Towards Performance-Aware Engineering of Autonomic Component Ensembles

Tomáš Bureš[1, 2], Vojtěch Horký[1], Michał Kit[1], Lukáš Marek[1], Petr Tůma[1]

[1] Faculty of Mathematics and Physics
Charles University in Prague
Prague, Czech Republic
[2] Institute of Computer Science
Academy of Sciences of the Czech Republic
Prague, Czech Republic

{bures, horky, kit, marek, tuma}@d3s.mff.cuni.cz

Abstract. Ensembles of autonomic components are a novel software engineering paradigm for development of open-ended distributed highly dynamic software systems (e.g. smart cyber-physical systems). Recent research centered around the concept of ensemble-based systems resulted in design and development models that aim to systematize and simplify the engineering process of autonomic components and their ensembles. These methods highlight the importance of covering both the functional concepts and the non-functional properties, specifically performance-related aspects of the future systems. In this paper we propose an integration of the emerging techniques for performance assessment and awareness into different stages of the development process. Our goal is to aid both designers and developers of autonomic component ensembles with methods providing performance awareness throughout the entire development life cycle (including runtime).

Keywords: ensemble-based systems, component systems, performance engineering

1 Introduction

Autonomic component ensembles (ACEs) emerged in the recent years as an abstraction for modeling and constructing open-ended distributed highly dynamic systems (e.g. smart cyber-physical systems as featured by the EU H2020 program). ACEs provide a way of describing dynamic goal-oriented groups of otherwise autonomic components, which combine well the autonomic and cooperative behavior.

ACEs operate in open and partially uncertain environments. This implies that potential interactions of components and their environment are very difficult, often impossible, to fully predict. ACEs also exhibit emergent behavior, which arises from collective actions taken by interacting components.

To cope with the uncertainty and emergent behavior, the software engineering of ACEs typically relies on a dedicated software development process, e.g. Ensemble Development Lifecycle [1], which considers design time and run time as two parallel and mutually interacting adaptation loops. The loops integrate development with analysis and monitoring, thus making it possible to discover and reflect the emergent behavior and unanticipated reactions of the environment. The unanticipated behavior (e.g., the occurrence of certain message or reachability of certain state) is subsequently reported back to developers, thus providing input for incremental development.

It is relatively well-understood how to capture functional properties and detect their violations. Moreover, functional properties are (at least partially) addressed by various existing tools that aim to validate their satisfiability. The contrary, however, holds about the performance properties. This is because performance of a system cannot be easily isolated. Performance is highly impacted by resource sharing (e.g. CPU, I/O, network), even between otherwise unrelated and seemingly independent components. Additionally, due to their statistical nature, performance properties are often more complex to express and validate – performance measurements typically involve a complex setup and non-trivial statistical computations over the measured values. Recently, there have emerged helpful mechanisms for assessing performance (e.g. modeling to provide estimates during development, monitoring to provide information during execution), however, as more or less independent and unrelated approaches, they do not provide a comprehensive support being integrated within a development process. This is especially true for engineering ACEs.

In this paper, we strive to fill in this gap in the scope of the ACEs development process. Our goal is to aid both designers and developers of ACEs with methods providing performance awareness throughout the entire development life cycle (including runtime). In particular, we focus on the following three goals: (i) to discuss performance-related issues and objectives of ACEs development process, (ii) to show how the performance objectives can be targeted within the ACEs development process, and (iii) to overview suitable models, techniques and tools that together bring the performance-aware engineering of ACEs.

The structure of the paper is as follows. In Section 2, we elaborate on our running example – a scenario coming from our involvement in the ASCENS EU project. In Section 3, we detail the Ensemble Development Lifecycle, which we extend in Section 4 by performance-awareness (at both design/development time and runtime). We describe how existing tools can be used to address performance awareness in different phases of the development process and exemplify this on the running example. Section 5 surveys the related work, while Section 6 concludes the paper.

2 Case Study

To illustrate a typical representative of a distributed highly-dynamic system and its modeling using ACEs, we consider a scenario of intelligent vehicle navigation, in which vehicles are to efficiently get to given destinations, taking into account the current traffic, road closures, fuel consumption, etc. Vehicles are equipped with a route

planning utility that allows a vehicle to autonomously compute the optimal route to its destination. For this purpose, a vehicle is aware of the factors influencing the route planning (i.e. traffic situation, fuel state and average fuel consumption, etc.). Such awareness is obtained from the internet (e.g. by 4G/3G based connection) and from the vehicles nearby, by short-range peer-to-peer communication (e.g. based on IEEE 802.15.4). When internet connection is available, a vehicle delegates the route computation to a centralized cloud-hosted service (called Traffic Manager – TM), which balances the traffic on a global level. TM also acts as a primary source of traffic information.

