A Model-driven Approach to Catch Performance Antipatterns in ADL Specifications

Martina De Sanctis[a], Catia Trubiani[b], Vittorio Cortellessa[c], Antinisca Di Marco[c], Mirko Flamminj[c]

[a]*Fondazione Bruno Kessler, Trento, Italy*
[b]*Gran Sasso Science Institute, L'Aquila, Italy*
[c]*University of L'Aquila, L'Aquila, Italy*

Abstract

Context: While the performance analysis of a software architecture is a quite well-assessed task nowadays, the issue of interpreting the performance results for providing feedback to software architects is still very critical. Performance antipatterns represent effective instruments to tackle this issue, because they document common mistakes leading to performance problems as well as their solutions.

Objective: Up today performance antipatterns have been only studied in the context of software modeling languages like UML, whereas in this manuscript our objective is to catch them in the context of ADL-based software architectures to investigate their effectiveness.

Method: We have implemented a model-driven approach that allows the automatic detection of four performance antipatterns in Æmilia, that is a stochastic process algebraic ADL for performance-aware component-oriented modeling of software systems.

Results: We evaluate the approach by applying it to three case studies in different application domains. Experimental results demonstrate the effectiveness of our approach to support the performance improvement of ADL-based software architectures.

Conclusion: We can conclude that the detection of performance antipat-

Email addresses: `msanctis@fbk.eu` (Martina De Sanctis), `catia.trubiani@gssi.infn.it` (Catia Trubiani), `vittorio.cortellessa@univaq.it` (Vittorio Cortellessa), `antinisca.dimarco@univaq.it` (Antinisca Di Marco), `mirko.flamminj@univaq.it` (Mirko Flamminj)

terns, from the earliest stages of software development, represents an effective instrument to tackle the issue of identifying flaws and improving system performance.

Keywords: Architecture Description Languages, Performance Antipatterns, Model-Driven Engineering, Æmilia ADL, Software Performance Analysis

1. Introduction

The need of early non-functional analysis of software architecture is nowadays well-assessed, as it generates positive effects on the whole software development process [1]. In fact, early detection of violations of non-functional requirements allows developers to save a lot of effort in the testing phases where bugs are hard and expensive to fix [2]. In fact, the investigation of non-functional attributes in software architectures helps to compare different alternatives that are equivalent from a functional viewpoint, thus introducing an additional value in the decisional process of software architects [3, 4, 5].

In the last two decades the performance analysis of a software architecture has become a well-assessed task. Several modeling languages (like UML) allow models to be annotated with performance input parameters, then annotated models can be transformed into performance models (like Queueing Networks), and analysis tools can be used to obtain performance indices. Syntax and semantics of several Architecture Description Languages (such as Æmilia [6]) allow performance parameters of a software architecture to be specified, and their supporting tools can analyze the architecture performance (besides the typical functional analysis) without needing to transform the architecture description into a performance-specific language.

In the literature, several approaches have been proposed for the performance analysis at the software architectural design level [7, 8, 9], whereas the issue of interpreting the performance results for providing architectural refactorings to software architects is still very critical. This is mostly due to the gap between performance results – i.e., mean values, variances, and-or probability distributions of indices like throughput, response time, etc. – and expected refactorings, like architectural alternatives, that can help to remove possible problems identified during the performance analysis phase. In most cases, software analysts (with no expertise in performance) build different architectural alternatives to try overcoming performance problems or, in the best case, performance experts provide suggestions based on their previous experience.

Therefore, further approaches are necessary to support and facilitate the process of performance results interpretation and software refactoring generation. Moreover, automation in this process would be breakthrough for this task. Up to now only *bottleneck analysis* [10] has been used for this goal. It allows the identification of cases where the performance of a software system is limited by a number of overloaded software components and–or hardware resources. However, it falls short to identify more complex cases.

Performance antipatterns [11, 12, 13] represent effective instruments to tackle the issue of interpreting performance results, because they document: (i) common mistakes leading to performance problems and (ii) solutions in terms of software refactorings. Their effectiveness has been demonstrated, among others, by our recently consolidated results: (i) we formalized the representation of antipatterns by means of first-order logic rules that express a set of system properties under which an antipattern occurs [14]; (ii) we introduced a methodology to prioritize the detected antipatterns and solve the most promising ones [15].

In the literature performance antipatterns have been only studied in the context of UML-like software modeling languages [16, 17], whereas in this paper we tackle the problem of identifying their occurrences in ADL-based software architectures. The main goal is to close a round-trip process that allows the performance analysis of an architecture and the results interpretation and architectural refactoring. This would contribute to an easier adoption of analysis practices in the daily life of software architects. Performance antipatterns are defined in terms of their own specific vocabulary [12], and they are founded on different aspects of a software system referring to *static*, *behavioral*, and *deployment* characteristics [14] as well as on *performance measures*. As a consequence, the ADLs suitable for the detection of performance antipatterns are those that better fit these characteristics, namely the ones that better overlap with the antipatterns vocabulary and that allow performance analysis of a software architecture.

Various ADLs allow performance analysis (e.g., ABACUS [18], OSAN [19], EAST-ADL [20]). Nevertheless, none of their supporting environments allows the interpretation of performance results. Based on our experience and supported by the software performance engineering community [11, 12, 13], the most promising ADLs for performance antipatterns detection are Æmilia [6] and AADL [21].

Æmilia is an ADL aimed (among other) at the performance evaluation of software systems. It allows a software architecture to be specified from a functional viewpoint as well as performance parameters (such as rates and probabilities of actions) and to evaluate performance indices of interest (such as throughput, response time and utilization) to be defined. Æmilia is very powerful in the specification of performance measures because it relies on rewards that can be assigned to architectural elements, and performance analysis co-exists with functional verification, such as the reachability of a certain state or deadlock-freeness. AADL is an ADL designed (among other things) for the specification and analysis of software systems. It supports, in fact, both performance analysis and functional simulation. AADL allows latency analysis on flow specifications of components and connections to be performed [22]. However, such analysis is limited to worst and best cases, whereas the reward-based mechanism of Æmilia enables a much wider range of performance specifications. On the basis of these considerations, we have adopted the Æmilia language as the ADL for this work because of its stronger ability in the specification of performance measures, while we defer AADL for our future work.

This paper is an extension of [23] where we have shortly illustrated an approach for performance antipatterns detection on Æmilia specification, which enables the actual usage of first-oder logic rules [14] in a concrete ADL-based software architecture. In detail, our approach starts by converting an Æmilia textual description into an Æmilia model conforming to an enriched Æmilia metamodel that we define in the following. The Æmilia model is subsequently annotated with performance results provided by the performance evaluation executed by means of the tool TwoTowers [24], which is the Æmilia supporting environment. At this point, the performance antipatterns detection is performed by our engine that analyzes the annotated Æmilia model using a set of OCL rules that model the detectable performance antipatterns. A list of detected antipattern occurrences, if any, is given as a result. Each antipattern includes in its definition the corresponding solutions that are alternative architectures allowing their removal. Æmilia allows to represent a subset of performance antipatterns, in particular its expressiveness enables the (full or partial) detection of seven performance antipatterns, as it will be illustrated later.

The choice of the OCL language for the definition of rules modeling the performance antipatterns is founded on the expressivity of OCL as compared to first-order logic with which performance antipatterns have been formally represented without the possibility of automating their detection [14]. From

[25], it follows that the expressive power of OCL is no lower than first-order logic, we do not loose expressivity by defining OCL rules for performance antipatterns. In addition, OCL is well suited for implementation purposes, being applicable, for example, to Eclipse Modelling Framework-based models, as it has been done in this work.

The novel contributions of this paper are: (i) a detailed description of an Æmilia metamodel enriched with performance antipatterns-related concepts; (ii) the automation of the performance results annotation that allows the inclusion of performance values in Æmilia models; (iii) an extended experimentation that includes further details on the original case study and two additional ones; (iv) a comparison of the antipatterns-based process with the bottleneck analysis.

As future work, we plan to apply our approach on other formalisms, starting from AADL that appears as the most promising one among the ADLs. Moreover, we intend to introduce automation in the application of antipatterns solutions. Lastly, we plan to experiment our approach on real systems coming from industrial experiences.

The paper is organized as follows: related work is discussed in Section 2. Section 3 provides background information on Æmilia and presents a preliminary study that has brought us to identify which performance antipatterns are detectable in Æmilia descriptions. Section 4 describes the whole approach of representation and detection of performance antipatterns in Æmilia. Validation results are reported in Section 5, where our approach is applied to three case studies. Section 6 discusses the open issues raised by the proposed approach and Section 7 argues the threats to validity. Finally, Section 8 concludes the paper with final considerations and directions for future work.

2. Related work

The broader scope of this paper is the analysis of non-functional concerns at the architecture level. In the literature, the analysis of quality properties at the architectural level has been discussed from different perspectives, including modeling strategies (e.g., architecture viewpoints and perspectives [5, 26]) that enable analysis methods [27, 28, 29]. There are two main streams of approaches in this direction: (i) qualitative (e.g., scenario-based architecture analysis [30], architecture trade-off analysis [31]) and (ii) quantitative (e.g., costs and benefits of architectural decisions [32], metric-based architecture optimization [33]). In the following, we focus on quantitative performance analysis of software architecture because we are interested in collecting a set of figures, such as resource utilization, service throughput and response time, thus to estimate the system quality with the support of quantitative results.

In the literature there are some ADLs that allow the specification of performance parameters for the goal of carrying out performance indices (ABACUS [18], OSAN [19], EAST-ADL [20]). However, none of their supporting environments allows the interpretation of such indices, nor do they make direct use of them. One exception is represented by AADL [21], which supports the performance verification [34]; five real-time design patterns (i.e., Synchronous data-flow, Ravenscar, Blackboard, Queued buffer and Unplugged) have been recently applied to enable schedulability analysis on AADL models [35]. However, no feedback is generated to improve the performance indices.

Differently from patterns, antipatterns look at the negative features of a software system and describe solutions to commonly occuring problems that generate negative consequences [36, 37].

Performance antipatterns deal with the performance issues of software systems. In the literature, the main source of performance antipatterns is the work done across years by Smith and Williams [11, 12, 13] that have ultimately defined 14 notation- and technology- independent antipatterns. The benefit of using these antipatterns is that they offer solutions in forms of alternative architectures to performance flaws and several works recently have demonstrated their usefulness [38, 39, 40]. Other sources of antipatterns focus on *technology- specific* problems, e.g., in [41, 42] antipatterns are expressed in terms of Java's trouble spots for programming, such as too much data in JSP sessions, problematic stateful session beans, etc.

Enterprise technologies and EJB antipatterns are analyzed in [43]: antipatterns are represented as a set of rules loaded into a detection engine. The matching between pre-defined rules and application properties is performed to detect EJB (i.e., technology specific) antipatterns. However, these antipatterns operate at the level of code, whereas our approach intends to work at the architectural level.

In [44], performance problems are identified before the implementation of the software systems, but they only refer to bottlenecks and long paths. The analysis is conducted on Layered Queueing Network (LQN) performance models. The main limitation of such approach is that it only applies to LQN models, hence its portability to other notations is yet to be proven and it may be quite complex.

In [45], meta-heuristic search techniques are used for improving different non functional properties of component based software systems: evolutionary algorithms search the architectural design space for optimal trade-offs. The main limitation of such approach is that it is quite time-consuming because the design space may be huge.

In [46], a performance antipatterns detection approach, called *Performance Problem Diagnostics* (PPD), is presented. The goal of this approach is to find the root causes of the identified performance problems by means of a decision tree derived from a *Performance Problem Hierarchy.* To detect specific antipatterns, detection strategies encapsulating heuristics have been defined. This approach can only capture implementation details as sources of performance problems, hence it is applied late in the software development process. In contrast, our approach can detect antipatterns occurrences early during the design phase.

In [47], the authors present DECOR (*DEtection and CORrection*), a method characterizing the required steps (i.e., *description analysis, specification, processing, detection and validation*) for the specification and detection of both code and design smells. Moreover, as an instantiation of the DECOR method, the authors present a detection technique called DETEX (*DETection EXpert*). Briefly, DETEX starts from a taxonomy and classification of smells from a domain analysis phase. Smells are then specified using a rule-based and declarative DSL; a set of rules (i.e., rule card) describe the properties that a class must have to be considered a smell. Subsequently, detection algorithms are automatically generated from the models of each smell and then applied on systems models. In this approach, 29 smells are studied. However, the focus is more on code smells (21 types), which are more related to the implementation, than on design antipatterns (8 types), which could be detected early at design phase. In contrast, our approach is focused on design antipatterns leading to performance flaws.

