Chapter II.5

Supporting Performance Awareness in Autonomous Ensembles*

Lubomír Bulej[1], Tomáš Bureš[1], Ilias Gerostathopoulos[1], Vojtěch Horký[1], Jaroslav Keznikl[1], Lukáš Marek[1], Max Tschaikowski[2], Mirco Tribastone[2], and Petr Tůma[1]

[1] Department of Distributed and Dependable Systems, Faculty of Mathematics and Physics, Charles University, Czech Republic
[2] Electronics and Computer Science, University of Southampton, United Kingdom

Abstract. The ASCENS project works with systems of self-aware, self-adaptive and self-expressive ensembles. Performance awareness represents a concern that cuts across multiple aspects of such systems, from the techniques to acquire performance information by monitoring, to the methods of incorporating such information into the design making and decision making processes. This chapter provides an overview of five project contributions – performance monitoring based on the DiSL instrumentation framework, measurement evaluation using the SPL formalism, performance modeling with fluid semantics, adaptation with DEECo and design with IRM-SA – all in the context of the cloud case study.

Keywords: performance, monitoring, modeling, adaptive systems, autonomic systems

1 Introduction

The ASCENS project deals with adaptive systems formed as ensembles of components that both possess and exchange knowledge. In general, an ensemble achieves awareness by observing 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.

Each of the individual steps that combine to achieve awareness can be related to performance. Consider the example of an adaptive cloud application, used throughout this chapter and outlined in more detail in Chapter IV.3 [38]. Components of such an application may measure the request arrival rate or the request processing time, aiming to adjust resource pool sizes – such as caches or threads – to match the actual workload. Additionally, the components may also monitor the utilization of the host platform and form ensembles with components of other applications on the same host, reacting to possible overload with

* This research was supported by the European project IP 257414 (ASCENS).

M. Wirsing et al. (eds.): Collective Autonomic Systems, LNCS 8998, pp. 291–322, 2015.

coordinated migration. Besides utilizing the measurements directly, the components may also derive useful knowledge by analyzing long term trends or periodic behavior observed in performance, or by comparing observed performance with model based predictions, to support proactive rather than reactive adaptation.

Many research contributions of the ASCENS project deal with awareness in general, rather than focusing on a particular aspect such as location awareness or performance awareness. These include results described in other chapters of this book – such as reasoning and learning in Chapter II.4 [23], SOTA in Chapter II.1 [4], SCEL in Chapter I.1 [41], IRM-SA in Chapter III.4 [12] – whose neutral character makes the project results more broadly applicable. Performance awareness represents a concern that cuts across these results, we therefore focus on contributions that facilitate integration of performance awareness in the broader 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 in the context of DiSL [37], described in Section 2.

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, SPL [9], is presented in Section 3.

Besides integration in system behavior specification, the SPL formalism can be embedded in system implementation, permitting smooth transition between the specification and the implementation. The SPL formalism also does not require differentiating between measurements of a real system and predictions of a high level model, which allows us to efficiently integrate performance modeling activities. The support for SPL at the implementation level is outlined in Section 4 and available in the form of prototype tools with open source licensing [25].

The performance modeling techniques for ensembles are introduced in Section 5. Ensemble performance modeling is challenging due to a high number of potentially interacting components. Models that track the state of individual components encounter state explosion issues. The ASCENS project investigates fluid modeling techniques that rely on symmetries in the behavior of individual components to keep the model both accurate and tractable.

Finally, Section 6 demonstrates the integration of performance awareness in the DEECo component model, and Section 7 presents the process of designing for performance adaptation with the IRM-SA method.

This is an overview text that connects multiple previously published research results of the ASCENS project. We refer the reader to the original publications as appropriate, especially where the detailed formal proofs and experimental evaluation is concerned. In particular, a broader overview of the ASCENS project can

be found in [57], more information about the DiSL instrumentation framework is in [37,35], the SPL formalism description is cited and summarized from [9,25,8], the introduction on performance modeling with fluid semantics is condensed from [54], and various elements of the case study with DEECo and IRM-SA have been published in [7]. Where reasonably possible, we have also refrained from printing code, and instead encourage the reader to access our evolving research prototypes directly on the ASCENS project website.

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, the ensemble must be able to monitor performance at any location that the reasoning process can consider. At the same time, the ensemble must avoid continuous monitoring of many locations, which would induce high overhead and therefore unduly influence the ensemble behavior.

The combination of the two requirements necessitates performance monitoring with dynamic instrumentation that can be inserted and removed on demand. To execute on a specific platform, such an instrumentation has to solve various issues of highly technical nature. The focus of the ASCENS project with dynamic instrumentation is on Java, a platform used in the autonomic cloud case study and the jRESP and jDEECo frameworks.

At a glance, Java provides several technologies with potential use for performance monitoring, each with a particular set of advantages and limitations. The *JVM Tool Interface* (JVMTI) [43] is a powerful native interface used for monitoring, debugging, profiling and similar application analyses. The *java.lang.instrument* API provides class-loading hooks that allow instrumenting an application using a custom Java agent. In addition, Java also provides a standard interface for delivering performance data to applications, based on *Java Management Extensions* (JMX).

To combine the available Java technologies in a robust instrumentation solution, we participate in the development of DiSL [37], a domain specific language and framework that allows to conveniently monitor an application using instrumentation. Using the aspect oriented programming model, DiSL can insert code fragments into Java applications. We use DiSL to specify and execute the performance monitoring code, whose output events are processed by the custom SPL framework, described in Sections 3 and 4.

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.

The method entry time is sampled in the onMethodEntry method. DiSL is guided by the @Before annotation to insert the entire body of onMethodEntry at

