Software model refactoring based on performance analysis: better working on software or performance side?

Davide Arcelli, Vittorio Cortellessa
Dipartimento di Ingegneria e Scienze dell’Informazione, e Matematica
Università degli Studi dell’Aquila
{davide.arcelli, vittorio.cortellessa}@univaq.it

Several approaches have been introduced in the last few years to tackle the problem of interpreting model-based performance analysis results and translating them into architectural feedback. Typically the interpretation can take place by browsing either the software model or the performance model. In this paper, we compare two approaches that we have recently introduced for this goal: one based on the detection and solution of performance antipatterns, and another one based on bidirectional model transformations between software and performance models. We apply both approaches to the same example in order to illustrate the differences in the obtained performance results. Thereafter, we raise the level of abstraction and we discuss the pros and cons of working on the software side and on the performance side.

1 Introduction

Identifying and removing the causes of poor performance in software systems are complex problems due to a variety of factors to take into account. Similarly to other non-functional properties, performance is an emergent attribute of software, as it is the result of interactions among software components, underlying platforms, users and contexts [29]. The current approaches to these problems are mostly based on the skills and experience of software developers or, in the best cases, of performance analysts.

Quite sophisticated profiling tools have been introduced for run-time performance monitoring [23], but it is well-known that the costs of solving performance problems at runtime is orders of magnitude larger than the ones at early phases of the software lifecycle [15]. Therefore, instruments that help to identify and remove causes of software performance problems early in the lifecycle are very beneficial.

In Figure 1 a round-trip Software Performance Engineering (SPE) process is schematically represented. The forward path starts from a software model that is transformed into a performance model (e.g. [30]) that can be solved with common performance analysis techniques/tools to obtain performance indices [20, 3]. The backward path consists of a problems detection/solution step that processes the performance indices, in conjunction with the software artifact and/or the performance model, to detect and remove possible sources of performance problems. Hence, a set of refactoring actions that may apply to the software artifact and/or the performance model is obtained. This round-trip process is reiterated until satisfactory performance indices are obtained.

Two options have been represented in Figure 1 for what concerns the reiteration mechanism, and they are identified by non-continuous arrows. Dotted arrows represent the option of working on the performance model to detect and remove performance problems. In this case a transformation from the performance model to the software model has to take place when satisfactory performance indices are obtained. Dotted-line arrows represent the option of working on the software model where refactoring actions are applied. In both cases the forward path has to be run at each iteration to obtain the performance indices of a refactored (performance or software) model.
Software model refactoring based on performance analysis

In the last 5 years several approaches have appeared for identification and removal of performance problems either in the software model \cite{5,1,21} or in the performance model \cite{12,31}. Although these two categories of approaches nicely fit into the round-trip process of Figure 1, they work in different modeling environments, under different assumptions. Hence, not only they might achieve different results, but also they can show very different characteristics in terms of automation, scalability, effectiveness, etc.

The goal of this paper is to highlight the differences between these two categories of approaches in order to envisage the contexts where they can be more appropriately used. For this goal we consider two approaches that we have recently introduced.

The first approach is based on the detection and solution, on the software model, of performance antipatterns that are used for “codifying” the knowledge and experience of analysts by means of the identification of a problem, i.e. a bad practice that negatively affects software performance, and a solution, i.e. a set of refactoring actions that can be carried out to remove it \cite{1}. This approach involves dotted-line arrows of Figure 1.

The second approach is based on bidirectional model transformations between UML software models and Queueing Network (QN) performance models. A forward transformation is used to generate the performance model from an initial software model. The corresponding backward transformation is used to generate a new software model from a satisfactory performance model obtained by means of changes made by the analyst on the performance side \cite{12}. This approach involves dotted arrows of Figure 1.

We apply both the approaches to the same running example in the E-Commerce domain in order to illustrate the differences in the obtained results. Thereafter, we raise the level of abstraction and we discuss the issues that we have to cope with when working on the software side or the performance side to detect and remove performance problems.

The paper is organized as follows: in Section 2 a running example in the E-Commerce domain is presented and performance analysis results obtained by applying the two approaches are illustrated; Section 3 discusses the major issues related to working on the software or the performance side in a round-trip SPE process; Section 4 presents related works, and finally Section 5 concludes the paper.