Recently, the need of automatic antipatterns detection approaches has emerged also in the Service Oriented Computing (SOC) field. In [48], the authors describe the impact of REST antipatterns on the increasingly used RESTful services. Because RESTful systems evolve continuously, to adapt to the dynamic context in which they operate, these evolutions may cause the introduction of REST antipatterns, that are added to those that can be introduced at design phase. A heuristics-based approach called SODA-R (Service Oriented Detection for Antipatterns in REST) for detecting antipatterns in RESTful systems is presented. By analysing the antipatterns descriptions, relevant static and dynamic properties are extracted and then combined for the definition of detection heuristics for each antipattern. Heuristics can be applied as detection algorithms on both the requests from clients and the responses from servers on different REST APIs. Furthermore, in [49], authors investigate RESTful APIs design from a linguistic point of view to define and detect linguistic (anti)patterns affecting the understandability and reusability of software systems. To this aim, the authors introduce the DOLAR (Detection Of Linguistic Antipatterns in REST) approach, which combines both syntactic and semantic analyses for the detection of linguistic (anti)patterns.

Related work	**Late (/code)**	**Early (/model)**	**Architectural Feedback**
B. Dudney et al. [41]	√		
B. Tate et al. [42]	√		
J. Xu et al. [44]		√	
A. Koziolek et al. [45]		√	√
A. Wert et al. [46]	√		
N. Moha et al. [47]	√		
F. Palma et al. [48] [49]		√	√
T. Parsons et al. [43]	√		
Our approach		√	√

Table 1: Overview of related works.

Table 1 provides an overview of the presented related works. Approximately half of the approaches apply late in the development process (i.e., at the code level) and the remaining ones deal with early model abstractions. Only two approaches among the latter ones (i.e., [45] and [48, 49]) give feedback to support software architects, like we do in our approach. In particular, the type of feedback provided by our approach is similar to the one in [48, 49], because it consists in a list of detected performance antipatterns that may cause (more or less severe) performance problems. As opposite, the approach in [45] searches for all available architectural alternatives and then checks if they lead to improved performance. The latter approach has to be preferred when a simple search engine can be implemented and an unpredictable searching time can be afforded. The effort of implementing a more sophisticated antipattern-based detection engine is payed back by a deeper control of the refactoring process. In this case, in fact, the process can be driven by specific performance problems (e.g., because they concern critical subsystems), instead of searching for refactored architectures on the basis of a synthetic fitness function that flattens performance problems into a simple metric.

The main advantage that our approach provides with respect to these latter ones is the independence from any architectural paradigm, whereas [45] targets component-based systems and [48, 49] targets service-oriented architectures. At the same time, its main limitation is that the ability to detect and remove antipatterns depends on the syntax and semantics of Æmilia. Hence, the results presented in this paper are a first experience applied to an ADL, although this limitation would have been obviously encountered with any other ADL.

3. Reasoning on Performance Antipatterns in Æmilia

In this section, we provide an overview of the performance antipatterns that can be specified in Æmilia. Section 3.1 first provides basic information on Æmilia, then Section 3.2 reports on the detection and solution of antipatterns in Æmilia.

3.1. An Architecture Description Language: Æmilia

The architectural description language Æmilia [6] is based on the stochastic process algebra $EMPA_{gr}$ [50], and, similarly to other ADLs, it provides a formal architectural description of complex software systems.

The Æmilia specification describes how a system is structured: (i) the software architecture structure (topology) is specified by components, connectors, channels, and ports; (ii) the software architecture dynamics (behavior) is specified by states, transitions, and interactions. The study of the properties of Æmilia specifications is conducted with the TwoTowers tool [24] that enables functional verification via model checking or equivalence checking, as well as performance evaluation through the numerical solution of continuous-time Markov chains [51] or discrete-event simulation [52].

In the following, we report some of the Æmilia concepts that are relevant for antipatterns detection and solution, while a complete description can be found in [6].

```
ARCHI_TYPE <name and formal parameters>

ARCHI_ELEM_TYPE
<architectural element types:
behaviors and Interactions>

ARCHI_TOPOLOGY
ARCHI_ELEM_INSTANCES <architectural element instances>
ARCHI_INTERACTIONS <architectural interactions>
ARCHI_ATTACHMENTS <architectural attachments>
```

Listing 1: Structure of the Æmilia architectural specification.

Listing 1 reports the structure of the Æmilia architectural specification. There are three main concepts: (i) *Architectural Type*, i.e., an intermediate abstraction between a system and an architectural style; it contains the specification of formal parameters; (ii) *Architectural Element Type* (AET), i.e., a component or a connector type that describes the internal behaviors and the interactions of the component; (iii) *Architectural Topology* contains the specification of *Architectural Element Instances* (AEI), i.e., an instance of a given AET type, their *Architectural Interactions* and *Architectural Attachments*, i.e., communication links among instances modeled as synchronization actions.

The performance evaluation of an Æmilia specification is executed by means of the tool TwoTowers [24]. It has been realized, besides other purposes, also for the performance evaluation of software systems modelled with the Æmilia language. The Performance Evaluator of the TwoTowers tool can generate, by starting from an Æmilia Architectural Specification, a performance model represented as a Markov chain. On this model, it calculates a set of instant-of-time, stationary/transient performance measures specified through state and transition rewards [53]. It can also estimate the values of the mean, variance and distribution of some performance measures specified as extensions of the state and transition rewards with the number and length of the simulation intervals.

Listing 2 reports a simple example of the Æmilia performance specification that defines the throughput measure by assigning a transition reward equal to 1.0 to an architectural element instance (AEI) interaction under analysis.

```
MEASURE throughput IS
ENABLED
(AEI.interaction) -> TRANS_REWARD(1.0)
```

Listing 2: A simple example of the Æmilia Performance Specification.

Listing 3 reports a simple example of the Æmilia performance results stored in a .val file: the value of the average throughput is equal to 12.

```
Value of measure "throughput":
12
```

Listing 3: A simple example of the Æmilia Performance Results.

3.2. Performance Antipatterns in Æmilia at a Glance

The management of performance antipatterns is limited by the expressiveness of the target ADL, because antipatterns have a notation-independent representation [14] that must be adapted to the actual ADL under analysis. Hence, depending on the target ADL, there are antipatterns that can be detected and solved, others that can be partially detected and-or partially solved, and finally some others that are neither detectable nor solvable.

Performance antipatterns are very complex (as compared to software patterns) because they are founded on different characteristics of a software system, spanning from *static* to *behavioral* to *deployment*, and they additionally include values of performance indices [14]. As consequence, depending on the ability of the target ADL to model the software system features needed to define each performance antipattern, we can have different detection capabilities. In Æmilia, deployment features cannot be modeled leading to a partial capability of antipatterns detection and solution, as detailed in the following.

Type	**Antipattern**		**Detectable**	**Solvable**
Snapshot	Blob		$\simeq$	$\simeq$
	Unbalanced Processing	Concurrent Processing Systems	$\times$	$\times$
		Pipe and Filter Architectures	$\surd$	$\surd$
		Extensive Processing	$\surd$	$\surd$
	Circuitous Treasure Hunt		$\times$	$\times$
	Empty Semi Trucks		$\times$	$\times$
	Tower of Babel		$\times$	$\times$
	One-Lane Bridge		$\simeq$	$\simeq$
	Excessive Dynamic Allocation		$\simeq$	$\times$
Evolution	Traffic Jam		$\surd$	$\surd$
	The Ramp		$\surd$	$\times$
	More is Less		$\times$	$\times$

Table 2: Detectable and solvable performance antipatterns in Æmilia (symbols $\surd$, $\times$, $\simeq$ mean *yes*, *no*, and *partially*, respectively).

Table 2 lists the most common performance antipatterns that we examine (from [12]). Such list has been enriched with an additional attribute: the *Snapshot* antipatterns are detectable by looking at mean, max or min values of performance indices, whereas the *Evolution* antipatterns need, to

Antipatterns	**Textual Descriptions**
Blob	It is an antipattern whose bad practice is on a single software component either performs all of the work of an application or holds all of the application's data. Either manifestation results in excessive message traffic that can degrade performance [12].
Circuitous Treasure Hunt	It is an antipattern whose bad practice is on an efficient management of database systems, in particular an object must look in several places to find the information it needs [12]. The inner organization of database systems cannot be modeled in Æmilia.
Concurrent Processing Systems	It is an antipattern whose bad practice is on the unbalanced assignment of processing to the available processors [12], but in Æmilia no hardware properties can be specified.
Empty Semi Trucks	It is an antipattern whose bad practice is on an inefficient use of available bandwidth [12], but in Æmilia the communication flow is more abstract and does not include information on bandwidth.
Excessive Dynamic Allocation	It is an antipattern whose bad practice is on the excessive memory storage due to objects that are rarely used: an object is created when needed and immediately destroyed afterwards. It may happen that an application unnecessarily creates and destroys large numbers of objects during its execution [12].
Extensive Processing	It is an antipattern whose bad practice is represented by a long running process that monopolizes a processor and prevents a set of other jobs to be executed until it finishes its computation. Unlike the pipe and filter, other work does not have to pass through this stage before proceeding, hence it is particularly problematic if the extensive processing is on the processing path that is most frequently executed [12].
More is Less	It is an antipattern whose bad practice is on the overhead spent by the system in thrashing, as compared to accomplish the real work, because there are too many processes in comparison to the available resources [12]. Currently thrashing cannot be modeled in Æmilia.
One-Lane Bridge	It is an antipattern whose bad practice is on concurrent systems when the mechanisms of mutual access to a shared resource are badly designed. The problem appears in all cases of mutual access to resources, or in the case of multiple processes making synchronous calls in a non multi-threaded environment [12].
Pipe and Filter Architectures	It is an antipattern whose bad practice represents a manifestation of the unbalanced processing antipattern [12]. It occurs when the throughput of the overall system is determined by the slowest filter. It means that there is a stage in a pipeline which is significantly slower than all the others, therefore most stages have to wait the slowest one to terminate.
The Ramp	It is an antipattern whose bad practice is on the amount of processing required by a system to satisfy a request since it increases over time. It is due to the growing amount of data the system stores and, as the time goes on, the data grow and the processing time required to perform an operation on such data becomes unacceptable [12].
Tower of Babel	It is an antipattern whose bad practice is on the translation of information into too many exchange formats, i.e. data is parsed and translated into an internal format, but the translation and parsing is excessive [12]. In Æmilia the data flow is more abstract and it does not include information on data formats.
Traffic Jam	It is an antipattern whose bad practice is on the observation of a significative variability in response time. It is due to different causes, and the problem also occurs when a large amount of work is scheduled within a relatively small interval [12].

Table 3: Performance Antipatterns textual description [12].

be detectable, to look at the trend or evolution of the performance indices along the time. Table 2 is organized as follows: each row represents a specific antipattern and it is characterized by three fields (one per column), that are: the *antipattern* name, *detectable* and *solvable* that specify the degree of detection and solution in the Æmilia ADL. Such attributes can be of three different values: $\surd$ meaning that the antipattern is fully detectable and solvable, $\simeq$ meaning that the antipattern is partially detectable and solvable, and $\times$ if we think that the antipattern cannot be detected and cannot be solved.

Table 2 points out that the most interesting antipatterns in Æmilia are: Pipe and Filter Architectures, Extensive Processing, Traffic Jam and The Ramp (see Table 3 for their textual description).

The first three antipatterns can be referred as *solvable* antipatterns, because they can be detected and solved. In particular, the Pipe and Filter Architectures and the Extensive Processing antipatterns are defined by considering static features of the software system and performance indices values, which belong to any Æmilia model. The Pipe and Filter Architectures is detected by identifying the slowest filter component whose execution limits the throughput of the overall system. The Extensive Processing is detected by identifying a process whose execution absorbs a processor, thus starving other jobs and leading to very high response time. The Traffic Jam antipattern is defined by considering behavioural feature of the software system along with performance indices over time, which again belong to any Æmilia model. The detection of this antipattern is based on identifying excessive variabilities of response time values in subsequent time slots.

In contrast, The Ramp can be referred as *detectable* antipattern, because it can be detected by observing the trend of the response time and the throughput over time [12]. The solution of such antipattern refers to the selection of algorithms or data structures based on maximum size or the usage of algorithms that adapt to the size [12], hence it cannot be solved because it refers to an abstraction layer not included in the current Æmilia language.

Table 2 reveals that there are three antipatterns (i.e., Blob, One-Lane Bridge, and Excessive Dynamic Allocation) that can be partially detected.

Blob and One-Lane Bridge can be referred as *semi-solvable* antipatterns, because they can be partially detected and solved. Their detection is based on static and behavioral features of the system as well as on deployment aspects. In particular, to detect the Blob antipattern we must identify, from a static point of view, a single software component that either performs all the work of an application or holds all the application data [12]. Either manifestation results, from a behavioral perspective, in an excessive message traffic with several other software components. In addition, the Blob antipattern depends on the deployment of software components, because its manifestation can degrade the utilization of hardware resources that cannot be modelled in Æmilia. Because the Blob antipattern is also founded on deployment aspects, it is *semi-solvable*. The One-Lane Bridge antipattern occurs in concurrent systems whose mechanisms of mutual access to a shared resource are badly designed. It also arises in the case of multiple processes making synchronous calls in a non multi-threaded environment [12]. It is detected by analyzing, for each software component in the system, the number of synchronous calls it receives for one of its services and how this affects the service response time. Moreover, it is also required to measure both the service time and the waiting time of the process nodes on which the software components are deployed. The Blob and One-Lane Bridge antipatterns are semi-detectable and semi-solvable because they can be partially modeled in Æmilia specifications, since the deployment constraints required for their detection are left out.