When modeling the case study using ACEs, the vehicles as well as the TM become autonomic components. The autonomic components are dynamically grouped into ensembles (i.e. dynamic goal-oriented groups of mutually cooperating components) to fulfill joint goals. For instance, vehicles that are close to each other group into ensembles to share their awareness about the traffic. Similarly, vehicles form an ensemble with TM in their area if internet connectivity is available.

Because a particular semantics of ACEs can be considered only within a particular component model when it comes to a realization, we exemplify our case-study using the DEECo component model [2]. In DEECo, a component consists of component knowledge, which holds the state of the component and its belief about other components. For instance the knowledge of the vehicle component includes its actual geographical position, the route it needs to follow (computed either locally or remotely by

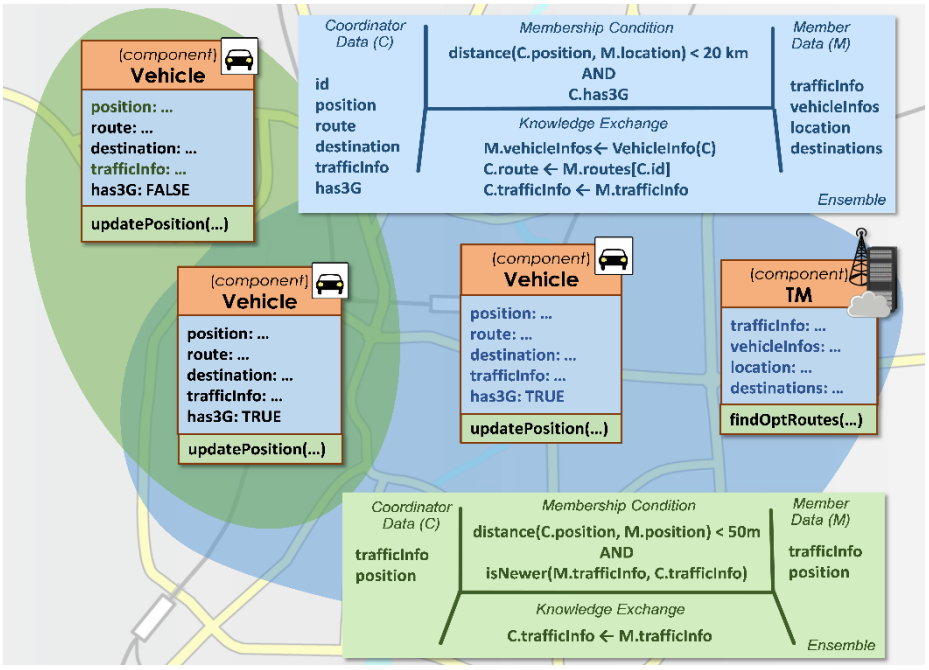

Figure 1: Modelling scenario with use of DEECo primitives. UML representation of DEECo components together with ensemble description.

```
1.  class Vehicle extends Component {
2.     public Route route;
3.     public Position position;
4.     public TrafficInfo trafficInfo;
5.     public double speed;
6.     public Position position;
7.     ...
8.     @Process
9.     @PeriodicScheduling(250)
10.    public static void updatePosition(
11.      @Out("position") Position p) {
12.      Sensors.SPEED.readCurrentPosition(position);
13.    }
14. }
```

Knowledge

Process

Figure 2: An example of the Vehicle component definition in jDEECo

the TM) and the traffic information (see Figure 1). The knowledge is continuously updated by the component processes based on their sensing and internal computation.

Ensembles in DEECo capture component composition and communication. An ensemble defines how to establish the ensemble instances – dynamic groups of components – and how to exchange knowledge among the components in a particular ensemble instance. Components to be included in an ensemble instance are determined by so called *ensemble membership condition* – a first-order logic predicate over the knowledge of components. This is exemplified in Figure 1, where the upper ensemble groups a TM with vehicles in its sphere of activity, while the lower ensemble groups a vehicle with vehicles in its close vicinity. The communication within an ensemble instance is defined by so called *knowledge exchange function*, which describes how a part of component's knowledge is transformed and stored to the knowledge of another component (see Figure 1). Defined using a relation among a number of components, an ensemble may naturally exist in a system in multiple instances.

Technically, the execution of ACEs is managed by a runtime framework (e.g. JDEECo as a runtime framework for DEECo). The runtime framework includes the necessary programming constructs for definition of components and ensembles in a particular programming language and provides the distributed infrastructure for execution of components, formation of ensembles and knowledge exchange within ensembles.

2.1 Performance Considerations

In addition to functional goals (such as vehicle navigation to its destination), ACEs are typically subjects to a number of performance goals. An example of such a high-level goal in our case-study is the time for a navigation utility to plan a suitable route. This high-level goal is further dependent on other (still high-level) goals such as the limits on the route planning time or guarantees on the traffic information propagation delays. When designing for such high-level goals, the developer needs information on multiple performance-relevant properties on lower levels of the design (for example,

the route planning time depends on the computing power available in the vehicle computer, and the traffic information propagation delays in the vehicle-to-vehicle networks depend on the information forwarding capacity of each vehicle).