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));
  }
}
```

the beginning of each monitored method. Similarly, the method exit time is sampled and the method duration calculated in onMethodExit, which is inserted at the end of each monitored method. The use of DiSL removes the need for manual instrumentation, as well as complex handling of situations such as exceptional method exits.

DiSL provides other features useful for dynamic instrumentation, including the ability to insert monitoring routines at arbitrary code locations. DiSL contains specialized *SyntheticLocal* and *ThreadLocal* variables that allow efficient communication between the monitoring code that handles related events. 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, *Scopes* and *Guards*. *Scope* is a language construct for defining patterns restricting class and method instrumentation. *Guard* is a standard Java class that allows to evaluate complex instrumentation conditions during weaving. As a vital property from the performance monitoring perspective, DiSL does not insert any instrumentation besides snippets, and therefore does not incur any hidden overhead. The monitoring code is prevented from modifying the control flow of the application and the instrumentation does not violate the virtual machine 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.

Related to DiSL are instrumentation frameworks such as AspectJ [33], which offers similar features but less control over the instrumentation process and limited dynamic instrumentation support. Better control over the glue code is offered by Javassisst [15] or ASM [2], however, this requires working at the bytecode level. Higher level tools, such as Perf4J [44], rely on these instrumentation frameworks for inserting probes into code. On the whole system level, generic monitoring tools such as DTrace [13] or SystemTap [49] are also available. For more thorough comparison and information about DiSL, we refer the reader to [37]. The DiSL framework is available for download at [36], the monitoring prototype is available at [48].

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 Service Level Agreement (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 processor cache layout 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) and was originally introduced in [9] in the context of software performance evaluation, with broader applications discussed in [7] and practical experience reported in [25]. See [8] for formal proofs of the SPL properties presented here.

SPL is related to previous research on languages for expressing performance properties. An early example of such a language is PSpec [45], a language for expressing performance assertions in performance tests. Unlike SPL, it requires that the performance expectations are specified against absolute bounds. Performance expectations are associated with behavior specification in PIP [46]. Assertion checking and runtime adaptation are also possible with the PA language [55]. The SPL framework implementation offers features similar to JUnitPerf [17], an extension of JUnit [50] is for unit testing of performance.

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, graph density, etc.

Definition 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.*

Definition 2. Method workload *is a random variable* $L^{p_1,\ldots,p_n}$ *such that*

$$L^{p_1,\ldots,p_n} = \mathfrak{L}(p_1, \ldots, p_n)$$

for a given workload class $\mathfrak{L}$ *and parameters* $p_1, \ldots, p_n$*.*

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. 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.

Definition 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* $p_1, \ldots, p_n$ *returns a random variable, whose observations correspond to execution duration of method* M *with input parameters obtained from observations of* $L_M^{p_1,\ldots,p_n} = \mathfrak{L}_M(p_1, \ldots, p_n)$*, 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 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:*

$$x \leq x \tag{1}$$

$$(x \leq y \wedge y \leq x) \leftrightarrow x = y \tag{2}$$

$$(x \leq y \wedge y \leq z) \rightarrow x \leq z \tag{3}$$

$$\begin{gathered} \textit{For each pair } tl, tr \in FunT \textit{ such that} \\ \forall o \in \mathbb{R} : tl(o) \leq tr(o), \textit{there is an axiom} \\ P_M(x_1, \ldots, x_m) \leq_{p(tl,tr)} P_M(x_1, \ldots, x_m) \end{gathered} \tag{4}$$

$$\begin{gathered} (P_M(x_1, \ldots, x_m) \leq_{p(tm,tn)} P_N(y_1, \ldots, y_n) \wedge \\ P_N(y_1, \ldots, y_n) \leq_{p(tn,tm)} P_M(x_1, \ldots, x_m)) \leftrightarrow \\ P_M(x_1, \ldots, x_m) =_{p(tm,tn)} P_N(y_1, \ldots, y_n) \end{gathered} \tag{5}$$

Using SPL, we can express assumptions about method performance. The lambda notation [3] with $id = \lambda x.x$ is introduced for brevity:

Example 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."

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

3.2 Logic Interpretations

To ensure correspondence between the SPL formula in Example 1 and its textual description, we need to introduce the appropriate semantic that provides the intended SPL interpretation. In [9], we first introduce an expected-value-based interpretation, where the SPL relations are defined over expected values of the random variables that represent execution duration. This interpretation is useful when the expected values are known, such as when performance is computed using analytical models. When performance is observed through monitoring, an interpretation based on the observed samples is needed.

Simple Sample-Based Interpretation. To formulate the sample-based interpretation from [9], we first fix the set of observations for which the relations will be interpreted. We define an *experiment*, denoted $\mathcal{E}$, as a finite set of observations of method performances under a particular method workload.

Definition 5. *Experiment* $\mathcal{E}$ *is a collection of* $\mathcal{O}_{P_M(p_1,\ldots,p_m)}$*, where*

$$\mathcal{O}_{P_M(p_1,\ldots,p_m)} = \{P_M^1(p_1, \ldots, p_m), \ldots, P_M^V(p_1, \ldots, p_m)\}$$

is a set of V *observations of method performance* P_M *subjected to workload* $L_M^{p_1,\ldots,p_m}$*, and where* $P_M^i(p_1, \ldots, p_m)$ *denotes i-th observation of performance of method* M*.*

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

Definition 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*

$$\mathrm{H}_0 : E(tm(P_M^i(x_1, \ldots, x_m))) \leq E(tn(P_N^j(y_1, \ldots, y_n)))$$

  cannot be rejected by one-sided Welch's t-test [56] 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*

$$\mathrm{H}_0 : E(tm(P_M^i(x_1, \ldots, x_m))) = E(tn(P_N^j(y_1, \ldots, y_n)))$$

  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 $E(tm(P_M^i(\ldots)))$ *and* $E(tn(P_N^j(\ldots)))$ *denote the mean value of performance observations transformed by function tm or tn, respectively.*

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 [25]. When SPL is used to make adaptation decisions at runtime, the number of collected samples might be smaller by several orders of magnitude, and the individual samples might suffer from many kinds of disruptive artefacts.

We discuss two kinds of disruptive artefacts – initial transient conditions and run-to-run fluctuations. We assume that system execution consists of stationary episodes termed *runs*.[3] Within a run, system performance would be considered stable, except for initial transient conditions disrupting the run. From run to run, system performance can exhibit fluctuations that are measurable and statistically significant, but not controllable and not significant from the adaptation perspective [30]. The interpretations in the following sections explicitly handle runs.

3.3 Handling Initial Transient Conditions

On many computing platforms, runs are exposed to mechanisms that may introduce transient execution time changes. Measurements performed under these

[3] Practical reasons for the existence of runs are for example rejuvenation episodes with virtual machine restarts.

conditions are typically denoted as warmup measurements, in contrast to steady state measurements.[4]

One well known mechanism that introduces warmup is just-in-time compilation. With just-in-time compilation, the method whose execution time is measured is initially executed by an interpreter or compiled into machine code with selected optimizations based on static information. During execution, the same method may be compiled with different optimizations based on dynamic information and therefore exhibit different execution times. This effect is illustrated on Figure 1.[5]

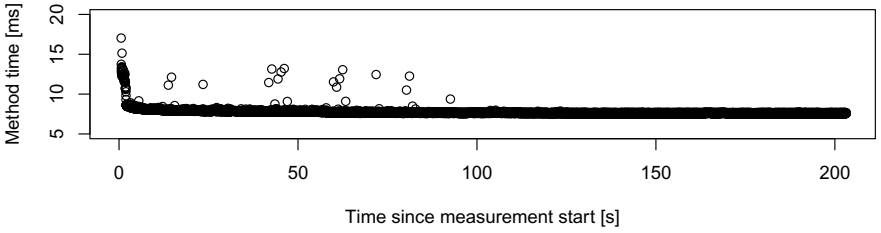


Fig. 1. Example of how just-in-time compilation influences method execution time

The warmup measurements are not necessarily representative of steady state performance and are therefore typically avoided. Sometimes, such measurements can be identified by analyzing the collected observations. Intuitively, long sequences of observations with zero slope (such as those on the right side of Figure 1) likely originate from steady state measurements, in contrast to initial sequences of observations with downward slope (such as those on the left side of Figure 1), which likely come from warmup. This intuition is not always reliable, because the warmup measurements may exhibit very long periods of apparent stability between changes. These would look like steady state measurements when analyzing the collected observations. Furthermore, the mechanisms that introduce warmup may not have reasonable bounds on warmup duration. As one example, just-in-time compilation can be associated with events such as change in branch behavior or change in polymorphic type use, which may occur at any time during measurement.

[4] The illustrative measurements in this section were collected on an Intel Xeon E5-2660 machine with 2 sockets, 8 cores per socket, 2 threads per core, running at 2.2 GHz, 32 kB L1, 256 kB L2 and 20 MB L3 caches, 48 GB RAM, running 64 bit Fedora 20 with OpenJDK 1.7.

[5] The method is SAXBuilder::build, used to build a DOM tree from a byte array stream, from the JDOM library [29]. The selection is ad hoc, made to illustrate practical behavior.

Given these obstacles, we believe that warmup should not be handled at the level of logic interpretation. Instead, knowledge of the relevant mechanisms should be used to identify and discard observations collected during warmup.

In addition to the transient initial conditions, the logic interpretation has to cope with run-to-run fluctuations. In contemporary computer systems, the execution conditions include factors that stay relatively stable within each run but differ between runs – for example, a large part of the process memory layout on both virtual and physical address level is determined at the beginning of each run. When these factors cannot be reasonably controlled, as is the case with the memory layout example, each run will execute with possibly different conditions, which can affect the measurements. The memory layout example is one where a significant impact was observed in multiple experiments [30,40]. Therefore, no single run is entirely representative of the observable performance.

A common solution to the problem of changing conditions between runs is collecting observations from multiple runs. In practice, each run takes some time before performing steady state measurements, the number of observations per run will therefore be high but the number of runs will be low. In this situation, the sample variance S^2 (when computed from all the observations together) is not a reliable estimate of the population variance σ^2 and the sample-based logic interpretation becomes more prone to false positives, rejecting performance equality even between measurements that differ only due to changing conditions between runs. The problem can be avoided by introducing a sensitivity limit [25], or by explicitly considering runs in the logic interpretations, done next.

3.4 Parametric Mean Value Interpretation