Excessive Dynamic Allocation can be referred as *semi-detectable* antipattern, because it can be partially detected by observing the system response time. Its manifestation also depends on the overhead caused by a high number of creation and destruction of objects, which cannot be modeled in the Æmilia specification [12]. The solution of such antipattern refers to the recycle of objects or the avoidance of creating new objects [12], hence it cannot be solved because it refers to an abstraction layer not included in the current version of the Æmilia language.

Eventually, Table 2 indicates that there are five antipatterns (i.e., Concurrent Processing Systems, Circuitous Treasure Hunt, Empty Semi Trucks, Tower of Babel, and More is Less) neither detectable nor solvable in Æmilia. For each of these performance antipatterns, their detection cannot be performed because they rely on particular static and/or behavioral features of software systems that are intrinsically difficult to model in any specification, even in Æmilia. Going more into details, the Concurrent Processing Systems antipattern stems from the unbalanced assignment of processing to the available processors [12]. It relies on hardware properties, such as the queue length of the involved platform devices, as well as their utilization, that Æmilia cannot model. Looking at other ADLs, this antipattern can be partially detected if EAST-ADL [20] is employed. In fact, such language allows hardware modelling with which it is possible to define the utilization of devices, whereas it lacks the concept of queue length. The Circuitous Treasure Hunt antipattern reports the inefficient management of database systems occurring when an object must look in several places to find the information that it needs, thus causing additional overhead that degrades performance [12]. However, it is not always possible to identify a specific `elemType` as a database, as well as it is not possible to distinguish the messages sent to databases from all the other ones. This antipattern cannot be detected with other ADLs because, to the best of our knowledge, it is not possible to explictly model database components in ADL specifications. The Empty Semi Trucks antipattern is the result of an inefficient use of the available bandwidth. This happens especially in message-based systems, when a huge load of messages is exchanged over the network [12]. For the detection of the Empty Semi Trucks antipattern, information on the bandwidth utilization is needed, together with deployment information referring to the number of instances deployed on remote nodes. Again, deployment features cannot be modeled in Æmilia. Looking at other ADLs, this antipattern can be partially detected if AADL [21] is employed. In fact, such language allows hardware modelling with which it is possible to define communication networks, whereas it lacks the concept of bandwidth used for message passing. The Tower of Babel antipattern is triggered when information is translated into many exchange formats, thus requiring additional overhead for converting, analyzing, and translating data in a common exchange format (e.g., XML) [12]. This is a typical problem in distributed data-oriented systems, where the same information is translated several times. However, in Æmilia, the data flow does not include information on data formats. This antipattern cannot be detected with other ADLs because, to the best of our knowledge, it is not possible to explictly define data formats in ADL specifications. Finally, the More is Less antipattern is detected when there are too many processes in comparison to the available resources, thus causing overhead [12]. For example, the system spends more time in running secondary processes (e.g., trashing, connecting to databases and web services) with respect to the execution of the actual work. Also in this case, deployment features are needed to detect this antipattern, i.e., the deployment of software components on platform nodes, how they communicate, and how many messages they exchange. Moreover, the amount of available resources with respect to the running processes has to be known. Looking at other ADLs, this antipattern can be partially detected if EAST-ADL [20] and AADL [21] are employed, in fact: (i) EAST-ADL includes failure modelling thus it is possible to define internal and external faults that are either propagated to system failures or masked; (ii) AADL supports fault propagation analysis by specifying the fault category (benign, tolerated, catastrophic) and recovery actions. However, both these two ADLs lack the concept of database and web services connections.

Summarizing, our study on the detection of performance antipatterns in Æmilia demonstrates that there are some concepts that are intrinsically more difficult to express both in Æmilia and in other ADLs. As future work, we plan to deal with these specific concepts. A solution may be to augument the Æmilia metamodel with this information and let the designer manually annotate these values.

4. Representing and Detecting Performance Antipatterns in Æmilia

In this section, we present our model-driven approach to catch the performance antipatterns in the Æmilia ADL. Figure 1 illustrates the round-trip performance analysis process that we envisage considering the Æmilia context, where shaded boxes represent the main contributions of this paper. The presented approach is tool supported, we developed an Eclipse-based tool used to automatically detect the performance antipatterns. Our tool can be downloaded on-line [54].

Figure 1 is partitioned in two parts: the *M1:Models* part contains activities and artifacts (i.e., textual specifications and models of the software system under analysis) implementing the process, whereas the *M2:Metamodels* part contains grammars, metamodels, and OCL rules.

The lower side has been partitioned in three phases: *Modeling*, *Analysis* and *Refactoring*. In the *Modeling* phase an Æmilia specification of the software architecture, conforming to the Æmilia grammar, is produced. In the *Analysis* phase the performance evaluation activity takes as input the architecture, along with the performance parameters and the specification of indices of interest (also conforming to the Æmilia grammar), and it produces as output the values of specified indices (such as throughput and response time). These two phases are carried out within the TwoTowers [24] tool, that is the Æmilia supporting environment.

The *Refactoring* phase is the core of this paper. The approach that we adopt to detect performance antipatterns in an Æmilia architectural specification is based on our previous work on antipatterns. In particular, we base on [14, 55] to introduce here a set of OCL rules that encode performance

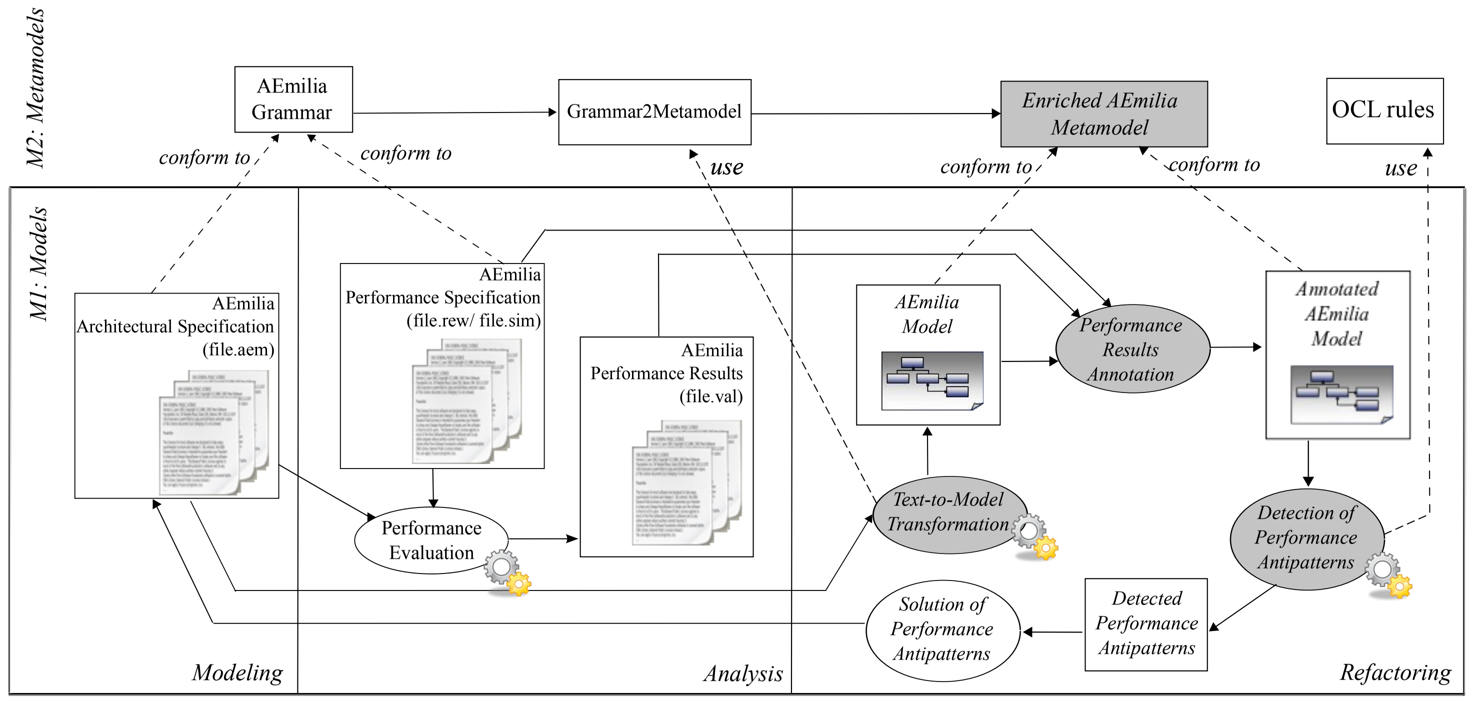

M2: Metamodels
AEmilia Grammar
Grammar2Metamodel
Enriched AEmilia Metamodel
OCL rules
conform to
conform to
use
conform to
conform to
use
M1: Models
AEmilia Architectural Specification (file.aem)
AEmilia Performance Specification (file.rew/ file.sim)
Performance Evaluation
AEmilia Performance Results (file.val)
AEmilia Model
Text-to-Model Transformation
Performance Results Annotation
Annotated AEmilia Model
Detection of Performance Antipatterns
Detected Performance Antipatterns
Solution of Performance Antipatterns
Modeling
Analysis
Refactoring

Figure 1: Round-trip performance analysis process in Æmilia.

antipatterns and allow their detection on models conforming to the Æmilia metamodel we have defined.

The Refactoring phase starts with a *Text-to-Model Transformation* from an Æmilia architectural specification to an Æmilia model, where the latter conforms to the enriched Æmilia metamodel (described in Section 4.1). Because antipatterns insist on performance indices, beside architectural characteristics, *Performance Results Annotation* activity is devised to enrich an Æmilia Model with performance parameters and indices as coming from the Analysis phase. An Annotated Æmilia Model, again conforming to the enriched Æmilia Metamodel, is thus ready for the *Detection of Performance Antipatterns*.

The detection activity produces a list of performance antipatterns occurrences identified in the Æmilia model of the software system under analysis. This list represents a relevant instrument to guide software architects while looking for causes of performance problems, as shown in Section 5. The round-trip process is closed by the performance antipatterns solution activity that appropriately removes antipatterns in the Æmilia specification to achieve better performance.

Hereafter, we define a metamodel (Section 4.1) that defines the model elements used in the specification of antipatterns. Hence, the Æmilia specification and its performance evaluation can be represented as a model (Section 4.2) conforming to the metamodel. Such representation is suitable to automatically detect the performance antipatterns (Section 4.3). Finally, we discuss precision and recall metrics (Section 4.4).

4.1. A metamodel for Æmilia

Figure 2 reports the *Enriched Æmilia metamodel*. An `AEmiliaSpecification` specifies (and hence contains) an `ArchiType` defining a family of architectures sharing the same structure and behavior. An `ArchiType` has a name (*atName*) and is composed of three elements: *i)* an `AT_Header` representing the name of the architectural type and specifying a set of constant parameters (e.g., rates and instances number); *ii)* the `ArchiElemTypes` specifying a set of `ElemTypes`; and *iii)* the `ArchiTopology` defining the topology of the running system at the architectural level.

`ElemType` is denoted by an *etName* and defines a new type of software component by specifying its header (`ET_Header`), its internal `Behavior`[1] and

[1]`AT_Header` and `ET_Header` are detailed in the Headers package, `Behavior` is detailed

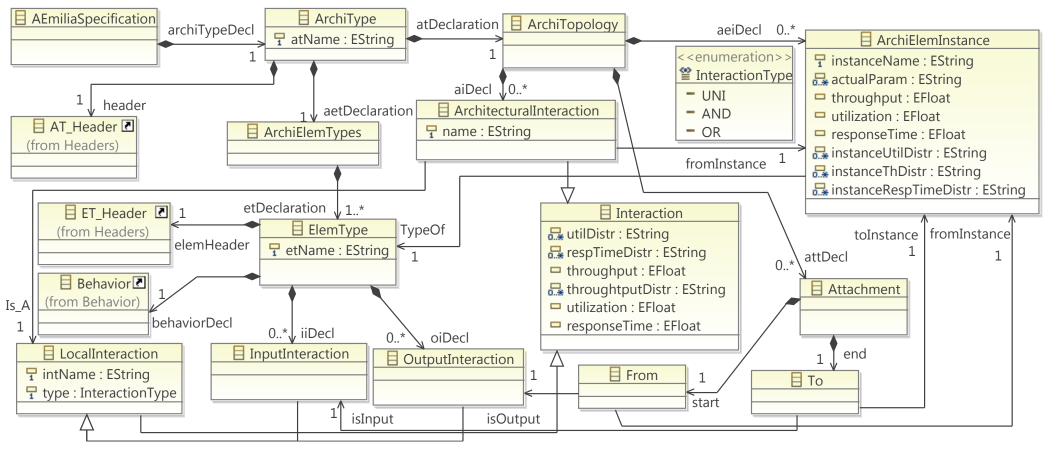


Figure 2: Enriched Æmilia metamodel.

its interactions (i.e., `InputInteraction` and `OutputInteraction`). The interactions allow the `ElemType` instances to communicate with other instances. In particular, the `InputInteraction` (`OutputInteraction`) models the `LocalInteraction` incoming (respectively outcoming) the `ElemType`. A `LocalInteraction` is denoted by an *intName* and it could be of three *type*s (*UNI*, *AND*, *OR*) defining one-to-one, broadcast and client/server communications.