2 Running Experiment

In this section we show an application of the two approaches of interest in the E-Commerce domain. We first describe the E-Commerce System software architectural model, then numerical results obtained
from the experimentation are presented and discussed.

2.1 The E-Commerce System (ECS)

2.1.1 The ECS model

ECS is a web-based system that manages business data. We assume to have a multi-view model, composed by (i) Static and (ii) Dynamic Views.

Several components are in the (server-side) Static View of Figure 2 each one providing/requiring interfaces and/or operations called during services execution. Among all provided services in this paper we focus on the three ones of Figure 3 namely: (a) Register related to customers registration, (b) BrowseCatalog for consulting the products catalog, and (c) MakePurchase that is executed whenever a customer wants to purchase a product.

Note that, since we adopted the SAPone methodology [9] for generating a QN performance model [8] from a (platform-independent) UML software model in both the compared approaches, no information about the deployment of software components is provided. Hence, we are assuming that every software component is placed on a logical device, and that all the devices have the same “speed” but they can manage requests through queues of different capacity and different scheduling policy. When the performance model is generated from the software model, that assumption allows us to directly map each software component to a single service center, and to connect it with the other ones in respect with the interface realizations/usages in the Static View and the message flows in the Dynamic View.

Figure 2: Initial ECS - Static View.

2.1.2 Performance requirements, legacy constraints and performance annotations for ECS

Several performance requirements have been defined on services response time and hardware devices utilization:

R1: no service must have a response time greater than 4 seconds in the server-side when in the whole system there are 150, 300, and 50 users requesting respectively the MakePurchase, BrowseCatalog, and Register services;

R2: no hardware resource has to be used more than 90% when in the whole system there are 150, 300, and 50 users requesting respectively the MakePurchase, BrowseCatalog, and Register services.
Furthermore, a legacy constraint has been introduced:

**C1:** since we assume that the Database is a Commercial Off-The-Shelf component, it can neither be refactored in any way nor replaced by another database having better performance.

In order to carry out performance analysis, several input parameters for the performance model obtained from the ECS model by means of the forward transformation have to be defined:

**Workload characterization.** Performance requirements define the average number of users for workloads. Hence, we define a closed workload class for each considered service. In particular, the
average number of users for MakePurchase, BrowseCatalog, and Register are, respectively, 150, 300, and 50. This means that under an aggregate average number of users equal to 500, the 10% is registering, the 60% is browsing the products catalog, and the 30% is purchasing a product.

**Service demands definition.** Figure 4 shows service demands of the considered services. Values are expressed in seconds. Since Customer generates workloads, we consider it as the delay node whose think time is 15 seconds in order to represent the mean time needed by a user for elaborating a request and the arrival of the latter to the server-side. All the other service demands are orders of magnitude lower than think time because there is no time spent in thinking. Note that service demands are well proportioned relating to requests in the various services. Hence, we are confident about their adequacy.

![Figure 4: Service demands for the initial ECS.](source)

2.1.3 **Performance analysis for ECS**

All the performance analysis have been conducted by transforming each involved software architectural model into a QN performance model [8], and by solving the latter with the JMT tool [3].

Figure 5 shows performance analysis results for the QN corresponding to the initial ECS model for the indices of interest, i.e. response times and utilizations.

![Figure 5: Response times and utilizations of the initial ECS.](source)

As illustrated in Figure 5.(a), $R_1$ is widely violated by the MakePurchase service. In fact, under a workload of 150 users purchasing a product, in the server-side the mean time elapsed from a single request arrival to its departure (i.e. the response time) is $23.32 - 15.00 = 8.32$ seconds, i.e. more than double compared to the defined threshold of 4 seconds. Instead, the response times at server-side for the other two services don’t violate $R_1$. In fact, BrowseCatalog has a mean response time of $17.81 - 15.00 = 2.81$ seconds whereas Register has a mean response time of $16.77 - 15.00 = 1.77$ seconds.

As illustrated in Figure 5.(b), $R_2$ is violated by Database and ProductCatalog. In fact, under the specified workload, the former has an utilization of 96.66% whereas the latter has an utilization of 99.99%.

Since several performance indices of interest are not satisfactory we need to refactor the software model in order to satisfy violated requirements.