From the statistical perspective, measurements taken within a run have a conditional distribution depending on a particular run. This is typically exhibited as a common bias shared by all measurements within the particular run [31]. Assuming that each run has the same number of observations, the result statistics collected by a benchmark can be modeled as the sample mean of sample means of observations per run (transformed by tm as necessary):

$$\overline{M} = \frac{1}{ro} \sum_{i=1}^{r} \sum_{j=1}^{o} tm(P_M^{i,j}(x_1, \ldots, x_m))$$

where $P_M^{i,j}(x_1, \ldots, x_m)$ denotes the j-th observation in the i-th run, r denotes the number of runs and o denotes the number of observations in a run.

From the Central Limit Theorem, $\overline{M}$ and the sample means of individual runs $\overline{M}_i = \frac{1}{o} \sum_{j=1}^{o} tm(P_M^{i,j}(x_1, \ldots, x_m))$ are asymptotically normal. In particular, a run mean converges to the distribution $N(\mu_i, \sigma_i^2/n)$. Due to the properties of the normal distribution, the overall sample mean then converges to the distribution

$$\overline{M} \sim N\left(\mu, \frac{\rho^2}{r} + \frac{\overline{\sigma^2}}{ro}\right)$$

where $\overline{\sigma^2}$ denotes the average of run variances and ρ^2 denotes the variance of run means [31].

This can be easily turned into a statistical test of equality of two means, used by the interpretation defined below. Note that since the variances are not known, they have to be approximated by sample variances. That makes the test formula only approximate, though sufficiently precise for large r and o [31].

Definition 7. *Let $tm, tn : \mathbb{R} \to \mathbb{R}$ be performance observation transformation functions, P_M and P_N be method performances collected over r_M, r_N runs, each run having o_M, o_N observations respectively, $x_1, \ldots, x_m, y_1, \ldots, y_n$ be the 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*

$$\overline{M} - \overline{N} \leq z_{(1-\alpha)} \sqrt{\frac{o_M R_M^2 + \overline{S_M^2}}{r_M o_M} + \frac{o_N R_N^2 + \overline{S_N^2}}{r_N o_N}}$$

where $z_{(1-\alpha)}$ is the $1-\alpha$ quantile of the normal distribution,

$$\overline{S_M^2} = \frac{1}{r_M(o_M - 1)} \sum_{i=1}^{r} \sum_{j=1}^{o} \left(tm(P_M^{i,j}(x_1, \ldots, x_m)) - \frac{1}{o} \sum_{k=1}^{o} tm(P_M^{i,k}(x_1, \ldots, x_m)) \right)^2$$

$$R_M^2 = \frac{1}{r_M - 1} \sum_{i=1}^{r} \left[\left(\frac{1}{n} \sum_{j=1}^{o} tm(P_M^{i,j}(x_1, \ldots, x_m) \right) - \overline{M} \right]^2$$

and similarly for $\overline{S_N^2}$ and R_N^2.

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

$$\left|\overline{M} - \overline{N}\right| \leq z_{(1-\alpha)} \sqrt{\frac{o_M R_M^2 + \overline{S_M^2}}{r_M o_M} + \frac{o_N R_N^2 + \overline{S_N^2}}{r_N o_N}}$$

3.5 Non-parametric Mean Value Interpretation

The interpretation given by Definition 7 requires a certain minimal number of runs to work reliably. This is because the distribution of run means $\overline{M}_i = o^{-1} \sum_{j=1}^{o} tm(P_M^{i,j}(x_1, \ldots, x_m))$ is not normal even for relatively large values of o – illustrated on Figure 2.[6] Again, for a small number of runs this typically results in a high number of false positives, we therefore provide an alternative

[6] Each run collects $o = 20000$ observations after a warmup of 40000 observations. The method is SAXBuilder::build, used to build a DOM tree from a byte array stream, from the JDOM library [29]. The selection is ad hoc, made to illustrate practical behavior.

interpretation that uses the distribution of $\overline{M}_i$ directly. It works reliably with any number of runs (including only one run), however, the price for this improvement is that the test statistics has to be learned first (e.g. by observing performance across multiple runs of similarly behaving methods).

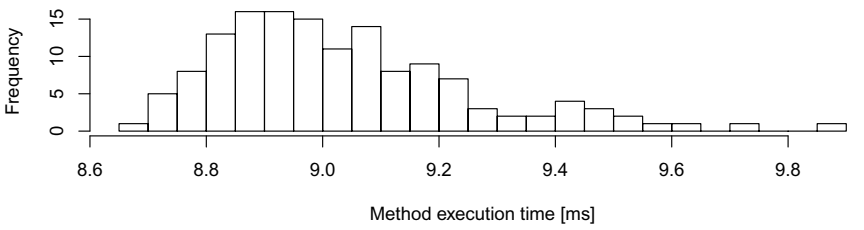


Fig. 2. Example histogram of run means from multiple measurement runs of the same method and workload

We assume that all observations $P_M^{i,j}(x_1, \ldots, x_m)$ in a run i are identically and independently distributed with a conditional distribution depending on a hidden random variable C. We denote this distribution as $B_M^{C=c}$, meaning the distribution of observations in a run conditioned by drawing some particular c from the hidden random variable C.

We further define the distributions of the test statistics as follows:

- $B_{\overline{M},r_M,o_M}$ is the distribution function of

$$(r_M o_M)^{-1} \sum_{i=1}^{r_M} \sum_{j=1}^{o_M} tm(\dot{P}_M^{i,j}(x_1, \ldots, x_m))$$

  where $\dot{P}_M^{i,j}(x_1, \ldots, x_m)$ denotes a random variable with distribution $B_M^{C=c}$ for c drawn randomly once for each i. In other words, $B_{\overline{M},r_M,o_M}$ denotes a distribution of a mean computed from r_M runs of o_M observations each.
- $B_{\overline{M},r_M,o_M-\overline{N},r_M,o_M}$ is the distribution function of the difference $\tilde{M} - \tilde{N}$, where $\tilde{M}$ is a random variable with distribution $B_{\overline{M},r_M,o_M}$ and $\tilde{N}$ is a random variable with distribution $B_{\overline{N},r_N,o_N}$.

After adjusting the distributions $B_{\overline{M},r_M,o_M}$ and $B_{\overline{N},r_N,o_N}$ by shifting to have an equal mean, the performance comparison can be defined as:

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

$$\overline{M} - \overline{N} \leq B^{-1}_{\overline{M},r_M,o_M-\overline{N},r_N,o_N}(1-\alpha)$$

where $\overline{M}$ denotes the sample mean of $tm(P_M(x_1,\ldots,x_m))$, $\overline{N}$ is defined similarly, and $B^{-1}_{\overline{M},r_M,o_M-\overline{N},r_N,o_N}$ denotes the inverse of the distribution function $B_{\overline{M},r_M,o_M-\overline{N},r_N,o_N}$ (i.e. for a given quantile, it returns a value).

– $P_M(x_1,\ldots,x_m) =_{p(tm,tn)} P_N(y_1,\ldots,y_n)$ iff

$$B^{-1}_{\overline{M},r_M,M-\overline{N},r_N,o_N}(\alpha) \leq \overline{M} - \overline{N} \leq B^{-1}_{\overline{M},r_M,o_M,-\overline{N},r_N,o_N}(1-\alpha)$$

An important problem is that the distribution functions $B_{\overline{M},r_M,o_M}$, $B_{\overline{N},r_N,o_N}$, and consequently $B_{\overline{M},r_M,o_M-\overline{N},r_N,o_N}$ are unknown. To get over this problem, we approximate the B-distributions in a non-parametric way by bootstrap and Monte-Carlo simulations [47]. This can be done either by using observations $P^{i,j}_M(x_1,\ldots,x_m)$ directly, or by approximating from observations of other methods whose performance behaves similarly between runs.

Finally, we define a non-parametric interpretation of the logic as follows:

Definition 8. *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, $\alpha \in \langle 0, 0.5\rangle$ be a fixed significance level, and let X, Y be methods (including M and N) whose performance observations are used to approximate the distributions of P_M and P_N, respectively.*

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*

$$\overline{M} - \overline{N} \leq B^{*-1}_{\overline{X},r_X,o_X-\overline{Y},r_Y,o_Y}(1-\alpha)$$

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

$$B^{*-1}_{\overline{X},r_X,o_X-\overline{Y},r_Y,o_Y}(\alpha) \leq \overline{M} - \overline{N} \leq B^{*-1}_{\overline{X},r_X,o_X-\overline{Y},r_Y,o_Y}(1-\alpha)$$

4 Coding for Performance Awareness

The basic role of SPL is to provide a versatile mechanism to express performance properties at various stages of the software development process – the design requirements, the developer assumptions, the test conditions, the health indicators, the adaptation triggers, and so on. Every specialized use of the formalism brings additional considerations, which must be addressed to achieve fitness for purpose. Here, we outline how the formalism is connected to data and code in the implementation environment, additional considerations in the context of adaptive systems are discussed in [10], the context of software documentation is examined in [26], software testing in [25].

4.1 Performance Data Sources

The introduction of SPL in Section 3 formalized performance as the execution time of a particular method under a particular workload, as collected by performance monitoring from Section 2. However, SPL provides considerable freedom as far as the input data is concerned. For example, in the autonomic cloud case study from Chapter IV.3 [38], applications migrate from busy to idle nodes and it is therefore useful to compare the system load metric to identify the busy and idle nodes.

The system load is typically represented as the number of ready threads on the node. 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 the load, we can rely on a trivial SPL interpretation – both L_A and L_B are scalars and plain comparison can therefore be used. If multiple observations of load are available, the comparison can rely on any of the sample-based interpretations. 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 important.

More systematically, applying SPL to data other than method execution times gives rise to the concept of *data sources*. The performance data is abstracted as a random variable and the data source is responsible for providing information on the random variable that the particular interpretation requires – for example, the expected value for the expected-value-based interpretation, the sample mean $\overline{X}$, sample size V_X and sample variance S_X^2 for the simple sample-based interpretation, and so on. Introducing data sources provides an important level of abstraction in the software development process. In particular, the same SPL formulas can be used in multiple software development phases from modeling to execution – the only difference is what data source is *bound* to the actual random variables in the SPL formula.

We illustrate the concept on an example of an adaptive application. Consider a problem that can be solved using two different algorithms, A and B, with A performing better on larger and B on smaller inputs. The adaptation consists of choosing the better performing algorithm depending on the actual input size. The adaptation is simple when the limit size – size s_{limit} such that A performs better for inputs larger than s_{limit} and B for inputs smaller than s_{limit} – is known, however, s depends on the execution platform and therefore cannot be included in the application design. Instead, an equally simple SPL formula can be used to describe the condition for selecting particular algorithm for given input size s – we use A if $A(s) \leq B(s)$ and B otherwise.

In the early application design phase, modeling might be used to assess the application behavior – and because data on the actual performance would not yet

be available, the model would bind $A(s)$ and $B(s)$ to data sources that roughly estimate performance from the algorithm complexity. 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 a 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 actual timing of the particular execution platform.

4.2 Language Integration Support

The outlined uses of SPL require integrating the support for performance monitoring and formula evaluation with data source binding in a particular implementation environment. In the ASCENS project, the choice for the prototype integration environment is Java– it is a well-known multi-platform language that is used in the case studies and in the jRESP and jDEECo frameworks. 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.

To indicate performance properties (requirements, expectations, conditions) at code level, SPL formulas are attached in the form of annotations to the relevant method, as outlined in the example in Listing 2.

Listing 2. Java annotation 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 annotation 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.

The annotations are suitable for use by external tools that evaluate the performance properties at development time or deployment time – such as with testing, outlined in [25]. 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 [6].

Besides the static language integration based on annotations, we have also designed an API for evaluating SPL formulas directly from application code at runtime. Listing 3 depicts a code fragment that uses the API to check 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 3. 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 enough samples were collected).
 */
if (formula.evaluate() == Result.FALSE) {
    logger.warn("myMethod is too slow!");
}
```