`ArchiTopology` specifies the architecture configuration of the running system. It is composed of (possible zero) `ArchitecturalInteractions`, `ArchiElemInstances`, and `Attachments`. An `ArchiElemInstance` models an instance of an `ElemType`. It is denoted by an *instanceName* and the list of *actualParam* that provides the actual parameters for the formal ones specified in the `ElemType`'s header. `ArchitecturalInteraction` is exclusively needed in case of the hierarchical specification of the system. It has a *name* attribute and it models the interactions that the modelled (sub)system has with its surroundings specified in an external `AEmiliaSpecification`. It is provided by an `ArchiElemInstance` and it corresponds to one of the `LocalInteractions` defined in the `ArchiElemInstance`'s `ElemType`. An `Attachment` models a communication link between two `ArchiElemInstances`. The communication is oriented and goes from an output interaction of an instance (modeled by

in the Behavior package of the metamodel. The interested reader can refer to [54] for more details on them.

the `FROM` element) to an input interaction of a different instance (modeled by the `TO` element).

To allow the detection of performance antipatterns, `Interaction` and `ArchiElemInstance` classes of the Æmilia metamodel have been enriched with attributes that permit to annotate the model with performance analysis results obtained with the TwoTowers tool [24]. The results that can be used to annotate an `Interaction` or an `ArchiElemInstance` are: the throughput (*throughput*), the response time (*responseTime*), the utilization (*utilization*), and their distributions (annotated in *throughputDistr*, *respTimeDistr* and *utilDistr* attributes for the interaction, and *instanceThDistr*, *instanceRespTimeDistr*, and *instanceUtilDistr* attributes for the architectural element instance).

Finally, we have implemented a set of OCL rules [54] to code some constraints imposed by the Æmilia ADL (such as each local interaction must be involved in at least one attachment). Such rules check the validity of the Æmilia model before performing the detection of performance antipatterns.

4.2. Transforming an Æmilia specification into an Æmilia model

The process of transforming an Æmilia specification into an Æmilia model is carried out by two operational steps (see Figure 1): *1)* a *text-to-model transformation* step that produces an Æmilia model; *2)* a *performance results annotation* step that annotates the model with the results obtained from the performance analysis.

4.2.1. Text-to-Model Transformation

After a metamodel for the Æmilia ADL has been defined, a text-to-model transformation converts an Æmilia textual specification into a model. This text-to-model transformation is performed by using Gra2Mol (Grammar to Model Language), i.e., a domain specific transformation language that defines the relationships between grammar elements and metamodel elements [56]. Indeed, the approach is based on the definition of a grammar-to-model transformation language, that is Gra2Mol. It is oriented to extract models conforming to a target metamodel, starting from source programs conforming to a specific grammar.

The Gra2Mol transformation process starts by building a syntax tree from both the source grammar and the source code. The defined Gra2Mol transformation interacts with the syntax tree and, by exploiting the source grammar, it defines the nodes types of the syntax tree. Finally, the model conforming to the target metamodel is defined by applying the transformation rules.

As already introduced, Gra2Mol is a rule-based language. Each rule defines a relation between a grammar element and a metamodel element. A rule is made by four parts, namely (i) *from* specifies the non-terminal source grammar symbol, (ii) *to* specifies the destination meta-class, (iii) *queries* contains a set of queries that allow the engine to navigate the syntax tree in order to retrieve the information sought, and (iv) *mapping* contains a set of associations which assign a value to the properties of the elements of the target model (an example of a transformation rule is reported in Listing 5).

In Figure 3, we report the detail of Gra2Mol execution process contextualized to our work. The parameters of a Gra2Mol transformation are:

- the **source grammar**, i.e., the *ÆmiliaGrammar.g* file, in which the grammar of the source language is defined in EBNF format, as required by Gra2Mol;
- the **source code**, i.e., the *ÆmiliaSpec.aem* files, to be transformed;
- the **target metamodel**, i.e., the enriched *Æmilia metamodel*, which the resulting model must conform to;
- the definition of the **Gra2Mol transformation**, i.e., the *ÆmiliaTranformation.g2m* file, in which the rules for the transformation between source elements (grammar) and target elements (metamodel) are defined.

The text-to-model transformation has been realized as a standalone project. We have first defined the EBNF format of the Æmilia ADL grammar and, then, we have defined all the transformation rules to automate the Æmilia models generation.

An excerpt of an Æmilia Architectural Specification (see Figure 3) is reported in Listing 4. An `Archi_Type` contains a set of `Archi_Elem_Type`s, e.g., a mobile application of the travel domain (`MA_Type`) has the following behavior: (i) generates a request for the best path (i.e., *generate_best_path_req*); (ii) transmits the request for the best path (i.e., *transmit_best_path_req*); (iii) waits until receiving the best path (i.e., *receive_best_path*). Moreover, the `Archi_Type` specifies also the `Archi_Topology` of the running system at the architectural level.

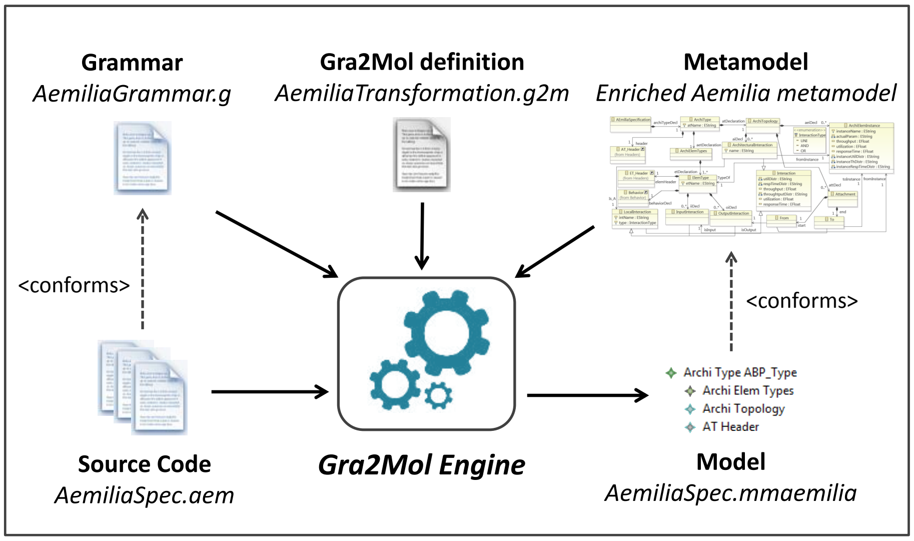


Figure 3: The execution process of Gra2Mol [56].

```
ARCHI_TYPE gbp(<name and formal parameters>)
ARCHI_ELEM_TYPES

 ELEM_TYPE MA_Type(void)
    BEHAVIOR
     MobileApp(void; void) =
         <generate_best_path_req, inf> .
                    <transmit_best_path_req, inf> .
                     <receive_best_path, _> . MobileApp()
  INPUT_INTERACTIONS
    UNI receive_best_path; UNI generate_best_path_req

  OUTPUT_INTERACTIONS
    UNI transmit_req_best_path

     ELEM_TYPE NetDown_Type(..)
     ELEM_TYPE NetUp_Type(..)
     ELEM_TYPE Balancer_Type(..)
     ELEM_TYPE Server_Type(..)
     ELEM_TYPE DB_Type(..)

ARCHI_TOPOLOGY
  ...
END
```

Listing 4: A simple example of an Æmilia Architectural Specification.

Listing 5 reports a simple example of a rule that defines how to map the Æmilia architectural specification to the Æmilia model. In particular, the mapping is performed *from* the elem_type of the Æmilia specification *to* the ElemType metamodel element, and *queries* support the retrieval of ElemType features by traversing the syntax tree (i.e., the header, the input and output interactions, and the behavior).

```
rule 'mapElemTypes'
 from elem_type et
 to ElemType
 queries
  name : /#et;
  elemHeader : /et/#et_header;
  inputInt : /et//interaction_list_input//#interactionInput;
  outputInt : /et//interaction_list_output//#interactionOutput;
  behavior : /et/#behavior_equation_list;
 mappings
  etName = name.WORD;
  elemHeader = elemHeader;
        ...
end_rule
```

Listing 5: A simple example of a Text-to-Model Transformation Rule.

Figure 4 shows the Æmilia Model (see Figure 3) automatically obtained through the defined text-to-model transformation rules [54] applied to the Æmilia Architectural Specification in Listing 4. In such a model the performance results are omitted for sake of readability, for example the shaded input interaction *generate_best_path_req* has a throughput (i.e., a performance index specified in the metamodel, see Figure 2) set to zero.

In the following section, we will see how the Æmilia model generated by the text-to-model transformation process is annotated with the results obtained from the performance analysis.

4.2.2. Performance Results Annotation

The process of annotating an Æmilia model with performance results gives as output the Annotated Æmilia Model (see Figure 1), i.e., the model representation suitable for the detection of antipatterns.

Listing 6 reports a simple example of the Æmilia performance specification that defines the measure `MA_throughput` for all mobile applications by assigning a transition reward equal to 1.0 for the *generate_best_path_req* interaction.

```
MEASURE MA_throughput IS
FOR_ALL i IN 1..ma_num
ENABLED
  (MA[i].generate_best_path_req) -> TRANS_REWARD(1.0)
```

Listing 6: A simple example of the Æmilia Performance Specification.

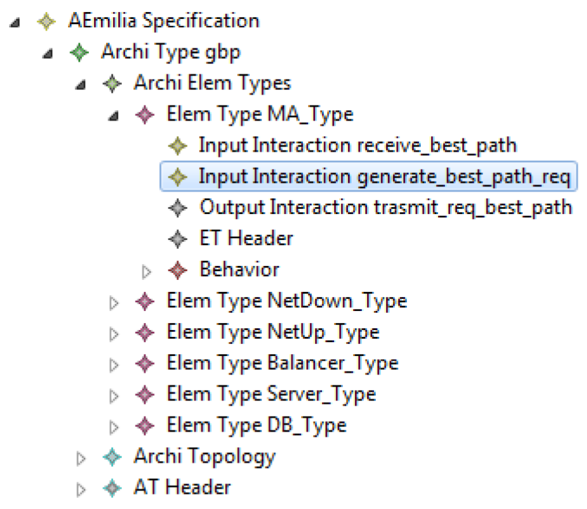


Figure 4: A simple example of the Æmilia Model.

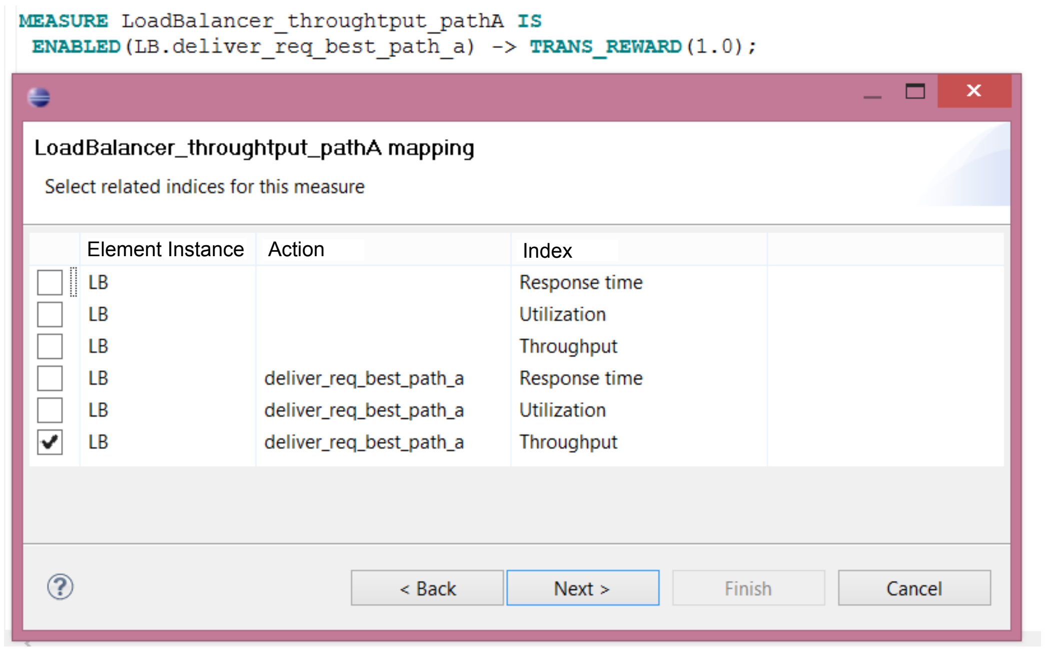
Listing 7 reports a simple example of the Æmilia performance results stored in a .val file (see Figure 1): the value of the measure `MA_throughput` as defined in Listing 6 is equal to 36.5841. Once the performance evaluation of an Æmilia specification has been executed, the resulting values of the performance measures can be used to annotate the Æmilia model. For example, the value stored in the .val file of the Listing 7 has been annotated to the input interaction *generate_best_path_req* (see Figure 4) where the throughput is set to 36.5841.

```
Value of measure "MA_throughput":
36.5841
```

Listing 7: A simple example of the Æmilia Performance Results.