The availability of this performance relevant information varies with the type of the information (e.g., settling time and accuracy of a particular sensor, execution time of a particular method) and the stage of the design. Some information is available very early in the design process, because it is actually a part of the design choices made – for example, we are likely to have a good idea of how reliable the vehicle speed information is because we know what speed sensor we use. Other performance relevant information is more difficult to obtain or guarantee – this may concern for example the frequency and accuracy of GPS position updates, which is strongly influenced by actual signal reception conditions, or the upper bound on the route calculation time, which may depend on the complexity of the map used. Finally, some performance relevant information comes from interactions among the ensemble components, which are especially difficult to predict in an open system.

For the design to progress at all, the developer has to make reasonable assumptions about all the lower level performance properties that contribute to the high-level performance goals. As a particular hallmark of the ensemble development process, relying on wrong assumptions is not necessarily a developer error – in the open environment, some initially reasonable assumptions can turn out to be wrong as the environment continuously evolves. We therefore need a development process that can track the individual performance assumptions between the design and execution phases and, as a matter of course, monitor and reflect on the possible violations of the assumptions.

3 Ensemble Development Lifecycle

The development of ACEs typically follows a dedicated life-cycle model, which in turn provides a concrete frame for supporting performance considerations. In this section, we overview the Ensemble-Development Lifecycle (EDLC), which is one of the primary life-cycle models for ACEs. Taking EDLC as the basis, we then show how it can be extended to address performance-related issues.

3.1 General model

EDLC is a dedicated lifecycle model for engineering of autonomic component ensembles. EDLC features a "double-wheel" development process (see Figure 3), which combines iterative development (captured by the "first wheel") with two-level adaptation – the autonomous self-adaptation at runtime (captured by the "second wheel") and the developer-controlled adaptation (captured by the feedback loop between the design time and runtime).

Going into more detail, the design part ("first wheel") consists of requirements engineering, modeling/programming and verification/validation. These activities are iteratively executed until a desired product is created. The verification and validation play

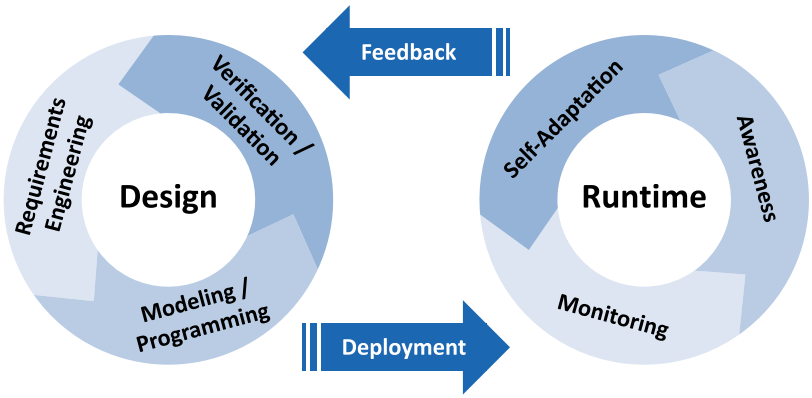

Figure 3: Ensemble Development Life Cycle

a very prominent role here. It involves static analysis and simulations, which are employed to predict the large-scale system behavior, accounting for its potentially emergent nature.

The runtime part ("second wheel") reflects the execution of the system. In contrast to traditional systems, ACEs possess a high degree of self-awareness and self-adaptivity. This is in EDLC embodied by the monitoring, awareness (reasoning) and self-adaptation activities – similar to the MAPE adaptation loops known from autonomic computing [3].

The two wheels of EDLC are connected by deployment of a system and feedback, which brings data collected by monitoring at runtime back to design. The feedback data is used to observe and analyze the behavior of the system and its self-adaptation responses in face of originally unanticipated situations. If the analysis shows that the system was not able to gracefully cope with a particular situation, it is reengineered, analyzed/simulated and again deployed.

Technically, EDLC can be supported by a number of tools – for instance DEECo can be used for modeling/programming, deployment and execution of ACEs. Similarly, DEECo can be also employed for rudimentary support for monitoring, awareness and self-adaptation. The other design activities are covered by IRM [4] / SOTA [5] (for requirements engineering) and by ARGoS [6] / GMC [7] (for verification of functional aspects).

3.2 Performance perspective

From the performance perspective, the role of the ensemble development process is (1) to collect and deliver available information about performance to developers in relevant situations and (2) to propagate the performance relevant assumptions made during development to the runtime for monitoring and adaptation feedback.