The *SPL.instrument()* method uses DiSL, outlined in Section 2, to instrument the running Java application 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 a formula to refer to the method performance. In addition to plain Java, the prototype implementation includes support for OSGi, where the use of class loaders for component isolation poses additional technical challenges.

4.3 Integrating Predictive Models

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 [19]. This option is handled in 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 Section 6, 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 Modeling Performance

Reasoning about the performance of ensembles introduces a level of difficulty due fact to the system under study comprises of potentially many interacting components. To cope with this inherent complexity, in the ASCENS project we followed an established line of research on finding symmetries at the model level which induce a suitable coarsening of the state space that retains some information about the original one. In this respect, the classical results on bisimilarity allow to relate processes of possibly different state space sizes which are however equivalent with respect to an external observer [39]. Analogous notions have been developed form Markovian process algebra with a discrete-state Markov chain semantics. For instance, the notions of Markovian bisimulation for MTIPP [20] and strong equivalence for PEPA [22] are equivalence relations that give an exactly aggregated Markov chain in terms of the theory of CTMC *lumpability* [5].

A similar line of research in the ASCENS project leads to exact as well as approximate notions of aggregation for Markovian process algebra. Different from the listed literature, we targeted *fluid semantics*. This has recently emerged as an alternative to the classical Markovian semantics, describing the model dynamics in terms of a system of ordinary differential equations (ODEs) [21,34]. These can be interpreted as a deterministic approximation to the expectation of the Markov chain [14,16,18,51]. When the model under consideration consists of many copies of processes in parallel, the ODE system size is independent of the multiplicities of such copies. This is considerably more convenient than the Markovian representation which suffers from the well-known problem of *state explosion*, where the number of states grows exponentially (in the worst case) with the number of concurrent processes in the model.

5.1 Fluid Process Algebra

The motivating observation here is that not all models of ensemble-based systems enjoy a compact ODE description [53]. Indeed, the problem of aggregating large-scale models based on ODEs has attracted the attention of researchers in a variety of other disciplines including control theory [1], theoretical ecology [28], and chemical engineering [42]. Here we consider a *Fluid Process Algebra*, presented in [52] as a fragment of the Markovian process algebra PEPA [22].

Exact Fluid Lumpability. In [52] we define the notion of *exact fluid lumpability* (EFL). It establishes an equivalence relation between processes such that their associated ODE solutions have equal trajectories whenever they are initialized with the same conditions. To be concrete yet informal for the purpose of overviewing our results, let us consider the process

$$\Big(P_1[N_1] \parallel_K P_2[N_2] \parallel_K \cdots \parallel_K P_D[N_D]\Big) \parallel_L Q[M] \tag{6}$$

where, for all $1 \leq i \leq D$, P_i is some sequential component that is replicated N_i times, and $\parallel_K$ is the parallel operator, parameterised by an action set K, in a CSP-like fashion. EFL may essentially reduce the analysis of such a model by considering the fluid trajectory of a *representative* P_i, which is shown to be equal to that of any other P_j if, for all $1 \leq i, j \leq D$, it holds that $N_i = N_j$ and P_i and P_j are isomorphic. Thus, denoting by $V_S(t)$ the ODE solution related to the sequential component S, EFL would yield $V_{P_i}(t) = V_{P_j}(t)$ for all t. EFL has been exploited in [53] as a building block to automatically simplify models that feature a pattern of *replicated composites* – large ensembles of composite processes which themselves consist of replicated copies of other composites, with an arbitrary level of nesting. However, in general symmetries are required both at the level of the sequential component and at the compositional level, by ensuring that all populations have the same size.

Taking EFL as the starting point of our investigation, it is possible to extend it along two orthogonal directions [54]. In one direction, we define a new notion of lumpability, called *ordinary fluid lumpability* (OFL), which relaxes assumptions on certain symmetries whilst still guaranteeing exactness of the aggregated system. In the other direction, we consider approximate versions of both EFL and OFL which can yield coarser aggregations, at the cost of losing exactness.

Ordinary Fluid Lumpability (OFL). Similarly to EFL, ordinary fluid lumpability considers symmetry through isomorphism at the sequential level; thus, it still requires that P_i and P_j be isomorphic for all i, j. However, it allows *heterogeneity* at the compositional level: in the example above, it may yield an exactly aggregated ODE system even if $N_i \neq N_j$. However, unlike EFL, where all the trajectories of the original ODE system can be obtained from the solution of the aggregate, in OFL the aggregate gives the exact sum of the solutions of its parts, but their individual trajectories cannot be recovered. Thus, for instance, OFL would define an aggregate ODE for some variable $W_P(t)$ and show that $W_P(t) = V_{P_1}(t) + V_{P_2}(t) + \ldots + V_{P_D}(t)$. More precisely, OFL identifies an aggregate ODE system where the solution to each ODE is the linear combination of solutions of ODEs belonging to the original system.