We implemented a wizard to support software architects in the task of annotating the specified measures with the actual performance indices to which they refer. In particular, the wizard shows all possible mappings among instances, their actions and the obtained performance indices. By selecting the proper matching we are able to automatically annotate the Æmilia model without incurring on wrong interpretations of measures specified with the Æmilia grammar.

Figure 5 reports all the alternative mappings for the example specification at the top of the Figure: the first three lines denote the system (response time, utilization, throughput) indices; the last three lines are related to a specific



```
MEASURE LoadBalancer_throughput_pathA IS
 ENABLED(LB.deliver_req_best_path_a) -> TRANS_REWARD(1.0);
```

LoadBalancer_throughput_pathA mapping

Select related indices for this measure

	Element Instance	Action	Index
☐	LB		Response time
☐	LB		Utilization
☐	LB		Throughput
☐	LB	deliver_req_best_path_a	Response time
☐	LB	deliver_req_best_path_a	Utilization
✓	LB	deliver_req_best_path_a	Throughput

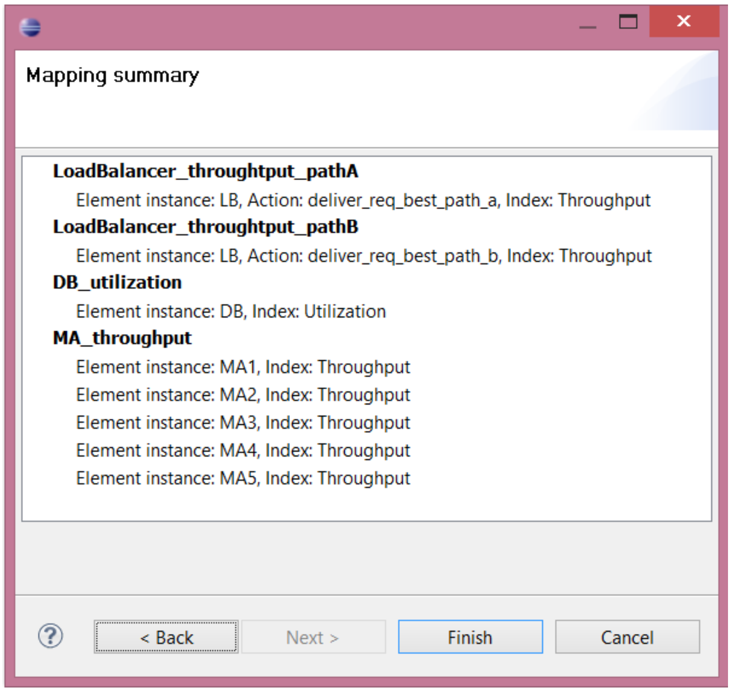
Figure 5: Screenshot of the wizard mapping Æmilia measures to performance indices.

action, i.e., *deliver_req_best_path_a*. The checkbox on the last line indicates that in this case the measure refers to the throughput of the specific action belonging to the LB element instance. In this way, designers are supported in the activity of matching performance results to architectural actions under analysis.

Figure 6 shows the wizard that represents the summary of all specified mappings. In this way we are able to fully annotate the Æmilia model.

4.3. Detection of Performance Antipatterns

The Model-Driven Engineering (MDE) approach has been widely used with the aim (among many others) of simplifying the design process of software systems, e.g., through the construction and the analysis of domain-specific models. By following a MDE approach, in this work, we have first introduced an enriched Æmilia metamodel, with the aim of providing an Æmilia conceptual model as a basis for reasoning on software architectures defined through the Æmilia language. After transforming an Æmilia textual specification into an Annotated Æmilia model, we exploit it for applying the automatic detection of performance antipatterns. In this section, we describe


Mapping summary

LoadBalancer_throughput_pathA
Element instance: LB, Action: deliver_req_best_path_a, Index: Throughput
LoadBalancer_throughput_pathB
Element instance: LB, Action: deliver_req_best_path_b, Index: Throughput
DB_utilization
Element instance: DB, Index: Utilization
MA_throughput
Element instance: MA1, Index: Throughput
Element instance: MA2, Index: Throughput
Element instance: MA3, Index: Throughput
Element instance: MA4, Index: Throughput
Element instance: MA5, Index: Throughput

Figure 6: Screenshot of the wizard: summary of all defined mappings.

the performance antipatterns detection process: we have defined OCL rules to analyze the (Annotated) Æmilia Model, with the aim to look for performance antipatterns. OCL rules implement the notation-independent rules that we have defined in [14], where a technique based on first-order logic has been introduced to formalize antipatterns. We use the OCL language because it is currently considered a standard rule-based validation language [57]. Moreover, OCL can be straightforwardly applied to Eclipse Modelling Framework (EMF)-based models. It allows developers to define unambiguous constraints that cannot be expressed by the metamodel itself, thus augmenting its expressivity.

Detection rules contain parameters that must be set on a specific software architecture whose performance indices have been already obtained. In fact, each antipattern is defined with a set of thresholds that basically represent systems features, in particular they may refer to upper/lower bounds for: (i) design properties (e.g., *excessive* message traffic); (ii) performance results (e.g., *high*, *low* utilization). Numerical values can be assigned by software architects basing on heuristic evaluations or they can be set on the basis of system monitoring information. Threshold values affect the detection and solution of performance antipatterns. In fact the set of detected antipatterns may change while varying the threshold values, as discussed in Section 6.

As introduced in Section 3.2, performance antipatterns can be classified into two categories: the *Snapshot* antipatterns are detectable by looking at mean, max or min values of performance indices, whereas for the detection of *Evolution* antipatterns it is needed to look at the trend or evolution of the performance indices along the time. In the following, we give the OCL rules for the detection of Snapshot antipatterns and Evolution antipatterns respectively[2].

As regard the *detectable* antipatterns in the Æmilia language, in the first category (Snapshot) we have the *Pipe & Filter* antipattern and the *Extensive Processing* antipattern.

```
1 def: pipeAndFilterPA(service: ArchitecturalInteraction) : Boolean =
2  let serviceThLB : Real = <threshold numerical value> in
3  let opResDemUB : Real = <threshold numerical value> in
4  let elemTypeName : ElemType = service.fromInstance.TypeOf in
5 if (isServiceExpOrInf(service) and service.throughput < serviceThLB) then
6  checkOpResDemand(service, opResDemUB)
7 else
8  false
9 endif
```

Listing 8: OCL code for the *Pipe and Filter Architectures* antipattern.

Listing 8 reports the OCL rule to detect *Pipe & Filter* antipattern that is identified in services that represent architectural interactions (line 1). Two threshold values are defined (lines 2-3): (i) *serviceThLB* denotes the lower bound for the service throughput; (ii) *opResDemUB* denotes the upper bound for the operation resource demand. The antipattern occurs if there exists a service whose throughput is lower than the *serviceThLB* threshold (line 5), and if the function *checkOpResDemand* finds an operation (executed within the service) having a resource demand greater than the *opResDemUB* threshold (line 6).

Listing 9 reports the OCL rule to detect *Extensive Processing* antipattern that is identified in services that represent architectural interactions (line 1). Three threshold values are defined (lines 2-4): (i) *respTimeUB* denotes the upper bound for the service response time; (ii) *opResDemLB* denotes the lower bound for the operation resource demand; (iii) *opResDemUB* denotes the upper bound for the operation resource demand. The antipattern occurs

[2]For sake of space in the sequel of this section we report the most representative OCL rules as examples, however the complete set of OCL rules can be found in [54].

if there exists a service whose response time is greater than the *respTimeUB* threshold (line 6), and if the function *unbalancedOpResDemand* finds two operations (executed within the service), i.e. op_1 and op_2, having an unbalanced resource demand: the resource demand of op_1 is lower than the *opResDemLB* threshold, whereas the resource demand of op_2 is greater than the *opResDemUB* threshold (line 7).

```
 1 def: extensiveProcessingPA(service: ArchitecturalInteraction) : Boolean =
 2  let respTimeUB : Real = <threshold numerical value> in
 3  let opResDemLB : Real = <threshold numerical value> in
 4  let opResDemUB : Real = <threshold numerical value> in
 5  let elemTypeName : ElemType = service.fromInstance.TypeOf in
 6 if (isServiceExpOrInf(service) and service.responseTime > respTimeUB) then
 7  unbalancedOpResDemand(service, opResDemLB, opResDemUB)
 8 else
 9  false
10 endif
```

Listing 9: OCL code for the *Extensive Processing* antipattern.

In the second category (Evolution), instead, we found the *Traffic Jam* antipattern and *The Ramp* antipattern. These antipatterns can be detected by observing the trend of the response time and-or the throughput over time.

Listing 10 reports the OCL rule to detect *Traffic Jam* antipattern that is identified in actions that represent the behavior of *ElemType* instances (line 1). One threshold value is defined (line 2): *respTimeUB* denotes the upper bound for the action response time.

The antipattern occurs if a significative variability in the response time of the action is observed. The function *calcIndexVariability* (see Listing 11 for its implementation) takes as input the *responseTimeDistr* attribute of an action and it calculates the response time variability for that action, even in several subsequent time intervals (line 6 of Listing 10).

```
 1 def: trafficJamPA(action: Behavior::Action) : Boolean =
 2  let respTimeUB : Real = <threshold numerical value> in
 3  let respTimeDistr : Sequence(Real) = action.actRespTimeDistr ->
 4  collect(x:String | x.toReal()) in
 5 if (respTimeDistr -> size() <> 0) then
 6  let diffArray : Sequence(Real) = calcIndexVariability(respTimeDistr) in
 7 if(diffArray -> exists(value: Real | value > respTimeUB)) then
 8  trafficJamSearch(diffArray, respTimeUB)
 9 else
10  false
11 endif
12 else
13  false
14 endif
```

Listing 10: OCL code for the *Traffic Jam* antipattern.

```
def: calcIndexVariability(set: Sequence(Real)) : Sequence(Real) =
 let result : Sequence(Real) = Sequence{} in
if(set -> size() <> 0)then
if(set -> size() > 1) then
 let secondElem : Real = set -> at(set -> indexOf(set -> first()) + 1) in
 let firstElem : Real = set -> first() in
 let diff : Real = secondElem - firstElem in
 let absValue : Real = diff.abs() in
 result -> including(absValue) -> union(calcIndexVariability(set ->
     excluding(firstElem)))
else
 result
endif
else
 result
endif
```

Listing 11: OCL code for the *calcIndexVariability* function.

Then, the existence of a peak value in the response time distribution greater than *respTimeUB* is verified (line 7 of Listing 10). If it exists, then the function *trafficJamSearch* verifies if there are occurrences of the traffic jam antipattern for the current action on the basis of its performance values and on its observed response time variability.