Rather than being acquired en bloc, performance information is collected and improved gradually throughout the development process. In principle, the initial information is limited to guesses about future system performance. This information can be improved along two principal axes:

Isolated computation. As soon as the initial implementation of selected ensemble components becomes available, the performance of these components can be measured in isolation, much in the same way as software components are tested in agile development methodologies. Measurement in isolation requires a test harness that manufactures the input required by the ensemble components. This task is made easier by the fact that the interface of each ensemble component is precisely specified, with communication taking place only through knowledge exchange. The input for testing therefore takes the shape of a snapshot(s) of the knowledge repository, initially prepared by knowledge generators implemented for that purpose [8].

Relying on an artificially generated workload can naturally limit the accuracy or representatives of the performance measurements collected. In the subsequent steps of the development process, this information can be improved by collecting knowledge samples from the executing ensemble. This knowledge can again be fed as input to the components for measurement purposes.

The ultimately authoritative information on individual component performance can be collected by monitoring the deployed ensemble. Thus, the obtained information can be confronted with the estimates and assumptions made in earlier development stages as necessary.

Knowledge exchange. Besides the performance of individual components, the performance relevant behavior of an ensemble is also determined by the interactions between components. These interactions determine both the content and the timing of the input knowledge that the components rely on.

Improving the initial estimates of ensemble performance requires that the development process has progressed enough to provide information on the ensemble communication architecture. Once this architecture is available, estimates on knowledge propagation delays can be made and following ensemble simulations can improve the available performance information.

As in the previous case, the ultimate information on ensemble performance comes from monitoring “live” ensembles once they become available. The entire process of performance information improvement has an iterative character, where each new contribution helps to gradually form the overall ensemble performance picture.

As a major stumbling block, we can eventually end up with too much information – either too much information to collect, with prohibitive measurement costs or disruptive measurement overhead, or too much information to process and accommodate, which can entail significant developer workload. To avoid this particular danger, it is necessary to formally track the process of refining the high-level performance goals into lower level performance assumptions or requirements. We need to monitor and collect performance information only in locations whose performance contributes to a high-level performance goal, and we need to report this performance only when it diverges from the assumptions made during development.

4 Performance-Awareness in EDLC

To bring the performance perspective into engineering of ACEs, we augment EDLC with an extension which addresses performance-related issues as discussed in Section 3.2. Overall, we view the performance-aware engineering as centered around the three principal feedback loops of EDLC (the design one, the runtime one, and the outer one connecting design and runtime). At design time, we introduce four principal activities connected to performance: (D-1) formulating high-level performance goals, (D-2) refining the high-level goals to time constraints on isolated computation and knowledge exchange (as outlined in Section 3.2), (D-3) collecting performance data by isolated benchmarking and simulations, (D-4) providing feedback about predicted performance to development of components and ensembles. At runtime, we (R-1) collect relevant performance indicators (as identified by the high-level and low-level performance goals) and we (R-2) analyze them to detect possible violations of the performance goals. When such a violation happens, (F-1) it is reported back to the design time. The outer feedback loop is additionally used for (F-2) obtaining real-life measurements (e.g. actual network latency, packet drops) to improve the design-time simulations.

This whole process is tool-supported. We use a special computer interpretable logic for capturing performance assumptions in D-1 and D-2; tools for automated performance evaluation, simulations, and analysis for D-3 and for the runtime monitoring and analysis of R-1 and R-2; and an extension to a development IDE (e.g. Eclipse) to provide relevant performance measurements as part of its contextual assistance for D-4. In the following, we overview in more detail the particular methods and tools driving the process described above. Furthermore, we demonstrate their use on our case-study.

4.1 Performance goals and their decomposition (D-1, D-2)

High-level performance goals formulation and their further decomposition are the activities that fall into the requirements engineering step of the EDLC. For this task we employ the Invariant Refinement Method (IRM) [4], which has already been used in DEECo for functional goal formulation and decomposition. IRM relies on the top-down approach, where top-level invariants constitute high-level (general) goals of the application and are further decomposed into more specialized (fine-grained) ones, which eventually map into concrete component processes and ensembles. In the context of the EDLC, the refinement of non-functional goals is not much different from the refinement of their functional counterparts and can use IRM as is. Similar to functional goals, performance goals are eventually mapped to component processes or knowledge exchange of the ACEs.