2.2 **ECS refactoring based on performance antipatterns**

In this section we perform two refactorings on the initial ECS model by using the approach based on performance antipatterns [11]. We first execute the antipatterns detection phase in order to identify bad
practices related to performance, then we randomly choose an antipattern to remove. Hence, given the corresponding pair (problem, solution), we execute a refactoring that applies the solution to the problem.

Let us assume that, among all the detected antipatters, we choose to remove the occurrence of the BLOB antipattern where the so called “Blob” entity is ProductCatalog. As stated in [25], this antipattern “occurs when a single class either 1) performs all of the work of an application or 2) holds all of the application’s data” and it can be solved “by refactoring the design to distribute intelligence uniformly over the application’s top-level classes, and to keep related data and behavior together”. Hence, let us assume that products managed by ECS are films and books and that these types of products are requested respectively for 80% and 20%. We can split the ProductCatalog in two components, i.e. FilmCatalog and BookCatalog. Figure 6 shows the refactored ECS static view, where the two components have been introduced and adequately connected to the other ones. Note that their interfaces/operations are conceptually coherent with data each one of them manages. Also dynamic and deployment views are affected by the refactoring. In particular, in the dynamic view a probability-weighted alternative fragment with two operands related to films (with a probability of 0.8) and books (with a probability of 0.2) replaces each portion of MakePurchase and BrowseCatalog services involving ProductCatalog, replicating that portion for both FilmCatalog and BookCatalog in an adequate manner.

Figure 6: Refactored ECS (BLOB removed) - Static View.

Figure 7 shows performance analysis results of the QN corresponding to the ECS model for the refactoring described in this section.

<table>
<thead>
<tr>
<th></th>
<th>Aggregate</th>
<th>MakePurchase</th>
<th>BrowseCatalog</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customers</td>
<td>15,000.00</td>
<td>15.000.00</td>
<td>15.000.00</td>
<td>15.000.00</td>
</tr>
<tr>
<td>WebServer</td>
<td>0.026468</td>
<td>0.026005</td>
<td>0.026057</td>
<td>0.026057</td>
</tr>
<tr>
<td>Dispatcher</td>
<td>0.046066</td>
<td>0.046066</td>
<td>0.046066</td>
<td>0.046066</td>
</tr>
<tr>
<td>UserCont.</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>Catalog</td>
<td>0.011036</td>
<td>0.011036</td>
<td>0.011036</td>
<td>0.011036</td>
</tr>
<tr>
<td>BookCont.</td>
<td>0.019824</td>
<td>0.019824</td>
<td>0.019824</td>
<td>0.019824</td>
</tr>
<tr>
<td>Database</td>
<td>1.962180</td>
<td>1.962180</td>
<td>1.962180</td>
<td>1.962180</td>
</tr>
<tr>
<td>FilmCont.</td>
<td>0.481706</td>
<td>0.481706</td>
<td>0.481706</td>
<td>0.481706</td>
</tr>
</tbody>
</table>

(a) Response times

Figure 7: Response times and utilizations after the BLOB antipattern removal.

As illustrated in Figure 7(a), R1 is no longer violated for the MakePurchase service. In fact, under a workload of 150 users purchasing a product, the response time at the server-side is 18.03 - 15.00 = 3.03 seconds, that are almost one second lower than the defined threshold of 4 seconds. Also the response time at server-side for BrowseCatalog improves, hence it still not violate R1. Instead, the response times at server-side for Register significantly increases, and it becomes 20.05 - 15.00 = 5.05 seconds, i.e. greater

Readers interested to all details related to the refactored views not shown in this paper can refer to the external resource available at [http://www.di.univaq.it/cortelle/docs/FESCA2013_appendix.pdf](http://www.di.univaq.it/cortelle/docs/FESCA2013_appendix.pdf)
than the defined threshold of 4 seconds.

As illustrated in Figure 7(b), $R_2$ is violated for Database and FilmCatalog. In fact, under the specified workload, the former has an utilization of 99.9% whereas the latter has an utilization of 95%.

Since several performance indices of interest are not satisfactory we need further refactoring in order to satisfy violated requirements.