Approximate Aggregations. To relax the requirement on the *exactness* of the aggregation, we study ε-variants of both EFL and OFL as a means of relaxing symmetries at the sequential level. These variants allow non-isomorphic processes to be aggregated if there exists a *perturbation* in the rates that makes them isomorphic. For instance, let us take $P_i \xrightarrow{(\alpha,r)} P_k$ and $P_j \xrightarrow{(\alpha,r+\varepsilon)} P_k$, for

some P_k, where the edges give the action/rate pair, specifying a label that identifies the activity and the rate of an exponential distribution determining its duration, with $r > 0$ and $\varepsilon > 0$. Then, these processes cannot be aggregated with either EFL or OFL because $\varepsilon > 0$ does not make them isomorphic. However, there exists a perturbation on the parameters of P_i and P_j that makes them isomorphic. For instance, one can take $P_j \xrightarrow{(\alpha,r)} P_k$ such that ε represents the degree of such perturbation. In fact, there exist infinitely many such perturbations. For instance, it would be possible to consider $P_i \xrightarrow{(\alpha,r+\varepsilon/2)} P_k$ and $P_j \xrightarrow{(\alpha,r+\varepsilon/2)} P_k$. In all these cases, it would hold that the model is ε-ordinarily fluid lumpable for any N_i and N_j. Clearly, the aggregated system will not be in exact correspondence with the original one. However, a theoretical bound shows that the aggregation error depends *linearly* in the intensity of the perturbation $|\varepsilon|$.

Exhibiting such near-symmetries may appear quite limiting for practical applications; however, there are models in the literature that do exhibit this characteristic. This has been recently studied also in [27], where a similar notion of approximate aggregation has been presented.

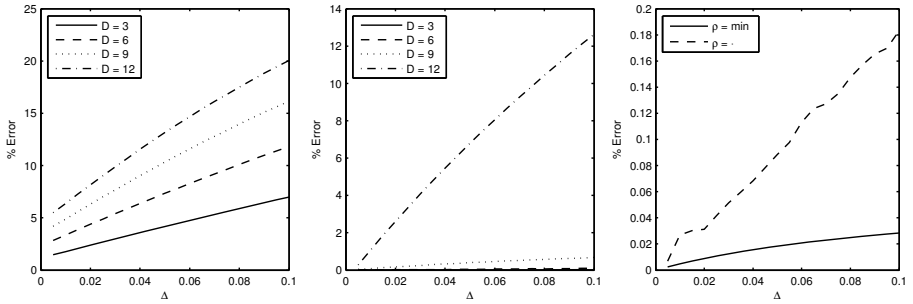
Characterisation of ODE Aggregations. When the aggregation is induced by a process algebra, it is possible to study the nature of such aggregation in two main ways.

1. The ODE aggregations can be induced by suitable notions of behavioural equivalence, which turn out to be congruences with respect to the parallel operator of Fluid Process Algebra.
2. We consider the nonrestrictive (syntactic) notion of model well-posedness originally defined in [52]. Under this assumption, processes which can be aggregated according to either EFL or OFL are related by *semi-isomorphism*. This is an extension of graph isomorphism to labelled transition systems with transition multi-sets, which does not distinguish between the multiplicity of arcs connecting two nodes whenever the total rate is the same. Furthermore, under well-posedness it holds that ε-EFL and ε-OFL imply the behavioural notion of ε-semi-isomorphism, the natural extension of semi-isomorphism which relates graphs up to changes in the transition rates. At the same time, however, processes that are semi-isomorphic cannot be aggregated according to EFL or OFL in general, essentially because two semi-isomorphic processes may be present in different contexts, which may impact their ODE expressions due to possibly different synchronisations.

5.2 Aggregation Error

To provide some numerical evidence of the aggregation error introduced by ε-EFL and ε-OFL, let us consider the model in (6) where the sequential components are defined, for $1 \leq d \leq D$, as

$$P_d \stackrel{def}{=} (\alpha, r_d).P'_d \qquad P'_d \stackrel{def}{=} (\beta, s).P_d \qquad Q \stackrel{def}{=} (\alpha, r).Q' \qquad Q' \stackrel{def}{=} (\gamma, w).Q \ . \tag{7}$$



(a) ε-EFL for $\rho = \min$ **(b)** ε-EFL for $\rho = \cdot$ **(c)** ε-OFL, for $D = 12$

Fig. 3. Numerical evaluation of ε-lumpability

These model two agents, P_d and Q, which cycle through states P'_d and Q', respectively. With this choice, the model can be interpreted as a high-level description of a *multi-class* service system where one resource, modelled by Q, can be accessed by different classes of clients, P_d, each with its own service demand characterized by r_d. We arbitrarily chose the rates of the independent actions, fixing $s = 0.5$ and $w = 15.0$, while we varied the values of r_d.

Our intent is to approximate ODEs systems where P_i and P_j are aggregated for some $1 \leq i, j \leq D$. Thus, in order to obtain non-isomorphic sequential components, we made r_d dependent on $1 \leq d \leq D$, setting $r_d = 1.0 + (d-1)\Delta$. Here, Δ is a parameter that was varied between 0.0005 and 0.1000 at 0.005 steps in our tests. In this way, it is directly proportional to the intensity of the perturbation. For instance, in a model with $D = 12$ and $\Delta = 0.1000$, we have $r_{10}/r_1 = 2.1$, showing a non-negligible difference between the rate parameters of P_1 and those of P_{10}. In order to enforce asymmetry also in the initial populations, we made the initial populations of P_d components dependent on d. Specifically, we considered initial conditions defined as $V_{P_d}(0) = 200 + (d-1)$, $V_{P'_d}(0) = 0$, $V_Q(0) = 400$, and $V_{Q'}(0) = 0$; thus, the components have initial populations separated by a few percent. For evaluating both ε-EFL and ε-OFL, we considered a perturbed model where r_d in (7) was made independent of d and set equal to the average value in the original model, i.e.,

$$\tilde{r}_d = 1.0 + (\Delta/D) \sum_{d=1}^{D} (d-1).$$

In such a perturbed model, all P_d sequential components are now isomorphic.

Assessment of ε-EFL We considered different values of D to numerically evaluate the impact of different initial conditions on the quality of the aggregation of ε-EFL. Specifically, we set $D = 3, 6, 9, 12$. Let us recall that (6) has $2D + 2$ ODEs. For each value of D and Δ, the model solution was compared against that of the perturbed model with the initial conditions set as follows: $V^{\varepsilon}_{P_d}(0) =$

$200 + (1/D)\sum_{d=1}^{D}(d-1)$, $V^{\varepsilon}_{P'_d}(0) = 0$, $V^{\varepsilon}_{Q}(0) = 400$, and $V^{\varepsilon}_{Q'}(0) = 0$. In this way, the initial population of P_d sequential components is made independent from d and is set equal to the average initial population across d, similarly to what done for the perturbation on r_d. It follows that, in the perturbed model, $\{\{P_1, \ldots, P_D\}, \{Q\}\}$ is an exactly fluid lumpable partition. Hence, the original model and the perturbed one are related by ε-EFL. Both models were solved over the time interval $[0; 100]$, so as to ensure convergence of the ODE solution to equilibrium for all parameterisations considered. Solutions were registered at 0.2 time steps. The approximation relative error for ε-EFL is as:

$$100 \times \max_{t \in \{0, 0.02, \ldots, 100\}} \max_{S \in \{P_1, \ldots, P_d, Q\}} \frac{|V_S(t) - V^{\varepsilon}_S(t)|}{V_S(0)},$$

where $V_S(t)$ is the solution of the original model and $V^{\varepsilon}_S(t)$ is the corresponding solution in the perturbed one. The absolute difference is normalised with respect to the total population of the component.

The results are presented in Figures 3a and 3b, for two distinct interpretations of the synchronisation operator. The first one defines synchronisation as the minimum of the rates of the synchronising components ($\rho = \min$) while the second one takes the product of their rates ($\rho = \cdot$). In both cases, it is possible to observe a linear growth of the error as a function of the perturbation Δ. For any fixed D, the case $\rho = \cdot$ yields more accurate aggregates than $\rho = \min$, with particularly small errors for $D = 3, 6, 9$. These tests show that even non-negligible perturbations (i.e., up to Δ ca 0.04) can produce acceptable errors (i.e., less than 10%) in practice.

Assessment of ε-OFL Analogous tests were performed for the assessment of ε-OFL, since in the perturbed model $\{\{P_1, \ldots, P_D\}, \{Q\}\}$ is also an ordinarily fluid lumpable partition. We analysed only the case $D = 12$, which yielded the worst accuracy in ε-EFL; the other cases showed the same errors (up to numerical precision of the ODE solver). A different error metric was used, to reflect the fact that OFL involves sums of ODE solutions of the unaggregated model. The approximation relative error is defined as:

$$100 \times \max_{t \in \{0, 0.02, \ldots, 100\}} \max \left\{ \frac{\left|\sum_{d=1}^{D} V_{P_d}(t) - W^{\varepsilon}_P(t)\right|}{\sum_{d=1}^{D} V_{P_d}(0)}, \frac{|V_Q(t) - W^{\varepsilon}_Q(t)|}{V_Q(0)} \right\}.$$

The numerical results are shown in Figure 3c. Overall, both for $\rho = \min$ and $\rho = \cdot$, the ε-OFL appears to be much more robust, with negligible errors across all values of Δ.