To illustrate the idea behind activities D-1 and D-2, we take a high-level performance goal of not needing more than 30 seconds to provide a suitable route. Following the functional IRM-based decomposition, this goal splits into two alternatives (OR-decomposition) as follows: If the vehicle has no connectivity to the TM, it computes locally the route to its destination, (optionally) relying on the traffic information obtained previously from TM or from other vehicles. Otherwise, the vehicle off-loads

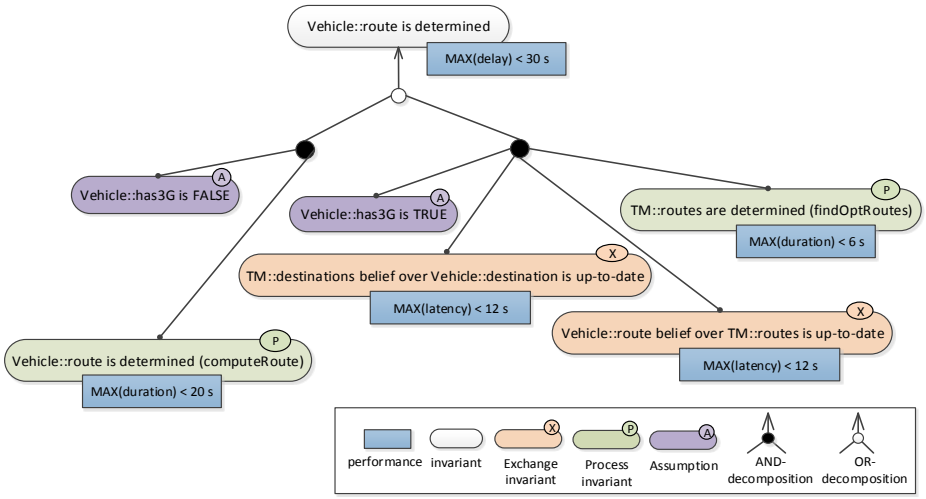

Figure 4: High-level performance goal decomposition in IRM

route computation to the TM and awaits the results. In this case, the high-level performance goal of vehicle navigation planning time being no longer than 30 seconds decomposes into several time constraints (as shown in Figure 4) that correspond to knowledge exchange (in both directions) and TM's route computation.

4.2 Formalization of performance constraints (D-2)

We capture performance goals by Stochastic Performance Logic (SPL) [9], which is a many-sorted first-order logic with well-defined semantics. SPL regards performance as a random variable with probability distribution dependent on a given workload. SPL features performance relational operators, which are based on statistical testing of various statistical measures such as mean, minimum, maximum or an arbitrary quantiles.

We employ SPL as the formal framework for expressing performance goals at design time as exemplified by rectangular boxes in Figure 4 (note that the $<$ operator used in the formulas stands for single-sided statistical testing whether a hypothesis of a negation can be rejected at a given confidence level α). Similarly, we use SPL on the level of the code, where we reflect the performance requirements in the form of annotations (`@Performance`) – see Figure 5. Tying performance goals to particular methods in the code brings the performance goals to the level where they can be automatically tested.

4.3 Benchmarking of isolated computation (D-3)

The SPL-based code annotations can be used at run-time to check that the implementation conforms to the specification and at development time to test the computation performance of components in isolation. Testing in isolation allows getting rough estimates of the performance in situations when an application is not ready to be deployed or when real deployment is too costly to be used for testing.

```
class TM extends Component {
  ...
  @Process
  @Performance("MAX(duration) < 6s")
  public static void findOptRoutes(
    @In("vehicleInfos") List<VehicleInfo> vehicleInfos,
    @In("trafficInfo") TrafficInfo trafficInfo,
    @InOut("routes") Map<String, Route> routes)
  { /* ... */ }
  ...
}

@Ensemble
class VehicleTM {
  ...
  @Performance("MAX(latency) < 12s")
  @KnowledgeExchange
  public static void exchange(
    @In("coord.id") String coordId,
    @In("coord.destination") Position coordDest,
    ...
  )
  { /* ... */ }
}
```

Figure 5: An example of requirement decomposition

Testing of isolated execution requires a developer (tester) to provide a sample workload. In traditional performance unit testing as described in [9], the tester needs to prepare a workload generator that creates the parameters for the method under test. When testing DEECo components, the tester has to provide artificial knowledge upon which the component can operate. This can be done by providing test-cases that are partitioned in a similar way as in functional black-box and white-box testing. Alternatively, it is possible to use knowledge valuation sampled previously in a real deployment.

The performance testing process itself and the evaluation of the results is driven by SPL tools [10], which take care of all steps necessary for precise and statistically relevant performance measurements. This involves workload preparation, actual measurements preceded by a sufficiently long warm-up, collection of measurement results and their statistical analysis. To improve the relevancy of the measurements, they can be collected on a remote machine, running the actual target hardware, instead of a local, developer's one. SPL tools also allow for regression testing, which makes it possible to detect performance degradation across software versions.

4.4 Benchmarking of knowledge exchange (D-3)

Contrary to computational performance, the performance of knowledge exchange has to be established on a system level – at least considering components of an ensemble and other components that use the same shared communication medium.