Let us assume that, among all the detected antipatterns, we choose to remove the occurrence of the EST antipattern in Register, in order to obtain a better response time for that service. As stated in [25], this antipattern "occurs when an excessive number of requests is required to perform a task. It may be due to inefficient use of available bandwidth, an inefficient interface, or both" and it can be solved by means of the application of the Session Facade design pattern [27]. Hence, two new communicating components, i.e. RemoteFacade and LocalFacade, are introduced between the one originating an excessive number of requests, i.e. UserController, and the destination of those requests, i.e. Database. This refactoring results in more efficient interfaces and also in a more efficient use of available bandwidth.

Figure 8 shows the refactored Register service of ECS, where the two Facade components have been adequately introduced in the dynamics of that service. Also static and deployment views are affected by the refactoring. In particular, also in the static view the two Facade components and their interfaces have been adequately introduced.

Figure 9 shows performance analysis results of the QN corresponding to the ECS model for the refactoring described in this section.

As illustrated in Figure 9(a), $R_1$ is no longer violated for the Register service. In fact, under a workload of 50 users requesting a registration, the response time at the server-side is $18.16 - 15.00 = 3.16$ seconds, that are almost one second lower than the defined threshold of 4 seconds. Also response times at server-side for BrowseCatalog and MakePurchase improve, hence they still not violate $R_1$. In fact, the response time at server-side for the former becomes $16.77 - 15.00 = 1.77$ seconds, whereas the one for the latter becomes $17.83 - 15.00 = 2.83$ seconds.

As illustrated in Figure 9(b), $R_2$ is violated for Database and FilmCatalog. In fact, under the specified workload, the former has an utilization of 99.57% whereas the latter has an utilization of 96.37%.

\footnote{Note that the EST occurrence detected in this second step could also have been detected in the previous iteration where we opted for the BLOB occurrence removal.}
Finally, utilizations of RemoteFacade and LocalFacade are, respectively, 6.89% and 11.01%, hence they do not violate R2.

Since several performance indices of interest are not satisfactory we need further refactoring in order to satisfy violated requirements but, at this point, we stop iterating the antipattern-based refactoring approach and turn back to the initial ECS model in order to start the application of the refactoring approach that uses bidirectional transformations.

### 2.3 ECS refactoring based on bidirectional transformations

In this section we perform two refactorings on the initial ECS model by using the approach based on bidirectional transformations. Hence, refactorings are executed by the performance analyst directly on the performance model, basing on his own expertise on interpreting performance results.

Starting from the QN model in the top-side of Figure 10 resulting from the application of the forward transformation to the initial ECS model, performance results of Figure 5 are obtained. As we stated in Section 2.2, we assume that products managed by ECS are films and books and that these types of products are requested respectively for 80% and 20%. Since the Database cannot be refactored in any way, we assume that the performance analyst refactors the QN model as shown in dashed boxes of Figure 10, hence by splitting ProductCatalog (i.e. the bottleneck) in two service centers, i.e. FilmCatalog and BookCatalog, for managing the two types of products. Service demands for new service centers are adjusted basing on percentiles assumed above.

Note that, by applying the backward transformation in order to go back to the software model, the same refactored ECS resulting from the first refactoring of Section 2.2 is obtained. Therefore, also the performance results are the same, hence several performance indices of interest are not satisfactory and we need further refactoring at performance model side in order to satisfy violated requirements.
Let us assume that, since the Database cannot be refactored in any way, the performance analyst refactors the QN model as shown in dashed boxes of Figure 11(hence by splitting FilmCatalog (i.e. the bottleneck) in two service centers, i.e. FilmCatalog1 and FilmCatalog2, and balancing their load. This means that requests for films are equally distributed between the two service centers, i.e. CatalogController routes the requests to the two service centers with an identical probability of 0.5. Service demands for new service centers are adjusted basing on that probability.

Figure 12 shows an excerpt of the static view for the refactored ECS obtained applying the backward transformation in order to go back to the software model. Also dynamic and deployment views are affected by the refactoring. In particular, in the dynamic view a probability-weighted alternative fragment with two operands related to request balancing (probability of 0.5) replaces the portion of MakePurchase and BrowseCatalog services involving FilmCatalog, by replicating that portion for both FilmCatalog1 and FilmCatalog2 in an adequate manner.

Figure 13 shows performance analysis results of the QN corresponding to the ECS model for the refactoring described in this section.