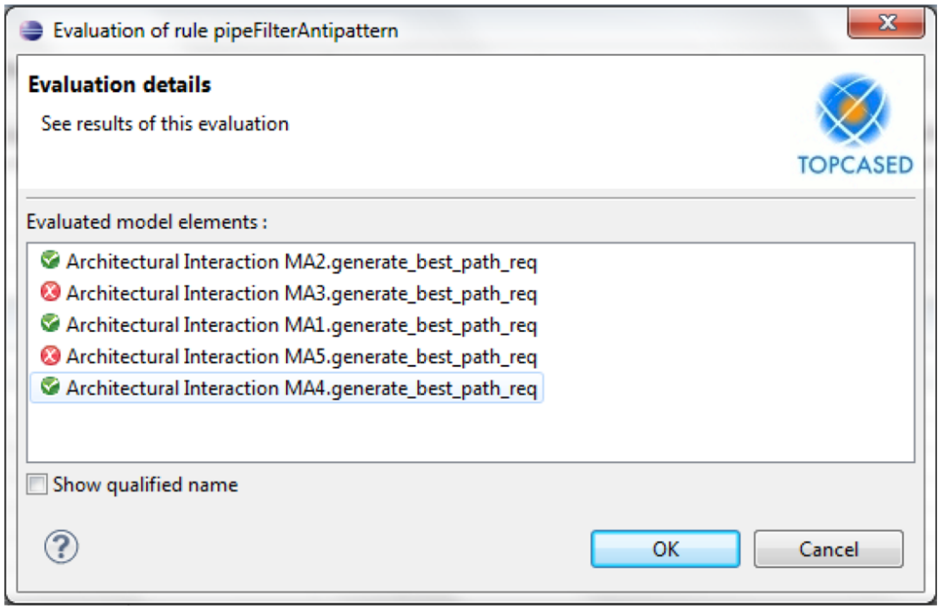
Listing 12 reports the OCL rule to detect *The Ramp* antipattern that is identified in actions that represent the behavior of *ElemType* instances (line 1). Two threshold values are defined (line 2-3): (i) *respTimeUB* denotes the upper bound for the action response time; (ii) *throughputUB* denotes the upper bound for the action throughput. The antipattern occurs if the amount of processing required by a system to satisfy a request increases over time. The function *calcAveragePI*, applied both on the performance value *respTimeDistr* and *throughputDistr* (lines 9-10), measures the observed trend of the response time and the throughput over time, respectively.

```
 1 def: theRampPA(action: Behavior::Action) : Boolean =
 2  let respTimeUB : <threshold numerical value> in
 3  let throughputUB : <threshold numerical value> in
 4  let respTimeDistr : Sequence(Real) = action.actRespTimeDistr ->
        collect(x:String | x.toReal()) in
 5  let throughputDistr : Sequence(Real) = action.actThDistr -> collect(x:
        String | x.toReal()) in
 6 if(respTimeDistr -> size() <> 0 and throughputDistr -> size() <> 0) then
 7  calcAveragePI(calcIndexVariability(respTimeDistr)) > respTimeUB and
 8  calcAveragePI(calcIndexVariability(throughputDistr))> throughputUB
 9 else
10  false
11 endif
```

Listing 12: OCL code for the *The Ramp* antipattern.

If there exists an action having both a response time growing more than the *respTimeUB* and a throughput decreasing more than the *throughputUB* (line 9-10), then an occurrence of The Ramp antipattern is detected.

The antipattern detection is performed by launching an OCL checker that takes in input the predefined antipatterns rules and analyzes the Annotated Æmilia model. As a result, a wizard listing the snapshot or evolution antipatterns rules that have been checked is shown. For each rule, the tool highlights if occurrences of the corresponding antipattern have been detected (represented by the $\surd$ symbol, see Figure 9) or if no occurrences have been found (represented by the $\times$ symbol, see Figure 10). Moreover, after selecting a detected antipattern's rule, a new wizard will show more details about it, such as the analyzed architectural elements, the involved ones because of their performance indices values. For example, in Figure 7 we can notice that there are five architectural interactions (i.e., $MA1, \ldots, MA5$) and three of them are detected as Pipe and Filter antipattern occurrences.


Evaluation of rule pipeFilterAntipattern

Evaluation details

See results of this evaluation

Evaluated model elements :

✓ Architectural Interaction MA2.generate_best_path_req
× Architectural Interaction MA3.generate_best_path_req
✓ Architectural Interaction MA1.generate_best_path_req
× Architectural Interaction MA5.generate_best_path_req
✓ Architectural Interaction MA4.generate_best_path_req

☐ Show qualified name

Figure 7: Pipe & Filter antipattern occurrences.

4.4. Precision and Recall for Performance Antipatterns

The set of detected performance antipatterns may change while varying the threshold values. To investigate this change, in [58] we have associated a *recall* metric to the detection activity and a *precision* metric to the refactoring activity. However, these metrics cannot be straightforwardly applied to the case of performance antipatterns because they are not based on deterministic values (such as the number of nested loops in a C-based software code), but mostly on stochastic values (such as the utilization of a CPU). Therefore, they have been re-defined as follows. Recall has been defined as the standard ratio between the number of detected antipatterns and the existing ones, but the latter quantity is considered as the number of all performance antipatterns that can be detected while varying the thresholds within predefined ranges. Precision has been defined as the ratio between the number of detected antipatterns that actually improve the system performance once removed and the total number of detected antipatterns. Precision and recall values are calculated at each step of antipatterns detection and solution, but their values change in an unpredictable manner across different solution steps. In fact, antipattern-based refactorings do not guarantee a priori performance improvements because the entire process is based on stochastic performance evaluation. After applying refactorings, we evaluate new systems in terms of performance, thus we get new precision and recall values that cannot be significant of any improvement/worsening when compared to values of the previous performance evaluation due to the heterogeneous sets of detected and solved antipatterns. In the next section, we show the application of these metrics on one of the presented case studies.

5. Validation

In this section, we present our approach at work on three examples to demonstrate that the process actually induces performance improvements on ADL-based software architectures[3]. To demonstrate the usefulness of the antipattern-based process, we compare our experimental results with the ones obtained by means of the well-known bottleneck analysis [10]. The first case study is extensively described to clearly explain the whole process, the second case study illustrates the precision and recall metrics, the third case study is used as support for the validation, and the description of the two last case studies is intentionally shortened.

[3]These examples have been modelled by graduate students of the Advanced Software Engineering course at the University of L'Aquila as part of their homeworks.

5.1. Case study 1: Bus on Air

The case study has been selected from a Business-Plan Competition, and it is a system called "Bus on Air" (BoA) [59]. It is aimed at developing a set of services for public transportation targeting both the end-users (passengers) and the suppliers (transportation agencies). In the following, we concentrate on the *receiveBestPath* service[4], i.e., the potential passenger arrives at the bus stop and she/he has access to information on the best path to reach a destination, such as lines that cover a path, how long to wait, etc.

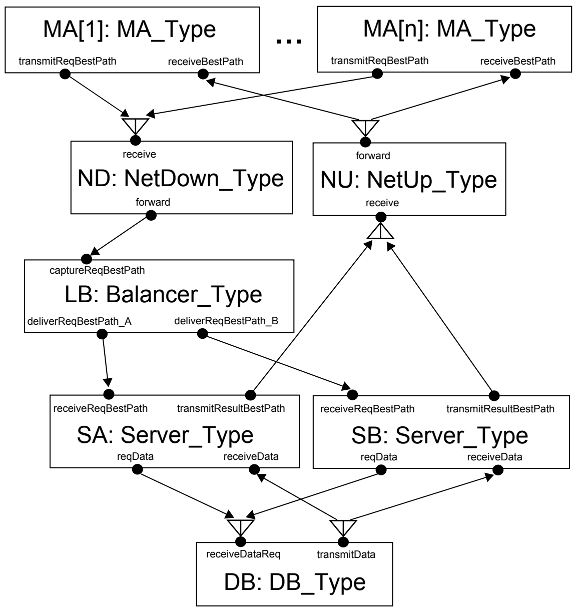


Figure 8: BoA - Flow Graph representation of the software architecture.

Modeling. Figure 8 shows a graphical representation of the BoA software architecture. Each request of one of the n passengers, i.e., mobile application instances (MA: MA_Type), flows through the download network (ND: NetDown_Type) and is forwarded to a load balancer (LB: Balancer_Type). Requests are distributed between two servers (SA: Server_Type, SB: Server_-Type) that retrieve data from a database (DB: DB_Type) and send data to

[4]The complete Æmilia architectural specification of the BoA system (BoA.aem) along with its performance specification (BoA.rew) and the performance results (BoA.val) have been reported in [54].

the upload network (NU: NetUp_Type). This latter network finally forwards the response to the mobile applications.

Analysis. Table 4 summarizes the performance analysis of the BoA software architecture. The first column reports the performance requirements, and the second column the corresponding predicted value, as obtained from the analysis of the BoA specification with the TwoTowers tool [24].

The performance requirements we consider in our experimentation are: the utilization of the *DB* must be lower than 0.6; the throughput of *receiveBestPath* must be greater than 200 requests/second; the throughput of *deliverReqBestPath_A* and *deliverReqBestPath_B* services must be greater than 100 requests/second, the response time of *receiveBestPath* must be lower than 2 seconds. All these requirements must be fulfilled under a workload of 250 requests/second, generated by an estimated total number of 2,000 end-users. Numerical values of performance requirements represent the goals to be achieved by architects for the software system under development and such values can be modified according to end-users expectations. Table 4 shows that all performance indices do not fulfill the required ones, hence we apply our approach to refactor the BoA software architecture by detecting and solving performance antipatterns.

Performance Requirements	Performance Analysis *BoA*
U(DB) < 0.6	0.99
Th($receiveBestPath$) $>$ 200 reqs/sec	36.58 reqs/sec
Th($deliverReqBestPath\_A$) $>$ 100 reqs/sec	24.39 reqs/sec
Th($deliverReqBestPath\_B$) $>$ 100 reqs/sec	12.19 reqs/sec
RT($receiveBestPath$) $<$ 2 sec	2.73 sec

Table 4: Analysis of the performance requirements in the BoA software architecture.

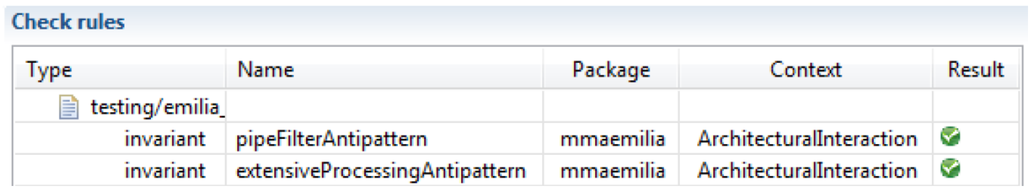
Antipattern-based Refactoring. The Æmilia architectural specification of the BoA system is automatically transformed into an Æmilia model and the performance results are manually annotated, thus to get an Annotated Æmilia model (that has been reported in [54]). This latter model is given as input to our detection tool to look for performance antipattern occurrences.

Detection rules must be parameterized, hence Table 5 reports the threshold values used for our case study. For example, for the *Pipe and Filter Architectures* antipattern two threshold values (see Listing 8) must be evaluated: (i) *serviceThLB* denotes the lower bound for the service throughput, and it is 200 (see Table 5) as stated in the requirements; (ii) *opResDemUB* denotes the upper bound for the operation resource demand, and it is the average value for the resource demand of all operations in the BoA system, i.e. 12682 (see Table 5).

Performance Antipattern	Threshold	Value
Pipe and Filter Architectures	$serviceThLB$ $opResDemUB$	200 12682
Extensive Processing	$respTimeUB$ $opResDemLB$ $opResDemUB$	4 11000000 19000000
Traffic Jam	$initInterval$ $endInterval$ $sizeInterval$ $respTimeGapUB$	0 23000 1000 2
...	...	...

Table 5: BoA - threshold numerical values.

The detection tool gives as output the occurrence of three antipattern occurrences: Pipe and Filter Architectures (*P&F*), Extensive Processing (*EP*), and Traffic Jam (*TJ*). In Figure 9, we can see the wizard showing the result of the *Snapshot* antipatterns' detection. The $\surd$ symbols on the result column indicate that occurrences of the *P&F* and *EP* antipatterns have been detected.



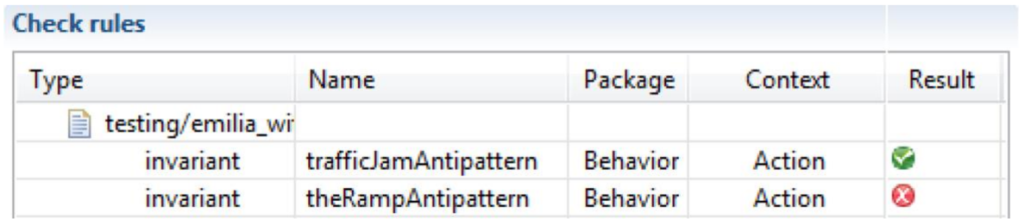
Check rules

Type	Name	Package	Context	Result
testing/emilia_				
invariant	pipeFilterAntipattern	mmaemilia	ArchitecturalInteraction	✓
invariant	extensiveProcessingAntipattern	mmaemilia	ArchitecturalInteraction	✓

Figure 9: Screenshot of the wizard: detection result for Snapshot antipatterns.

In particular, the *P&F* antipattern has been detected in the *receiveBestPath* service where the DB architectural element occurrence represents the slowest filter; the *EP* antipattern has been detected in the *receiveBestPath* service where the LB architectural element instance has two operations (*deliverReqBestPath_A*, *deliverReqBestPath_B*) with an unbalanced resource demand.

Figure 11 illustrates how the *TJ* antipattern has been detected. On the x-axis the simulation time is reported and on the y-axis the response time of



Check rules

Type	Name	Package	Context	Result
testing/emilia_wi				
invariant	trafficJamAntipattern	Behavior	Action	✓
invariant	theRampAntipattern	Behavior	Action	✗

Figure 10: Screenshot of the wizard: detection result for Evolution antipatterns.

the *receiveBestPath* service is depicted. Single points in the graph (denoted with the $\times$ symbol) represent the *observed values*, whereas the line depicts the *average trend*.