6 Performance Aware Ensembles

We present performance aware ensembles in the context of the cloud case study from Chapter IV.3 [38]. We assume a heterogeneous cloud where mobile devices,

such as smart phones, can offload computationally intensive applications to the nearby available computational nodes to improve battery lifetime [7]. To elaborate the example, we use the DEECo component model [11], 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 among the knowledge elements.

6.1 Scenario Description

The scenario elaborated in this section is that of a person 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 a cloud server machine located in the bus itself, and to save battery, it offloads the most computationally intensive tasks to that machine. Later, when the bus approaches its destination, the 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 server, provided by the terminal authority, and move some of the 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 migrating parts of the application to a different computer.

For our example, we assume that the application has a frontend component that cannot be migrated (such as the user interface, which obviously has to stay with the user, Af in our example) and a backend component that can be offloaded (typically the computationally intensive tasks, Ab in our example). Figure 4 depicts the adaptation architecture (the used notation is that of component systems, except for interfaces which are based on exchanging knowledge rather than invoking methods, various types of arrows denote various instances of interaction through knowledge described next).

6.2 Adaptation Architecture Components

The adaptation architecture on Figure 4 forms an overlay that reflects the application architecture. Central to the adaptation architecture is the Planner component, responsible for computing the optimum application component deployment. The Planner relies on Monitor components to provide information about application performance – each Monitor is a surrogate of one application component on one machine. The machines are represented by Device components. In more detail:

Planner. Each adaptive application is managed by a Planner component, whose implementation includes the application adaptation preferences. Specifically, given the alternatives for deploying each of the application components, the Planner selects the application deployment that best satisfies the preferences. We assume that the resulting deployment is described by a deployment plan,

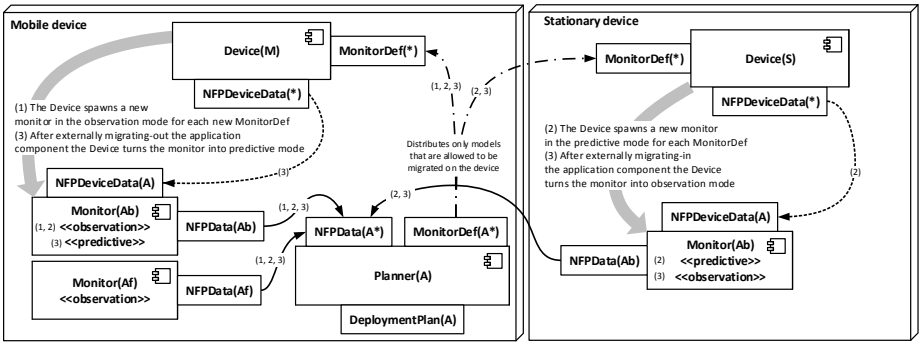


Fig. 4. Adaptation architecture. Numbers denote adaptation phases, 1 for Ab located at M, 2 for S discovered, 3 for Ab migrated to S.

and an external mechanism is responsible for performing the adaptation (for example by migrating components) as directed by the plan. To select among the available alternatives, the Planner is provided with data on non-functional properties (such as estimated frame rate or power consumption) that characterize the performance of the corresponding application component in a particular deployment (NFPData). The Planner also advertises definitions of Monitors for individual application components (MonitorDef).

Monitor. A single Monitor component exists for every application component in every deployment alternative. The Monitor component is responsible for providing NFPData for that particular combination of component and alternative. Depending on the actual deployment of the corresponding application component, the Monitor operates in one of two modes:

- The Monitor is in the *observation mode* if it resides on the same machine as the corresponding application component and therefore can observe the actual component execution. In this mode, NFPData is obtained by performance measurement of the running application component.
- The Monitor is in the *predictive mode* if it resides on a different machine than the one where the application component currently executes, and therefore represents a potential deployment alternative. NFPData is estimated from machine parameters in NFPDeviceData (such as estimating frame rate from the CPU and GPU parameters as a function $CPU \times GPU \rightarrow FPS$). In other words, the Monitor roughly predicts the performance that the application component would exhibit if it were deployed on a particular machine, relying on machine-specific data passed in NFPDeviceData.

Device. Each machine is represented by the Device component, which is responsible for instantiating Monitors advertised by newly discovered Planners and providing NFPDeviceData for Monitors operating in the predictive mode.

6.3 Adaptation Architecture Ensembles

In the assumed scenario, 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 4):

Planner and Device(s). Each Planner is a coordinator of an ensemble that distributes MonitorDefs (including the performance prediction model) of application components to Devices representing currently available machines (including the one the Planner is running on). The Planner is able to limit which MonitorDefs should be distributed to which Devices (effectively constraining the potential migration destinations for a particular application component).

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 managed 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 predictive mode residing on the corresponding machines.

6.4 Adaptation Interaction Example

Initially (phase 1, Figure 4), the ensemble distributes the MonitorDefs of both Af and Ab from Planner(A) to the Device(M) component of the mobile device, which subsequently spawns Monitors for both components and sets them to the observation mode. The Monitors start measuring NFPData of the locally executing components, which are eventually aggregated and delivered as knowledge to the Planner. So far, no deployment alternatives are discovered.

After the stationary device is discovered (phase 2, Figure 4), the ensemble propagates MonitorDefs of the components that could be (potentially) migrated (here only Ab) to the Device(S) component, which spawns a new Monitor. Because Ab is deployed on Device(M), this Monitor runs in the predictive mode. The Device(S) component feeds the Monitor with NFPDeviceData and, based on this NFPDeviceData and the performance prediction model of Ab, the Monitor produces NFPData describing the expected performance of Ab on S. Consequently, another ensemble delivers 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 (the one currently executing on M and the one modeled on S) and, assuming the adaptation is perceived as beneficial, decides to deploy Ab on the stationary device.

After Ab is migrated to the stationary device (phase 3, Figure 4), the Monitor on S switches to the observation mode. In turn, the Monitor on M is set to the predictive mode and the whole monitoring and planning process repeats.

When further stationary devices are discovered, new Monitors in the predictive mode are spawned, eventually providing new deployment alternatives for

consideration by the Planner. Disappearing devices are handled similarly (but the overlay does not tackle state loss).

7 Designing Performance-Based Adaptation

The dynamic membership and communication features of ensembles, the formal methods of expressing and evaluating performance properties, and the availability of dynamic instrumentation at implementation level are all elements that contribute to the support for building adaptive applications. Complementing these elements is a method for designing adaptation strategies – the Invariant Refinement Method for Self-Adaptation (IRM-SA), described in detail in Chapter III.4 [12]. IRM-SA is an extension to IRM [32] and guides the design of an application from high-level strategic goals and (performance) requirements to their realization in terms of system architecture with design choices that correspond to different adaptation alternatives.

Design with IRM-SA captures the high-level system goals and requirements in terms of interaction *invariants*. The invariants describe the desired state of the system-to-be at every time instant, and, in general, are to be maintained by the cooperation of the system elements (actors, components, ensembles). 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. In a sequence of design decisions, the identified top-level invariants are decomposed into combinations of more specific 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 component computation (processes) and interaction (ensembles), respectively.

To facilitate design with alternatives, IRM-SA features two decomposition types, *AND-decomposition* and *OR-decomposition*. The AND-decomposition is 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 can guarantee the parent invariant:

1. $I_{s1} \wedge ... \wedge I_{sn} \Rightarrow I_p$ (entailment)
2. $I_{s1} \wedge ... \wedge I_{sn} \not\Rightarrow false$ (consistency)

For the OR-decomposition, in the context of adaptation alternatives, we introduce the concept of *situations*. A situation is a state that the 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 choice is under the control of the running software.

The OR-decomposition is used for invariants that can be decomposed into two or more sub-invariants, with each sub-invariant corresponding to a different situation. The OR-decomposition of a parent invariant I_p into two or more sub-invariants I_{s1} ... I_{sn} is correct if in any situation (corresponding to some of the invariants I_{s1} ... I_{sn}) there is at least one sub-invariant that refines the parent invariant I_p. It is important to notice that the situations identified and elaborated in an OR-decomposition can potentially overlap. Overlapping of situations can add to the overall robustness of the system, as it essentially means that more than one design solution 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 other ones.