As illustrated in Figure 13(a), R1 is still not violated for MakePurchase and BrowseCatalog services. In fact, under a workload of 150 users purchasing a product, the response time at the server-side for the former is 17.39 - 15.00 = 2.39 seconds, resulting in an improvement greater than half a second whereas, under a workload of 300 users browsing the catalog, the response time at the server-side for the latter remains more or less unchanged, i.e. 17.14 - 15.00 = 2.14 seconds (improvement of 0.25 seconds). Instead, the response time at server-side for Register increases, and it becomes 20.85 - 15.00 = 5.85 seconds, i.e. still greater than the defined threshold of 4 seconds.

As illustrated in Figure 13(b), the only component having an utilization that violates R2 is Database.
Software model refactoring based on performance analysis

Figure 13: Response times and utilizations after the FilmCatalog split.

In fact, under the specified workload, it has an utilization of 99.9%.

Since several performance indices of interest are not satisfactory we need further refactoring in order to satisfy violated requirements but, at this point, we stop iterating the refactoring approach that uses bidirectional transformations in order to compare both the approaches on the same number of iterations.

3 Raising the abstraction level

In this section, in light of the example illustrated in Section 2, we discuss the issues that characterize refactoring approaches that work on the software side and the ones working on the performance side.

3.1 Level of automation and human role

In order to be adopted in practice, a round-trip SPE process needs to be supported by a high level of automation, due to the complexity of tasks that have to be executed and the decisions that have to be taken. Nevertheless, the human role in SPE should not be completely removed, because the experience and skills of software designers and/or performance analysts cannot be fully embedded in automated processes, as we discuss in the following.

Let us look at the steps in the forward path of Figure 1: software to performance model transformations have been fully automated in the last decade even for high complexity cases [6]; performance model solution is a well assessed task since decades and very sophisticated techniques are available today [3]. So, let us focus on the backward path of Figure 1. Automation in the problem detection and solution step is traditionally very well supported on the performance side, where a whole theory on bottleneck identification and removal has been introduced few decades ago (e.g. [20]) and has been continuously refined by more recent results (e.g. [14]). However, refactoring actions applied on the performance side have to be reported on the software side. The automation of the former task rests on bidirectional model transformations, as shown in Section 2.3. Bidirectional transformations are complex to build in this domain, mostly due to the low injectivity of forward transformations [24] that usually collapse several elements of a software model into an unique element of a performance model, making back-tracing more difficult. Instead, on the software side a certain level of automation has been introduced only recently, for example based on antipatterns (as illustrated in our example) or on metaheuristics that search the solution space looking for changes that can improve the performance indices [21].

However, in both cases it may be necessary to decide among alternative refactoring actions, because it is difficult to implement such a sharp detection and solution step that terminates with an unique suggestion of refactoring actions, especially in large scale systems.

The number of alternative refactoring actions coming from the detection and solution phase on the software side can be considerably higher than the one on the performance side. This is mostly due...
from the richness of software model notations that makes the space of possible solutions quite larger than
the one on the performance side. For example, the reduction of number of visits to a network node in
a QN performance model consists in modifying an integer parameter, whereas in UML may correspond
to several alternatives that reduce the number of messages exchanged in a behavioral model (i.e. the
EST removal in Section 2.2). This aspect, on one end, can be considered as an advantage of approaches
that work on the software side, because they propose a variety of choices to software designers that can
select the most appropriate one(s), for example considering cost factors or legacy constraints. On the
other end, a too large variety of solutions (i.e. the list of detected antipatterns) can be hard to manage.
Decision support mechanisms, for example based on convenience metrics, might help the designer to
mitigate this aspect [7, 21]. However, they are heuristic techniques that sometimes do not work better
than the designer’s experience.

3.2 Effectiveness of refactoring actions

Independently of the considered side, after a set of alternative refactoring actions will be determined,
either the software designer or the performance analyst has to decide which one(s) applying to the (soft-
ware or performance) model. This is a key decision for sake of process convergence. In fact, a decision
tree drives the process, where each branch (i.e. each set of actions applied) leads to a new model that has
to be solved in order to check whether performance problems have been removed (e.g. a performance
requirement, that was violated, is now satisfied).