An important fact is that the performance of the knowledge exchange depends heavily on the particular communication protocols being used. For instance, to date DEECo features two principal knowledge exchange approaches – centralized tuple space [11] and decentralized gossip-based communication [12]. In the former case, every time a process needs to be executed a remote tuple space is queried for the necessary data and the result is stored back immediately after the execution. In the latter case, knowledge is exchanged asynchronously in a best-effort manner. Naturally, the first option brings relatively fast knowledge exchange, which comes at the price of requiring a stable (and reliable) network infrastructure. Gossiping on the other hand seamlessly supports unreliable and continuously changing communication links (e.g. Mobile Ad-hoc NETworks – MANETs), which is carried by the cost of longer (by several orders of magnitude) communication times and weaker consistency of the whole system.

To predict the times for the knowledge exchange, we rely on simulations that take into account the realistic behavior of the network. In particular, in the frame of the DEECo component model, its runtime framework JDEECo supports integration with the OMNet++ network simulator, which is utilized to simulate network contention in static, wireless, mobile and MANET networks. This integration of JDEECo and OMNet++ makes it possible to gather different statistics (e.g. amount of packets exchanged, amount of drops, and latencies).

An obvious difficulty of the system-level simulation is that it requires a model of the infrastructure nodes (that act as component containers) and their network connectivity. Specifying such a model requires a non-trivial effort. Advantageously, once created, this model can be reused across development increments and possibly even across different ACEs applications. What then remains as an input for the simulation is the deployment plan (i.e. assignment of components to particular infrastructure nodes). Such a deployment plan can be specified in a relatively straightforward way – e.g. by assigning an instance of a particular component to each node of a specific type (e.g. Vehicle).

4.5 Collection and analysis of performance indicators at runtime (R-1, R-2)

In addition to isolated measurements and simulations at design time, we monitor ACEs also at runtime. This is performed using DiSL [13], which is an instrumentation framework targeted on dynamic analysis of applications. DiSL provides an AOP (Aspect Oriented Programming) inspired domain specific language hosted in Java using annotations, which makes it possible to insert arbitrary instrumentation into an observed application.

In our approach, we instrument component processes marked by the `@Performance` annotation. DiSL also allows for inside-process instrumentations (e.g. for measuring performance of a particular method or block of statements). Since the instrumentation in DiSL is on-line, meaning it is dynamically applied when the application is loaded, it can be easily switched off when no measurements are required, thus mitigating the runtime overhead. The results from DiSL measurement are stored as an online profile of a component. The online profile can be immediately evaluated by SPL backend and submitted to a component developer if some of the performance requirements are not met (as described in Section 4.6).

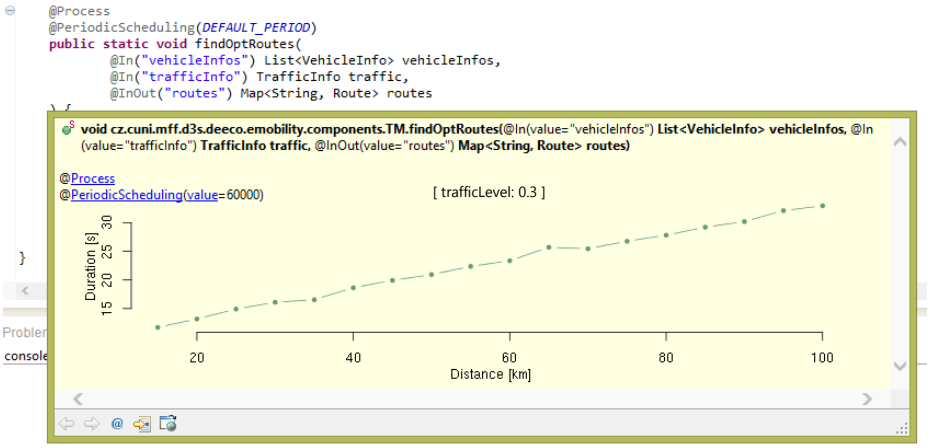

Figure 6: Prospective performance feedback integration into Eclipse IDE

4.6 Providing development feedback (D-4)

We envision that the measurement of computation and knowledge exchange performance should be available to the developer directly in the IDE and similarly to the way results of unit tests are reported and context help is shown in the context assist. This idea is exemplified in Figure 6 – it shows a mock-up of the Eclipse IDE with a context-assist displaying a graph of measured process performance.

Moreover, we envision that the process of providing the results to the IDE should resemble the one applied in continuous integration. In particular, we imagine that the performance measurements are triggered as soon as a particular artifact (component or ensemble) becomes available in a shape allowing for its deployment. This happens asynchronously (most likely on a dedicated server). Although their primary objective is to give a pass/fail answer (according to the specified performance goals), they can be used to determine which performance indicators are of relevance and their detailed statistics can be provided within the IDE.