Technically, each situation in the IRM-SA graph is associated with one or more assumptions (see Figure 5). These assumptions describe the conditions that are expected to hold under a given situation in a declarative way, and can in fact be understood as evaluation conditions or adaptation triggers for a given situation. The formalism used for describing the assumptions depends on the nature of the assumptions, especially on whether the assumption conditions can be observed and quantified.

7.1 Scenario Description

To illustrate the IRM-SA method, we return to the cloud case study and the computation offloading example. In the case study, multiple heterogeneous network nodes form an open cloud platform that runs user applications, some possibly computationally intensive. When such an application executes on a mobile device, it can take advantage of the nearby cloud nodes by offloading the computationally intensive processing to those nodes. These nodes can even be a part of a traditional cloud infrastructure, leased on demand when there is a need for computational resources. The general assumptions are that (i) the application can be partitioned to run on multiple nodes, and (ii) a mechanism for effectively migrating application components across cloud nodes exists. Given this scenario, the goal of the system-to-be is two-fold: (1) to guarantee an upper limit in the response time observed by the user; (2) to guarantee that the application components are distributed in line with the maximum capacity constraints and load of each node.

Figure 5 shows a possible IRM-SA 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 (left alternative from top) invariant (1) is decomposed into one assumption (2) and two invariants, (3) and (4), following Figure 5. Assumption (2) specifies that *"Mobile nodes have enough capacity to handle application*

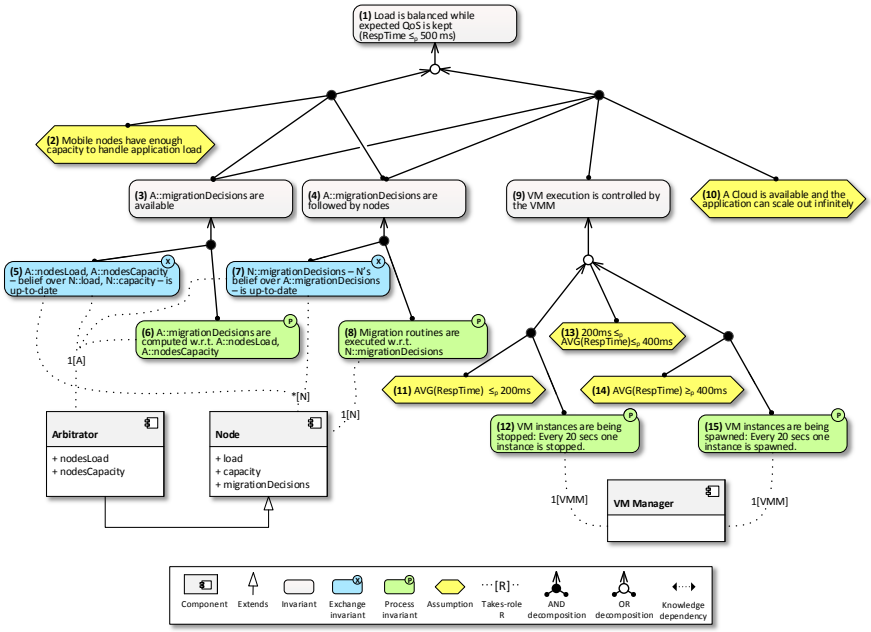


Fig. 5. Cloud case study with situations – IRM-SA graph

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 has to be included for design completeness and consistency. Invariants (3) and (4) are further refined into lower level semantics which specify the architecture of the load balancing mechanism. Specifically, the Arbitrator component (which is a specialization of the Node component, indicating that it contributes both to the invariants it takes a role in and to its parent's invariants, and inherits the knowledge of its parent) implements the load balancing logic by acquiring a view over every node's capacity and load (5), and devising a migration plan (6). In order for the migration plan to be followed, it has to be distributed to all nodes (7) and executed in every node separately (8).

In the second case (right alternative from top) invariant (1) is now decomposed into one assumption (10), and three invariants, (3), (4), and (9). Whereas invariants (3) and (4), which describe the load balancing mechanism, are shared between the two situations, invariant (9) is local to the second situation. In particular, invariant (9) specifies that virtual machine execution is controlled by the VM Manager component (VMM) in a manner described by the sub-invariants (12) and (15).

Here, three situations are distinguished, depending on the response time of the application: Low RespTime, Normal RespTime, and High RespTime, each as-

sociated with a different assumption regarding the average response time over some period in time (assumptions (11), (13) and (14) respectively). These 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 as SPL formulas, which can be evaluated at runtime using the SPL engine in conjunction with DiSL, as exemplified in Listing 1. 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 (11), the VMM needs to react by stopping virtual machines (12); if it is more than 400 ms (14), the VMM has to start new virtual machines (15); otherwise (13), do nothing.

7.2 Transforming Design into Code

IRM-SA is tailored towards producing system designs for DEECo [11]. The leaves of the IRM-SA graph can be process invariants, exchange invariants or assumptions. The first two types are mapped to the DEECo concepts of processes and ensembles, respectively. For assumptions, we distinguish between the ones that can be formally specified, observed and verified, and the informal ones, typically specified in natural language. Whereas informal assumptions cannot be checked at runtime (thus we optimistically assume that they hold during system execution), formal assumptions are checked at runtime by mapping them to runtime monitors.

When the formal assumptions concern performance, they can be specified using SPL. This is especially useful when the assumptions concern alternative decompositions that model different adaptation strategies – the performance properties described in SPL can be used to identify what situation the system is currently in, and to react accordingly (e.g. invariants (11), (13) and (14) on Figure 5). SPL can also specify performance invariants that are checked to validate the design (e.g. invariant (1) on Figure 5). Finally, the presence of SPL formulas in the IRM-SA graph gives an early (design time) indication of the need for monitoring, whose potential overhead needs to be balanced against the adaptation capabilities.

8 Summary

Besides dealing with many individual challenges inherent to the construction of collective autonomic systems, the ASCENS project also examines the overall lifecycle of such systems, considering where and how the proposed individual solutions interact and complement each other. Chapter III.1 [24] describes this perspective in general terms, introducing the concept of continuous development lifecycle, where repeated design and runtime activities interact with each other through deployment and feedback to manage system evolution.

In this chapter, we present the support for performance awareness in the same lifecycle context – starting with the runtime cycle, where instrumentation is used to monitor performance relevant system properties (Section 2), which are

evaluated (Section 3) by the system implementation (Section 4). Both the design and the runtime cycles may reflect on system performance through modeling (Section 5), the design cycle also provides the concept of dynamic ensembles to structure the implementation (Section 6), which can be architected by gradual refinement from the initial system requirements (Section 7).

References

1. Aoki, M.: Control of large-scale dynamic systems by aggregation. IEEE Trans. Autom. Control 13(3) (1968)
2. ASM (2014), `http://asm.ow2.org/`
3. Barendregt, H.: The Lambda Calculus: Its Syntax and Semantics. Mathematical Programming Study. North-Holland Publishing Company, Amsterdam (1984)
4. Bruni, R., Corradini, A., Gadducci, F., Hölzl, M., Lafuente, A.L., Vandin, A., Wirsing, M.: Reconciling White-Box and Black-Box Perspectives on Behavioral Self-adaptation. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems. LNCS, vol. 8998, pp. 163–184. Springer, Heidelberg (2015)
5. Buchholz, P.: Exact and ordinary lumpability in finite Markov chains. Journal of Applied Probability 31(1) (1994)
6. Bulej, L., Bureš, T., Horký, V., Keznikl, J., Tůma, P.: Performance awareness in component systems: Vision paper. In: Proc. COMPSAC 2012 CORCS (2012)
7. Bulej, L., Bureš, T., Horký, V., Keznikl, J.: 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), ACM Press, New York (2013)
8. Bulej, L., Bureš, T., Horký, V., Kotrč, J., Marek, L., Trojánek, T., Tůma, P.: SPL: Unit testing performance. Tech. Rep. D3S-TR-2014-04, Dep. of Distributed and Dependable Systems, Charles University in Prague (2014)
9. Bulej, L., Bureš, T., Keznikl, J., Koubková, A., Podzimek, A., Tůma, P.: Capturing performance assumptions using stochastic performance logic. In: Proc. ICPE 2012, ACM Press, New York (2012)
10. Bureš, T., Horký, V., Kit, M., Marek, L., Tůma, P.: Towards performance-aware engineering of autonomic component ensembles. In: Margaria, T., Steffen, B. (eds.) ISoLA 2014, Part I. LNCS, vol. 8802, pp. 131–146. Springer, Heidelberg (2014)
11. Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F.: DEECo – an ensemble-based component system. In: Proc. of the International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE '13), Vancouver, Canada, ACM, New York (2013)
12. Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F.: The Invariant Refinement Method. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems. LNCS, vol. 8998, pp. 405–428. Springer, Heidelberg (2015)
13. Cantrill, B.M., Shapiro, M.W., Leventhal, A.H.: Dynamic instrumentation of production systems. In: Proceedings of the USENIX Annual Technical Conference (ATC'04), Berkeley, CA, USA (2004)
14. Cardelli, L.: On process rate semantics. Theor. Comput. Sci. 391 (2008)

