extending the IRM method.

On the technical side, we have integrated our approach with the Java programming language and the jDEECo platform targeting ensemble-based component systems. Doing so, SPL behavior has been experimentally evaluated on realistic method execution time measurements.

As for the plans for the next iteration, we aim at adjusting SPL and its evaluation to reflect experience gathered so far, especially in the direction of (a) finding more suitable interpretations for the noisy environment of realistic measurements; (b) devising suitable syntax and semantic for basic operations on performance data such as addition of measurements; and (c) experimental evaluation of trend prediction data sources. Further, we plan to integrate SPL into ensemble membership evaluation (which will allow the definition of performance-related predicates in jDEECo), finalize the implementation on the science cloud case study, and provide support for IRM-driven reconfiguration in jDEECo.

The modular architecture of the SPL environment, where measurements can be substituted by results of evaluating predictive models, is particularly suitable for connection to those ASCENS project activities where stochastic models are developed. The ability to relate the actually observed system performance to the performance captured (or assumed) by such models is an essential component of the ASCENS development lifecycle, which explicitly deals with feedback between the design and runtime phases of an ensemble. As one such model, the project partners plan to define a Markovian extension of SCEL during the coming year. It will be used to support quantitative analysis of adaptive systems – in particular, it will focus on the *group oriented operations*. These constructs, specifically used to coordinate ensembles activities, can be used to interact with those components that satisfy a given predicate. The effort will examine possible alternatives that can be used to include quantitative aspects in SCEL at different level of abstractions ranging from concrete ones (where the underlying protocol governing the component interactions is considered) to abstract ones (where the underlying communication infrastructure is abstracted).

References

[Bar84] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Mathematical Programming Study. North-Holland Publishing Company, Amsterdam, 1984.

[BBH$^+$12] Lubomir Bulej, Tomas Bures, Vojtech Horky, Jaroslav Keznikl, and Petr Tuma. Performance Awareness in Component Systems: Vision Paper. In *Proceedings of COMPSAC 2012*, COMPSAC '12, 2012.

[BBHK13] Lubomír Bulej, Tomáš Bureš, Vojtěch Horký, and Jaroslav Keznikl. Adaptive deployment in ad-hoc systems using emergent component ensembles: vision paper. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, pages 343–346, New York, NY, USA, 2013. ACM.

[BBK$^+$12] Lubomir Bulej, Tomas Bures, Jaroslav Keznikl, Alena Koubkova, Andrej Podzimek, and Petr Tuma. Capturing Performance Assumptions using Stochastic Performance Logic. In *Proc. 3rd Intl. Conf. on Performance Engineering*, ICPE'12, Boston, MA, USA, 2012.

[BGeH$^+$12] Tomáš Bureš, Ilias Gerostathopoulos, Vojtěch Horký, Jaroslav Keznikl, Jan Kofroň, Michele Loreti, and František Plášil. Deliverable D1.5: Language Extensions for Implementation-Level Conformance Checking, November 2012.

[BGH$^+$13] Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. DEECo - an Ensemble-Based Component System. In *Proc. of the International ACM SIGSOFT Symposium on Component Based Software Engineering*, CBSE '13, Vancouver, Canada, 2013. ACM.

[KBP$^+$13] Jaroslav Keznikl, Tomas Bures, Frantisek Plasil, Ilias Gerostathopoulos, Petr Hnetynka, and Nicklas Hoch. Design of Ensemble-Based Component Systems by Invariant Refinement. In *Proc. of the 16th International ACM SIGSOFT Symposium on Component Based Software Engineering*, CBSE '13, Vancouver, Canada, 2013. ACM.

[MZA$^+$12] Lukáš Marek, Yudi Zheng, Danilo Ansaloni, Walter Binder, Zhengwei Qi, and Petr Tuma. DiSL: An extensible language for efficient and comprehensive dynamic program analysis. In *Proc. 7th Workshop on Domain-Specific Aspect Languages*, DSAL '12, pages 27–28, New York, NY, USA, 2012. ACM.

[Wel47] B. L. Welch. The Generalization of 'Student's' Problem when Several Different Population Variances are Involved. *Biometrika*, 34(1/2):28–35, 1947.