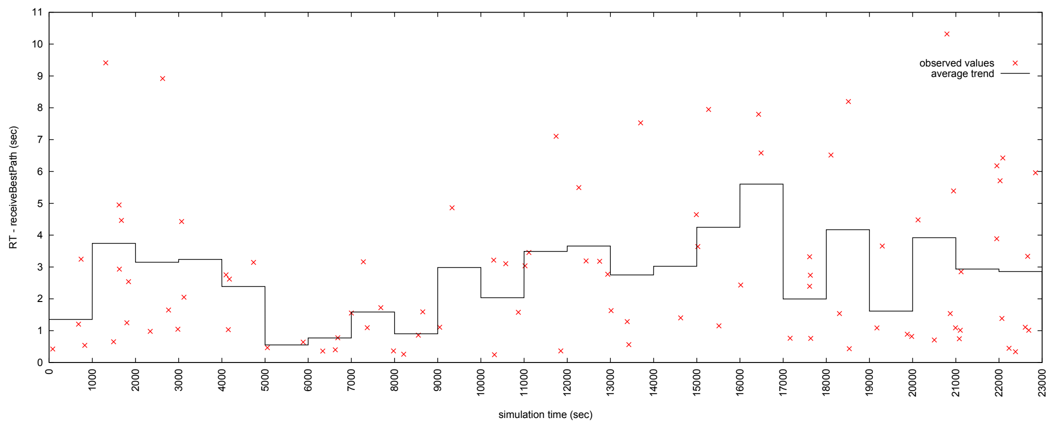
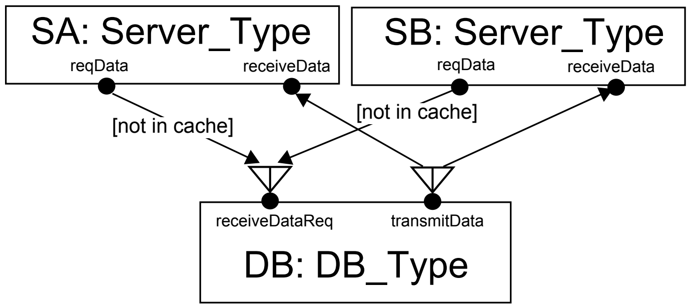


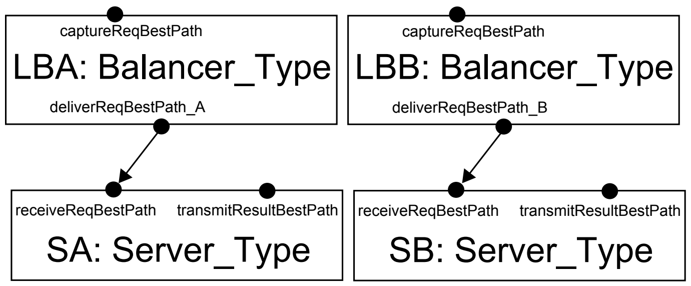
Figure 11: BoA - graphical representation of the *Traffic Jam* antipattern occurrence.

The antipattern has been detected while looking at simulation intervals of 1,000 sec (*sizeInterval*, see Table 5) starting from 0 (*initInterval*, see Table 5) up to 23,000 (*endInterval*, see Table 5): we can notice that there are RT gaps among multiple intervals larger than 2 sec (*respTimeGapUB*, see Table 5). For example, in the $[0, 1000]$ simulation time interval the average RT is 1.35 sec, whereas in the subsequent $[1000, 2000]$ interval the average RT is 3.74 sec, hence the gap among these two intervals is equal to 2.39 sec, i.e., larger than the defined threshold.

The detected antipattern occurrences have been solved on the basis of their solution specifications [12]. Figure 12 illustrates the refactorings that have been applied to the BoA system: the *P&F* antipattern has been solved



(a) *Pipe and Filter Architectures* refactoring.



(b) *Extensive Processing* refactoring.

```
ARCHI_TYPE boa( ...
  const rate server_req_rate:= 70000000,
  const rate server_result_rate:= 85995,
  const rate data_fetch_rate:= 55,
  ... )
```

(c) *Traffic Jam* refactoring.

Figure 12: BoA - representation of the antipattern refactorings.

by introducing a cache that decreases the database accesses, in particular the retrieval of data from servers to the database is conditioned by the presence of such data in the cache (see Figure 12(a)); the *EP* antipattern has been solved by introducing a further instance of balancer (LBA, LBB: Balancer_Type) that privileges the processing of fast requests despite slower ones (see Figure 12(b)); the *TJ* antipattern has been solved by increasing the processing power of the database (data_fetch_rate), thus speeding up the data retrieval (see Figure 12(c)). All these separate refactorings give rise to new Æmilia architectural specifications of the BoA system (BoA-P&F.aem, BoA-EP.aem, BoA-TJ.aem, reported in [54].

Performance Requirements	Performance Analysis			
	BoA	$BoA \setminus \{P\&F\}$	$BoA \setminus \{EP\}$	$BoA \setminus \{TJ\}$
U(DB) < 0.6	0.99	0.31	0.99	0.99
Th($receiveBestPath$) > 200 reqs/sec	36.58 reqs/sec	240.91 reqs/sec	36.58 reqs/sec	54.99 reqs/sec
Th($deliverReqBestPath\_A$) > 100 reqs/sec	24.39 reqs/sec	120.35 reqs/sec	18.29 reqs/sec	36.66 reqs/sec
Th($deliverReqBestPath\_B$) > 100 reqs/sec	12.19 reqs/sec	120.35 reqs/sec	18.29 reqs/sec	18.33 reqs/sec
RT($receiveBestPath$) < 2 sec	2.73 sec	0.41 sec	2.73 sec	1.82 sec

Table 6: Analysis of the performance requirements across different refactorings of the BoA software architecture.

Table 6 summarizes the performance results obtained across different refactorings of the BoA software architecture, after solving the detected antipatterns. The first two columns of Table 6 respectively report the performance requirements and the analysis of the BoA architecture (as already shown in Table 4). All the remaining columns refer to refactored architectures, each identified with *BoA* followed by the name of the removed antipattern.

In Table 6, we can notice that some refactorings do not lead actual benefits to performance indices of interest, e.g., $BoA \setminus \{EP\}$ slightly improves the throughput of the *deliverReqBestPath_B* service (from 12.19 to 18.29), but the throughput of the *deliverReqBestPath_A* service becomes worse (from 24.39 to 18.29). Other refactorings are beneficial to some performance indices of interest, e.g., $BoA \setminus \{TJ\}$ improves the response time of the *receiveBestPath* service (from 2.73 to 1.82) but it is not beneficial for other requirements. Finally, other refactorings may be beneficial to all performance indices of interest, e.g., $BoA \setminus \{P\&F\}$ allows all requirements to be fulfilled.

In this last case, the removal of P&F antipattern allows performance indices to be significantly improved. In Table 7, we report such improvements in percentage. In particular, the highest improvement is observed in the throughput of three key system activities (*receiveReqBestPath*, *deliverReqBestPath_A*, and *deliverReqBestPath_B*) that permitted to reduce the response time of the main BoA service (*receiveBestPath*). Table 7 additionally reports the architectural elements involved in the P&F antipattern: requests for the *receiveBestPath* service are delivered from the `LB: Balancer_Type` to the server instances, which are `SA: Server_Type` and `SB: Server_Type`, and then forwarded to the database instance (`DB: DB_Type`) representing the slowest filter.

Performance Requirements	Performance Analysis		
	BoA	*BoA* $\setminus \{P\&F\}$	Improvement (%)
U(*DB*) < 0.6	0.99	0.31	68.69
Th(*receiveBestPath*) > 200 reqs/sec	36.58 reqs/sec	240.91 reqs/sec	558.58
Th(*deliverReqBestPath_A*) > 100 reqs/sec	24.39 reqs/sec	120.35 reqs/sec	393.44
Th(*deliverReqBestPath_B*) > 100 reqs/sec	12.19 reqs/sec	120.35 reqs/sec	887.28
RT(*receiveBestPath*) < 2 sec	2.73 sec	0.41 sec	84.98
Architectural Elements	LB: Balancer_Type, SA: Server_Type, SB: Server_Type, DB: DB_Type		

Table 7: Performance improvement gained with the antipattern-based refactoring process in BoA case study.

Performance Requirements	Performance Analysis		
	BoA	Bottleneck	Improvement (%)
U(*DB*) < 0.6	0.99	0.97	2.02
Th(*receiveBestPath*) > 200 reqs/sec	36.58 reqs/sec	53.73 reqs/sec	46.88
Th(*deliverReqBestPath_A*) > 100 reqs/sec	24.39 reqs/sec	35.82 reqs/sec	46.86
Th(*deliverReqBestPath_B*) > 100 reqs/sec	12.19 reqs/sec	17.91 reqs/sec	46.92
RT(*receiveBestPath*) < 2 sec	2.73 sec	2.54 sec	6.96
Architectural Element	DB: DB_Type		

Table 8: Performance improvement gained with the bootleneck removal in BoA case study.

Bottleneck-based Refactoring. Bottleneck analysis identifies the database as bottleneck for the BoA system and its mitigation is performed by duplicating the DB architectural element instance into two instances (i.e.,

$DB1$ and $DB2$, both of DB_Type), thus to balance the incoming requests. Table 8 reports the performance results obtained after applying this architectural change aimed at achieving the bottleneck removal. The utilization and the response time requirements slightly improves (due to load balancing of requests), however they are both not fulfilled. Throughput requirements improve of a considerably notable percentage even if their fulfillment is not achieved. Comparing these results with the ones reported in Table 7, we can conclude that for this case study the antipattern-based refactoring process outperforms the bottleneck removal for all the requirements. Table 8 additionally reports the architectural element recognized as bottleneck of the system, which is the database instance (`DB: DB_Type`).

5.2. Case study 2: Vacation Planning System

Vacation Planning System (VPS) is a system for managing trips planned by users. For example, a user starts searching for a location that may be close enough to where she/he lives (e.g., within x miles). Moreover, the user starts searching a nice and comfortable hotel, good restaurants and places to see close to the chosen location. The user may want to book tickets for a couple of museums and reserve a table in a restaurant. After planning the trip, the user starts downloading the plan she/he had created and all the reservations. Before leaving, the user also needs a map and the best itinerary to reach the place.

	Performance Analysis			
Performance Requirements	VPS	$VPS \setminus \{P\&F\}$	$VPS \setminus \{EP\}$	$VPS \setminus \{TJ\}$
U($VacationPlanner$) < 0.5	0.67	0.59	0.49	0.66
Th($getDestinationDetails$) $>$ 0.1 reqs/sec	0.07 reqs/sec	0.03 reqs/sec	0.08 reqs/sec	0.12 reqs/sec
RT($createVacationPlan$) < 5 sec	6.89 sec	3.33 sec	24.37 sec	4.09 sec

Table 9: Analysis of the performance requirements across different refactorings of the VPS software architecture.

Table 9 summarizes the performance results obtained across different refactorings of the VPS software architecture, after solving the detected antipatterns. The first two columns of Table 9 respectively report the performance requirements and the analysis of the VPS architecture. All the remaining columns refer to refactored architectures, each identified with VPS followed by the name of the removed antipattern.

In Table 9, we can notice that $VPS \setminus \{P\&F\}$ slightly improves the utilization of the *VacationPlanner* (from 0.67 to 0.59), and well optimizes

Performance Requirements	Performance Analysis		Improvement (%)
	VPS	$VPS \setminus \{TJ\}$	
U($VacationPlanner$) < 0.5	0.67	0.66	1.49
Th($getDestinationDetails$) > 0.1 reqs/sec	0.07 reqs/sec	0.12 reqs/sec	71.43
RT($createVacationPlan$) < 5 sec	6.89 sec	4.09 sec	40.64
Architectural Elements	VP: VacationPlanner_Type, D: Destination_Type, L: Location_Type, H: Hotel_Type		

Table 10: Performance improvement gained with the antipattern-based refactoring process in VPS case study.

the response time of the *createVacationPlan* (from 6.89 to 3.33), but the throughput of the *getDestinationDetails* service becomes worse (from 0.07 to 0.03). $VPS \setminus \{EP\}$ improves the utilization of the *VacationPlanner* (from 0.67 to 0.49), but it is not beneficial for the response time requirement. $VPS \setminus \{TJ\}$ improves the throughput of the *getDestinationDetails* service (from 0.07 to 0.12), but it is not beneficial for the utilization requirement. For this case study, the antipattern-based refactoring does not help to fulfill all the stated requirements. Further analysis is needed and other iterations of the process are required to successfully get the performance improvement. However, considering the removal of the TJ antipattern, two over three requirements are satisfied and, as showed in Table 10, improvements are obtained for all performance indices. In particular, we can observe that the throughput of *getDestinationDetails* improves of the 71.43% and the response time of *createVacationPlan* shows an improvement of 40.64%. The improvement found for the utilization of *VacationPlanner* (only 1.49%) is insufficient to satisfy the stated performance requirement. Table 10 additionally reports the architectural elements involved in the TJ antipattern. To accomplish the *getDestinationDetails* service, requests are delivered from the `VP: VacationPlanner_Type` instance to the destination (`D: Destination_Type`) instance, thus experiencing a backlog of jobs while forwarding requests to `L: Location_Type` and `H: Hotel_Type` instances that manage location search and hotel booking, respectively.