5 Related Work

Being a relatively young concept, the performance of ACEs has not been so far systematically addressed; in particular, there are no existing works addressing performance in the context of the ACEs development process. Looking at our contribution from a broader perspective, we can identify three main research fields, which are at least partially related to our contribution (though they are not specialized for development of ACEs). These are: performance measurement frameworks used broadly in the context of component-based systems, instrumentation tools for computation time measurements, and approaches for communication latency assessment. We structure the rest of the section along this principal division.

In regard to performance measurement frameworks, in [14] authors present performance measurement framework designed for component systems called TAU. TAU provides a support for two kinds of instrumentation techniques that differ with respect to the flexibility level being traded off for a higher overhead. The later work (described in [15]) extends the TAU framework by performance-oriented regression tests. As a complete framework, TAU delivers a broad range of features, which in the case of the ACEs and the approach proposed in this paper seems to be redundant (introducing unnecessary overhead). The Palladio component model [16] comes with a simulation framework that allows for identification of performance bottlenecks during the design phase of the development process. Being a pure model-based solution, Palladio does not support on-line measurements of a (partially-) developed system, and relies only on individual component performance predictions. This effectively limits its applicability in further stages of the development process (i.e. programming), where more accurate (built on the implementation) estimates are available. An online measurement technique is presented in [17], which describes a method for online measurements of component-based applications. It builds on the Linux Trace Toolkit [18] to capture components performance. In particular, it measures remote invocation overhead (lookup and data marshaling) as well as inter-component communication delays. The measurements are used to predict behavior under different deployment strategies. The proposed solution, however, lacks run-time measurements of an already deployed application and is designed purely for production time analyses.

In terms of execution time measurements, we can distinguish between two common techniques. One is profiling, which periodically observes executed code and based on the acquired stack information creates a statistical execution profile of the observed system. The other technique is instrumentation, which uses probes, injected directly into the observed code, to measure its execution time. Profiling is not precise but has only small impact on the observed system. In contrast, instrumentation provides precise execution times but its high coverage may impose significant overhead. Profiling tools such as HPROF [19] or NetBeans Profiler [20] are in majority accommodating both of these techniques, however they lack the ability to scope the measurement to particular parts of the observed system. For more fine grained measurements (as we presume in our approach) an instrumentation tool is expected to support exact method (or block of code) measurement. Both Perf4J [21] and Xebia tools [22] use annotations to mark methods intended for execution time measurement. Compared to DiSL (which we employ), they lack the ability for more sophisticated measurement logic insertion.

Very often, the measurements may require certain conditions to hold or even perform more complex computation to decide whether to store or discard the measured values. To support such scenarios, the instrumentation framework needs to provide a possibility for an arbitrary instrumentation insertion into the observed code. One of such is the AspectJ framework [23]. It allows one to easily insert any block of code in the instrumented program in order to perform various dynamic analysis task. As it is not primarily designed for performance measurements, it imposes higher overheads than comparable tools. Another examples of instrumentation-based solutions are Sofya [24] and Chord [25], which support creation of custom analysis tools. In our approach, the DiSL framework was selected, as it provides both flexible enough and high-level language

for specifying custom instrumentation that is suitable in the context of component process performance analyses.With respect to the approaches for communication cost assessment, these usually build on simulation frameworks that benchmark developed applications against different deployment models. During the simulation process, various statistics are collected, the accuracy of which depends directly on the precision of the model being used. In our method we rely on the OMNet++ network simulator [26], which is a mature product with support for a multitude of network protocols (including wired and wireless IP-based networks, MANETs, etc.). OMNet++ further provides an API for statistical analyses, which makes possible collection of various performance indicators. Naturally, other network simulators exist (e.g. NS-2 and NS-3 [27]) and are used for the same purpose. While our approach incorporates the network simulator, it focuses mostly on the ACEs level of abstraction that allows for reasoning about communication performance at the application level (i.e. it answers questions about the perceived staleness of component knowledge, etc.).

6 Conclusion

In this paper, we have presented an approach for performance-awareness introduction in the development process of autonomic component ensembles. The approach is centered around EDLC, which we have extended by a set of well-defined activities for pinpointing the performance goals, measuring the corresponding performance indicators, and bringing the information about performance to the developer. This allows the developer to have an idea about the expected performance and related interactions already when developing them. Additionally, our approach includes integration with the runtime, which makes it possible to incorporate actual performance of ACEs in a deployment environment and reflect it back to the development. We have demonstrated the core ideas of our approach based on existing tools for monitoring and analyzing performance. To provide for a holistic solution, these tools have to be integrated within a development environment, whose sketch we have also provided. Such an integration and real-life evaluation constitute our future work.

Acknowledgments. This work was partially supported by the EU project ASCENS 257414 and by Charles University institutional funding SVV-2014-260100. The research leading to these results has received funding from the European Union Seventh Framework Programme FP7-PEOPLE-2010-ITN under grant agreement n°264840.