As mentioned above, heuristic techniques (possibly based on convenience metrics) might support
this decision both on the software and the performance side. However, the effectiveness of refactoring
actions will be given by the tradeoff between the refactoring complexity (i.e. the distance between the
original model and the refactored one) and the performance gain obtained from the refactoring. For
example, it may happen that heavy re-design actions, like splitting software components and modifying
their interaction patterns (i.e. FilmCatalog and BookCatalog that replace ProductCatalog in Section 2.2),
bring little performance benefits as compared to re-deploy a software component (ProductCatalog) on
another processing node (i.e. DatabaseNode of ECS). In most cases it is difficult to predict the perfor-
mance gain of a refactoring action without actually solving a refactored performance model. However,
again due to the usual richness of software modeling notations, the refactoring complexity is generally
lower on the performance side. At least, the portfolio of refactoring actions that can be applied, for ex-
ample on a Queueing Network, is certainly more limited than the one working on an UML model. On
one end, this implies that in order to solve nested performance problems more iterations may be needed
on the performance side compared to the software side. On the other end, the performance gain of refac-
toring actions applied to the software side is more unpredictable due to the forward transformation that
generates a refactored performance model from a refactored software model. In worst cases, software
refactoring actions could lead to performance degradation due to side effects that are not visible in the
software model (e.g. false positive performance antipatterns [4].

3.3 Scalability

Several factors contribute to the scalability of refactoring approaches. First, it comes straightforward
that a single loop of the round-trip process illustrated in Figure 1 is shorter in case of refactoring on the

3This problem is analogous to looking for bugs in software code: working on high-level languages (like C) gives a quite
rich syntax, but modifications must be translated in assembler before looking at their effects, whereas working in assembler
provides a more direct control on the effects of changes.
Software model refactoring based on performance analysis

performance side than on the software side. This is due to the need of applying a forward transformation to a refactored software model for obtaining a refactored performance model. Direct refactoring on the performance side, however, requires at the end of the process (i.e. when a satisfactory performance model has been obtained) the application of a backward transformation to build back a corresponding satisfactory software model. This observation leads to the scalability of the transformations.

Let us assume that the forward transformation has a $O(f_{\text{for}})$ complexity, whereas the complexity of the forth and back transformations are, respectively, $O_{\text{bid}}(f_{\text{for}})$ and $O_{\text{bid}}(b_{\text{ack}})$. By a rough counting, if N iterations are necessary to solve problems on the performance side and M iterations (usually with $M < N$) are necessary on the software side, then the scalability tradeoff can be expressed by:

$$M \times O(f_{\text{for}}) < N \times (O_{\text{bid}}(f_{\text{for}}) + O_{\text{bid}}(b_{\text{ack}}))$$

Of course, the transformation complexities depend on the distance between the specific source software metamodel and target performance metamodel.

Another factor that affects the scalability is the number of iterations necessary to obtain satisfying results. This translates to the depth of the decision tree that we have mentioned above. As discussed in the previous subsection, this obviously depends on the effectiveness of refactoring actions, but also on their dependencies. In fact, alternative refactoring actions that can be independently applied (i.e. the refactored model obtained by applying all of them does not depend on the application order, as the sequential removal of BLOB and EST antipatterns of Section 2.2), ease the tree navigation. As opposite, alternative actions that affect each other, as splitting a component introduced in a previous refactoring (i.e. the FilmCatalog splitting of Section 2.3), requires a (possibly expensive) backtrack process on the decision tree to be considered. Heuristic approaches to prune the decision tree would be suitable in this context. However, it looks easier (but likely less effective) to decide on the performance side than on the software side. For example, it is usually more evident the bottleneck to be first removed than the antipattern to be solved (among alternative ones).

4 Related work

At best of our knowledge this is the first paper that compares approaches for model refactoring based on performance analysis. Hence, we present a brief overview of works related to the approaches we compared, i.e. [1] and [12].

**Working on the software model side.** Very few model-based approaches for automated performance diagnosis and improvement have been introduced up today in the software modeling domain.

In [21] and [18] meta-heuristic search techniques are used for improving different non-functional properties of component based software systems by means of evolutionary algorithms. The main limitation of such approaches is that it is quite time-consuming because the design space may be huge.

In general, there has been a significant effort in software refactoring based on design patterns [13]. However, differently from patterns, antipatterns look at the negative features of a software system and describe commonly occurring solutions to problems that generate negative consequences [2] [19].