15. Chiba, S.: Load-time structural reflection in Java. In: Bertino, E. (ed.) ECOOP 2000. LNCS, vol. 1850, p. 313. Springer, Heidelberg (2000)
16. Ciocchetta, F., Hillston, J.: Bio-PEPA: A framework for the modelling and analysis of biological systems. Theor. Comput. Sci. 410(33–34) (2009)
17. Clark, M.: JUnitPerf (2014), http://www.clarkware.com/software/JUnitPerf
18. Hayden, R.A., Bradley, J.T.: A fluid analysis framework for a Markovian process algebra. Theor. Comput. Sci. 411(22-24) (2010)
19. Herbst, N.R., Huber, N., Kounev, S., Amrehn, E.: Self-adaptive workload classification and forecasting for proactive resource provisioning. In: Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE '13), ACM Press, New York (2013)
20. Hermanns, H., Rettelbach, M.: Syntax, semantics, equivalences, and axioms for MTIPP. In: Proceedings of Process Algebra and Probabilistic Methods, Erlangen (1994)
21. Hillston, J.: Fluid flow approximation of PEPA models. In: Proceedings of Quantitative Evaluation of Systems, IEEE Computer Society Press, Los Alamitos (2005)
22. Hillston, J.: A compositional approach to performance modelling. Cambridge University Press, New York (1996)
23. Hölzl, M., Gabor, T.: Reasoning and Learning for Awareness and Adaptation. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems. LNCS, vol. 8998, pp. 249–290. Springer, Heidelberg (2015)
24. Hölzl, M., Koch, N., Puviani, M., Wirsing, M., Zambonelli, F.: The Ensemble Development Life Cycle and Best Practices for Collective Autonomic Systems. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems. LNCS, vol. 8998, pp. 325–354. Springer, Heidelberg (2015)
25. Horký, V., Haas, F., Kotrč, J., Lacina, M., Tůma, P.: Performance regression unit testing: a case study. In: Balsamo, M.S., Knottenbelt, W.J., Marin, A. (eds.) EPEW 2013. LNCS, vol. 8168, pp. 149–163. Springer, Heidelberg (2013)
26. Horký, V., Libič, P., Marek, L., Steinhauser, A., Tůma, P.: Utilizing performance unit tests to increase performance awareness. In: Proc. ICPE 2015, ACM Press, New York (2015)
27. Iacobelli, G., Tribastone, M.: Lumpability of fluid models with heterogeneous agent types. In: DSN (2013)
28. Iwase, Y., Levin, S.A., Andreasen, V.: Aggregation in model ecosystems I: perfect aggregation. Ecological Modelling 37 (1987)
29. JDOM Library (2013), http://www.jdom.org
30. Kalibera, T., Bulej, L., Tůma, P.: Benchmark precision and random initial state. In: Proc. SPECTS 2005, pp. 853–862. SCS (2005)
31. Kalibera, T., Bulej, L., Tuma, P.: Automated detection of performance regressions: the Mono experience. In: 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, Sep. 2005, IEEE Computer Society Press, Los Alamitos (2005)
32. Keznikl, J., Bures, T., Plasil, F., Gerostathopoulos, I., Hnetynka, P., Hoch, N.: 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, ACM, New York (2013)
33. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, p. 327. Springer, Heidelberg (2001)

34. Kwiatkowski, M., Stark, I.: The continuous π-calculus: A process algebra for biochemical modelling. In: Heiner, M., Uhrmacher, A.M. (eds.) CMSB 2008. LNCS (LNBI), vol. 5307, pp. 103–122. Springer, Heidelberg (2008)
35. Marek, L., Zheng, Y., Ansaloni, D., Bulej, L., Sarimbekov, A., Binder, W., Tůma, P.: Introduction to dynamic program analysis with DiSL. Science of Computer Programming (2014)
36. Marek, L., Zhen, Y., Binder, W.: DiSL (2012), http://d3s.mff.cuni.cz/software/disl
37. Marek, L., Zheng, Y., Ansaloni, D., Binder, W., Qi, Z., Tuma, P.: DiSL: An extensible language for efficient and comprehensive dynamic program analysis. In: Proc. 7th Workshop on Domain-Specific Aspect Languages (DSAL '12), ACM Press, New York (2012)
38. Mayer, P., Velasco, J., Klarl, A., Hennicker, R., Puviani, M., Tiezzi, F., Pugliese, R., Keznikl, J., Bureš, T.: The Autonomic Cloud. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems. LNCS, vol. 8998, pp. 495–512. Springer, Heidelberg (2015)
39. Milner, R.: Communication and Concurrency. Prentice-Hall, Inc., Upper Saddle River (1989)
40. Mytkowicz, T., Diwan, A., Hauswirth, M., Sweeney, P.F.: Producing wrong data without doing anything obviously wrong. In: Proceedings of ASPLOS 2009, ACM Press, New York (2009)
41. De Nicola, R., Latella, D., Lafuente, A.L., Loreti, M., Margheri, A., Massink, M., Morichetta, A., Pugliese, R., Tiezzi, F., Vandin, A.: The SCEL Language: Design, Implementation, Verification. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems. LNCS, vol. 8998, pp. 3–71. Springer, Heidelberg (2015)
42. Okino, M.S., Mavrovouniotis, M.L.: Simplification of mathematical models of chemical reaction systems. Chemical Reviews 2(98) (1998)
43. Oracle: JVM Tool Interface (2006), http://docs.oracle.com/javase/6/docs/platform/jvmti/jvmti.html
44. Perf4J (2014), http://perf4j.codehaus.org/
45. Perl, S.E., Weihl, W.E.: Performance assertion checking. SIGOPS Oper. Syst. Rev. 27 (1993)
46. Reynolds, P., Killian, C., Wiener, J.L., Mogul, J.C., Shah, M.A., Vahdat, A.: Pip: Detecting the Unexpected in Distributed Systems. In: NSDI'06. USENIX (2006)
47. Sheskin, D.J.: Handbook of Parametric and Nonparametric Statistical Procedures. CRC Press, Boca Raton (2011)
48. SPL Tool (2013), http://d3s.mff.cuni.cz/software/spl
49. SystemTap (2014), http://sourceware.org/systemtap/
50. Tahchiev, P., Leme, F., Massol, V., Gregory, G.: JUnit in Action, 2nd edn. (2010)
51. Tribastone, M., Gilmore, S., Hillston, J.: Scalable differential analysis of process algebra models. IEEE Transactions on Software Engineering 38(1) (2012)
52. Tschaikowski, M., Tribastone, M.: Exact fluid lumpability for Markovian process algebra. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 380–394. Springer, Heidelberg (2012)
53. Tschaikowski, M., Tribastone, M.: Tackling continuous state-space explosion in a Markovian process algebra. Theoretical Computer Science 517 (2014)
54. Tschaikowski, M., Tribastone, M.: A unified framework for differential aggregations in Markovian process algebra. Journal of Logical and Algebraic Methods in Programming (2014)

55. Vetter, J.S., Worley, P.H.: Asserting Performance Expectations. In: Proc. 2002 ACM/IEEE Conf. on Supercomputing (Supercomputing '02), IEEE Computer Society Press, Los Alamitos (2002)
56. Welch, B.L.: The generalization of student's problem when several different population variances are involved. Biometrika 34(1/2) (1947)
57. Wirsing, M., Hölzl, M.M., Tribastone, M., Zambonelli, F.: ASCENS: Engineering Autonomic Service-Component Ensembles. In: Beckert, B., Damiani, F., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2011. LNCS, vol. 7542, pp. 1–24. Springer, Heidelberg (2013)