References

[1] T. Bures, R. De Nicola, I. Gerostathopoulos, N. Hoch, M. Kit, N. Koch, G. V. Monreale, U. Montanari, R. Pugliese, N. Serbedzija, M. Wirsing, and F. Zambonelli, “A Life Cycle for the Development of Autonomic Systems: The e-Mobility Showcase.” 2013.

[2] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil, “DEECo – an Ensemble-Based Component System,” in *Proc. of CBSE’13*, 2013, pp. 81–90.

[3] IBM, Ed., “An Architectural Blueprint for Autonomic Computing,” Jun. 2005.

[4] J. Keznikl, T. Bures, F. Plasil, I. Gerostathopoulos, P. Hnetynka, and N. Hoch, "Design of Ensemble-Based Component Systems by Invariant Refinement," in *Proc. of CBSE'13*, 2013, pp. 91–100.
[5] D. B. Abeywickrama, N. Bicocchi, and F. Zambonelli, "SOTA: Towards a General Model for Self-Adaptive Systems," in *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), IEEE 21st Int. Workshop*, 2012, pp. 48–53.
[6] "ARGoS," 2013. [Online]: http://iridia.ulb.ac.be/argos.
[7] "GIMPLE Model Checker," 2013. [Online]: http://d3s.mff.cuni.cz/~sery/gmc/.
[8] V. Horký, F. Haas, J. Kotrč, M. Lacina, and P. Tůma, "Performance Regression Unit Testing: A Case Study," in *Computer Performance Engineering SE - 12*, vol. 8168, M. Balsamo, W. Knottenbelt, and A. Marin, Eds. Springer Berlin Heidelberg, 2013, pp. 149–163.
[9] L. Bulej, T. Bureš, J. Keznikl, A. Koubková, A. Podzimek, and P. Tůma, "Capturing Performance Assumptions Using Stochastic Performance Logic," in *Proceedings of the 3rd ACM/SPEC Int. Conference on Performance Engineering*, 2012, pp. 311–322.
[10] "Stochastic Performance Logic (SPL)," 2014. [Online]: http://d3s.mff.cuni.cz/software/spl-java/.
[11] D. Gelernter, "Generative communication in Linda," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 1, pp. 80–112, Jan. 1985.
[12] R. Friedman, D. Gavidia, L. Rodrigues, A. C. Viana, and S. Voulgaris, "Gossiping on MANETs: the Beauty and the Beast," *ACM SIGOPS Oper. Syst. Rev.*, vol. 41, no. 5, pp. 67–74, Oct. 2007.
[13] "DiSL Framework," 2013. [Online]: http://disl.ow2.org/xwiki/bin/view/Main/.
[14] S. Shende, A. D. Malony, C. Rasmussen, and M. Sottile, "A performance interface for component-based applications," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. Int.*, 2003, p. 8.
[15] J. Davison De St. Germain, A. Morris, S. G. Parker, A. D. Malony, and S. Shende, "Performance Analysis Integration in the Uintah Software Development Cycle," *Int. J. Parallel Program.*, vol. 31, no. 1, pp. 35–53, Feb. 2003.
[16] S. Becker, H. Koziolek, and R. Reussner, "The Palladio Component Model for Model-driven Performance Prediction," *J. Syst. Softw.*, vol. 82, no. 1, pp. 3–22, Jan. 2009.
[17] C. Stewart and K. Shen, "Performance Modeling and System Management for Multi-component Online Services," in *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, 2005, pp. 71–84.
[18] K. Yaghmour and M. R. Dagenais, "Measuring and Characterizing System Behavior Using Kernel-level Event Logging," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2000, p. 2.
[19] "HPROF: A Heap/CPU Profiling Tool," 2014. [Online]: http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html.
[20] "NetBeans Profiler," 2014. [Online]: http://profiler.netbeans.org.
[21] "Perf4J," 2014. [Online]: http://perf4j.codehaus.org.
[22] "Xebia Tools," 2014. [Online]: http://code.google.com/p/xebia-france.
[23] G. Kiczales, "AspectJ(tm): Aspect-Oriented Programming in Java," in *Objects, Components, Architectures, Services, and Applications for a Networked World SE - 1*, vol. 2591, M. Aksit, M. Mezini, and R. Unland, Eds. Springer Berlin Heidelberg, 2003
[24] A. Kinneer, M. B. Dwyer, and G. Rothermel, "Sofya: Supporting Rapid Development of Dynamic Program Analyses for Java," in *Software Engineering - Companion, 2007. ICSE 2007 Companion. 29th Int. Conference on*, 2007, pp. 51–52.
[25] "Chord Group," 2014. [Online]: http://pag.gatech.edu/home.
[26] "OMNet++ Simulation Framework," 2013. [Online]: http://omnetpp.org.
[27] "Network Simulatior," 2014. [Online]: http://www.nsnam.org.