*Technology-independent* performance antipatterns have been defined in [25] and they represent the main references in our AP-based works [1] [5]. *Technology-specific* antipatterns have been specified in [11] [28] and, more recently, in [22], where EJB antipatterns have been represented as a set of rules loaded into a detection engine and they are detected by a monitoring mechanism that leads to get run-time system properties which are matched with pre-defined rules.

Note that the complexity of the forward transformation in the context of a bidirectional one is higher than a simple forward transformation, because tracing information has to be brought to obtain the backward transformation [12].
Working on the performance model side. The formal definition of a round-trip engineering process considering the non-totality and non-injectivity of model transformations is presented in [16]. Valid modifications on target models are limited to the ones which do not induce backward mappings out the source metamodel and are not operated outside the transformation domain.

In [31] performance problems are identified through the detection of bottlenecks and long paths on Layered Queueing Networks (LQN) models. Contrary to our approach based on bidirectional transformations [12], however, in [31] no clue is given on the back propagation of performance model changes to any software model notation.

Stevens [26] discusses bidirectional transformations focusing on basic properties which they should satisfy and pointing out some ambiguity about specification of non-bijective transformations.

Bidirectional transformations based on triple graph grammars (TGGs) are presented in [17], where models are interpreted as graphs and transformations are executed by using graph rewriting techniques.

Recently some interesting solutions based on lenses have been proposed. In [10] the authors illustrate a technique to support bidirectional transformations relying no more on mapping between models but on differences operable on them.

5 Conclusions

In this paper we have compared two performance-based model refactoring approaches at work on the same example. This allowed us to highlight the peculiar aspects of working on either the software side or the performance side. We have finally summarized our findings in Table 1.

<table>
<thead>
<tr>
<th>Classification parameters</th>
<th>AP-based approach (software side)</th>
<th>BT-based approach (performance side)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human skills required</td>
<td>Design skills</td>
<td>medium</td>
</tr>
<tr>
<td></td>
<td>Performance skills</td>
<td>low</td>
</tr>
<tr>
<td>Degree of automation</td>
<td>Problem detection</td>
<td>medium</td>
</tr>
<tr>
<td></td>
<td>Problem solution</td>
<td>high</td>
</tr>
<tr>
<td>Refactoring metrics</td>
<td>Number of actions</td>
<td>high</td>
</tr>
<tr>
<td></td>
<td>Complexity</td>
<td>low, medium, high</td>
</tr>
<tr>
<td></td>
<td>Performance gain predictability</td>
<td>low, medium</td>
</tr>
<tr>
<td>Scalability</td>
<td>Number of iterations</td>
<td>low, medium</td>
</tr>
<tr>
<td></td>
<td>Single iteration complexity</td>
<td>$O_{(forward)} + O_{bid(back)}$</td>
</tr>
</tbody>
</table>

Table 1: Overview of the compared approaches.

With the increasing interest on this type of refactoring approaches we retain very relevant to study contexts where different techniques can better work than other ones. This is an initial step for the comparison of such approaches. The afterthoughts of this experience will be either consolidated or turned down by applying the approaches to a significant amount of examples, that is our intent in the near future.

Several future directions will be investigated: (i) it could be very interesting to study a mixed approach that combines the ones we compared in this paper, i.e. by executing sequences of bottleneck analysis followed by detection and solution of performance antipatterns, and/or viceversa; (ii) the introduction of performance antipatterns at the performance model side could support the performance analyst in detection and solution of performance problems, although it should be considered that the number of alternative actions resulting from the detection and resolution phases on a performance model can be considerably fewer than the ones on the software side, due to the lower number of elements in performance models as compared to software models; (iii) finally, we are working on the introduction of measurement-based performance problem detection and solution at the code level by means of monitoring-driven testing techniques for cloud applications [4].
6 Acknowledgments

This work has been supported by the European Office of Aerospace Research and Development (EOARD), Grant/Cooperative Agreement Award no. FA8655-11-1-3055 on “Consistent evolution of software artifacts and non-functional models”.

7 Bibliography


[28] Bruce Tate, Mike Clark, Bob Lee & Patrick Linskey (2003): Bitter EJB.