Bottleneck-based Refactoring. Bottleneck analysis identifies the *VacationPlanner* as bottleneck for the VPS system and its mitigation is performed by splitting the functionalities of the VacationPlanner architectural element instance. In particular, the search of the location is disjointed by the booking of the hotel. This solution implies to concurrently perform these two functionalities: user requests for location search can be executed while other

Performance Requirements	Performance Analysis		Improvement (%)
	VPS	Bottleneck	
U($VacationPlanner$) < 0.5	0.67	0.44	34.32
Th($getDestinationDetails$) $>$ 0.1 reqs/sec	0.07 reqs/sec	0.08 reqs/sec	14.28
RT($createVacationPlan$) < 5 sec	6.89 sec	5.28 sec	23.37
Architectural Element	VP: VacationPlanner_Type		

Table 11: Performance improvement gained with the bottleneck removal in VPS case study.

users are performing the reservation of the hotel. Table 11 reports the results obtained with the bottleneck removal. The utilization improves of a quite remarkable percentage. It fulfills the corresponding requirement, whereas the throughput and response time requirements are not fulfilled. Table 11 additionally reports the architectural element recognized as bottleneck of the system, that is the *VacationPlanner* instance (`VP: VacationPlanner_Type`). Compared to the antipattern-based refactoring process (see Table 10), the two techniques are complementary: bottleneck removal is effective for the utilization requirement only whereas antipattern refactorings are beneficial for throughput and response time requirements. In [38], we explored the synergies between bottleneck analysis and performance antipatterns by executing these two techniques alternatively. We found that their combination may lead to better performance results.

Table 12 reports the TJ antipattern occurrences that are detected while varying threshold values. In particular, with the original thresholds, we monitored system services from 0 ($initInterval$) to $23,000$ sec ($endInterval$) while considering intervals of $1,000$ sec ($sizeInterval$) and with a response time variation fixed to 2 sec ($respTimeGapUB$), we detected one TJ occurrence, i.e., the *getDestinationDetails* service. While decreasing the threshold for the response time variation ($respTimeGapUB$) to the values of $\{1.5, 1, 0.5\}$ sec, we detected further occurrences of TJ antipattern, i.e., *bookHotel*, *searchLocation*, *retrieveDestination* services, respectively.

Table 12 summarizes the performance results obtained across different refactorings of the VPS software architecture, after solving each of the newly detected antipattern. For example, $U(VacationPlanner)$ drops to 0.66 sec while refactoring the original detected antipattern (see Table 9), whereas it drops to $\{0.78, 0.76, 0.67\}$ values while refactoring the newly detected antipatterns. Hence, the recall is increasing but the precision is not affected as none of the detected antipatterns is helpful to fulfill the stated requirement. In this case, the precision is always equal to zero because the original antipattern is not helpful for the corresponding requirement, i.e., U(VacationPlanner), and all other occurrences of the same antipattern (originated while moving the threshold values) are even less helpful than the original one. On the contrary, for the other two requirements, we observe that, while moving antipattern thresholds, the recall increases because a larger set of antipattern occurrences are detected, but the precision decreases. For example, for the response time requirement, the precision is decreasing from 1 to 1/4 because the original antipattern is helpful for the corresponding requirement, whereas all other occurrences have values larger than the stated requirement.

Performance Requirements	TJ antipattern occurrences			
	getDestinationDetails	*book Hotel*	*search Location*	*retrieveDestination*
U($VacationPlanner$) < 0.5	0.66	0.78	0.76	0.67
(Precision, Recall)	(0, 1/4)			
	(0, 2/4)			
	(0, 3/4)			
	(0, 4/4)			
Th($getDestinationDetails$) > 0.1 reqs/sec	0.12	0.09	0.08	0.07
(Precision, Recall)	(1/1, 1/4)			
	(1/2, 2/4)			
	(1/3, 3/4)			
	(1/4, 4/4)			
RT($createVacationPlan$) < 5 sec	4.09	5.45	5.62	5.95
(Precision, Recall)	(1/1, 1/4)			
	(1/2, 2/4)			
	(1/3, 3/4)			
	(1/4, 4/4)			

Table 12: TJ antipattern occurrences to measure precision and recall.

5.3. Case study 3: Line Plus System

Line Plus System (LPS) is a system for managing lines of people in public places (e.g., post offices, public administration offices, shops, etc.) whose main goal is to reduce the waiting time. There are two main actors: (i) Service Provider (SP), i.e., the person/organization that is providing some kind of service to some clients that justifies the presence of a line; (ii) Client, i.e., the person that has to be in line to receive the services provided by the SP. The basic idea behind LPS is that the SP installs the system in its offices, and then clients use a (mobile) application to know in real-time what is the status of the queue in the office they have to go. In this specific scenario, a Wireless Sensor Network (WSN) must be used in the office to get all the possible information about the status of the line. At the same time, the mobile application used by the clients can act as a virtual ticket for keeping the right order of the clients in line, and for notifying each client that his/her turn is coming.

Performance Requirements	Performance Analysis			
	LPS	$LPS \setminus \{P\&F\}$	$LPS \setminus \{EP\}$	$LPS \setminus \{TJ\}$
U(DB) < 0.7	0.84	0.52	0.93	0.78
Th($displayNextUser$) $>$ 0.1 reqs/sec	0.06 reqs/sec	0.13 reqs/sec	0.07 reqs/sec	0.08 reqs/sec
RT($sendQueueStatus$) $<$ 10 sec	32.33 sec	8.72 sec	29.21 sec	23.33 sec

Table 13: Analysis of the performance requirements across different refactorings of the LPS software architecture.

Performance Requirements	Performance Analysis		Improvement (%)
	LPS	$LPS \setminus \{P\&F\}$	
U(DB) < 0.7	0.84	0.52	38.09
Th($displayNextUser$) $>$ 0.1 reqs/sec	0.06 reqs/sec	0.13 reqs/sec	116.67
RT($sendQueueStatus$) $<$ 10 sec	32.33 sec	8.72 sec	73.03
Architectural Elements	D: Display_Type, S: Server_Type, DB: DB_Type		

Table 14: Performance improvement gained with the antipattern-based refactoring process in LPS case study.

Table 13 summarizes the performance results obtained across different refactorings of the LPS software architecture, after solving the detected antipatterns. The first two columns of Table 13 respectively report the performance requirements and the analysis of the LPS architecture. All the remaining columns refer to refactored architectures, each identified with LPS followed by the name of the removed antipattern.

In Table 13, we can notice that $LPS \setminus \{EP\}$ slightly improves the throughput of the *displayNextUser* service (from 0.06 to 0.07) and the response time of the *displayNextUser* service (from 32.33 sec to 29.21 sec), but the utilization of the *database* becomes worse (from 0.84 to 0.93). Also for this system the solution of the Pipe and Filter Architectures antipattern is beneficial for all requirements, in fact, after its removal, the throughput, utilization, and response time requirements are all fulfilled.

Table 14 reports the percentage of performance improvement gained by removing the P&F antipattern. The index that obtains the highest improvement is the throughput of *displayNextUser* with an improvement of 116.67%, followed by the response time of the *sendQueueStatus* reaching an improvement of 73.03%. Table 14 additionally reports the architectural elements involved in the P&F antipattern: requests for the *displayNextUser* service are delivered from `D: Display_Type` to the server instance `S: Server_Type`, and then forwarded to the database instance (`DB: DB_Type`) representing the slowest filter.

Performance Requirements	Performance Analysis		Improvement (%)
	LPS	Bottleneck	
U(*DB*) < 0.7	0.84	0.35	58.33
Th(*displayNextUser*) $>$ 0.1 reqs/sec	0.06 reqs/sec	0.11 reqs/sec	83.33
RT(*sendQueueStatus*) $<$ 10 sec	32.33 sec	7.84 sec	75.75
Architectural Element	DB: DB_Type		

Table 15: Performance improvement gained with the bottleneck removal in LPS case study.

Bottleneck-based Refactoring. Bottleneck analysis identifies the database as bottleneck for the LPS system, and its mitigation is performed by replacing the DB architectural element instance with one performing ten times faster. Table 15 reports the results from the bottleneck removal, where we can see that all requirements are fulfilled. This refactoring shows a quite considerable improvement in all requirements because it consists in the increase of computing power. It is a rather simplicistic refactoring, in fact the software architecture is not modified. Comparing these results with the ones reported in Table 14, both the antipattern-based refactoring process and the bottleneck removal are beneficial, even though the largest percentage of improvement (116.67%) is obtained for the throughput requirement with antipattern refactorings. Table 15 additionally reports the architectural element recognized as bottleneck of the system, that is the database instance (`DB: DB_Type`).

6. Discussion

The approach presented in this paper highlights the complexity of detecting performance antipatterns in ADL-based software architectures. In the following we discuss some key points raised by this work.

Early vs. late performance analysis. There is a trade-off between carrying on performance analysis in the early lifecycle phases, where detected problems are cheaper to fix but the amount of information is limited, and late performance analysis (possibly on running artifacts), where the results are much more accurate but more constraints have been imposed on the structural and behavioral aspects of a software system. Performance antipatterns that occur at the architectural phase are obviously related to architectural aspects and elements, such as components, connectors, and interactions. Moreover, as already mentioned, the syntax and semantics of any ADL affects the capability to detect antipatterns. Hence, although early performance antipattern detection helps to build a better performing software architecture of the system, this does not exclude that more performance antipatterns can manifest themselves later in the lifecycle. This can occur because in the later phases the software artifacts includes more details originated from design decisions. Such details can give rise to performance antipatterns by themselves or by their combination with predefined software characteristics. This is even more obvious if we observe the design and implementation languages, which are obviously richer than ADLs. Hence, non-detectable antipatterns in the architectural phase can certainly emerge in later design phases, when the missing information will be defined. Besides these ones, other (possibly language-specific) performance antipatterns can appear too, because the software evolves during the development process. To tackle this problem, antipattern detection engines for software artifacts of later lifecycle phases should be introduced.

Using Æmilia as reference ADL. The rationale of using Æmilia as reference ADL to catch performance antipatterns relates to its powerful specification of performance measures. Æmilia relies on a semantic model that is a continuous- or discrete-time Markov chain and a set of instant-of-time, stationary/transient performance measures can be specified through state and transition rewards associated to architectural elements. In addition, Æmilia supports security analysis that, through an analyzer, detects violations to security levels. The support provided by Æmilia to security analysis opens to future performance vs. security trade-off analyses in the same notation. Alternative ADLs that could be considered to detect performance antipatterns are ABACUS [18], OSAN [19], EAST-ADL, and AADL [21]. Among them, AADL seems the one that most suitably supports the specification of performance properties and in fact it is possible to perform a latency analysis on flow specifications of components and connections [22]. Such analysis is however limited to worst and best cases, whereas the reward-based mechanism of Æmilia enables a much wider range of performance specifications.

Expressivity of OCL. The choice of OCL language has been driven by the fact that it currently is a standard rule-based validation language. In [25], the authors conducted a study on the expressive power of OCL. From this study, it follows that the expressivity of OCL is higher than first-order logic, because transitive closures over object relations can be expressed in OCL whilst they cannot be expressed in first-order logic. This property implies that we do not lose expressiveness by defining OCL rules for performance antipatterns that were formalized in first-order logic. Furthermore, antipatterns detection rules are defined also in terms of thresholds that represent systems features and they are assigned by software architects usually based on heuristic evaluations and on their expertise. An issue concerning the verification of OCL rules is that the checker does not take into account the quality of data. Indeed, data defined by software architects derive from system monitoring information and from the domain knowledge, thus they may not be accurate. In this context, it would be interesting to consider some fuzzy measures when defining OCL rules. A way to manage fuzziness is provided by an extension of OCL [60] that exploits the concept of fuzzy sets and allows the definition of error margins and data range. Possible alternatives to the use of OCL can be *Schematron* [61] and the *Alloy specification language* [62], although several others exist. Schematron is a rule-based validation language that allows developers to define assertions checking for patterns instances in XML trees. It is expressed in XML and XPath and for this reason it is more suitable for XML trees than for architectural specifications. Alloy is a model specification language based on first-order logic that can be used by developers to create models and check their correctness by means of the Alloy Analyzer tool. It requires that both the model and the validation rules are expressed in the Alloy language. Furthermore, Alloy does not manage fuzziness, as stated in [63], whereas a valid alternative to Schematron is represented by *FuzzyXPath* [64] that has been introduced to provide support to fuzzy XML querying. Differently from these alternative languages, one of the advantages of using OCL is that it straightforwardly applies to EMF-based models.

Threshold values in the specification of antipatterns. As discussed in [14], thresholds cannot be avoided in performance antipatterns definition. The assignment of concrete numerical values to thresholds is a critical task. In this paper, we used existing heuristics [14] to set such values. However, they can be refined if other sources of information are available. In [58], we showed that thresholds multiplicity and estimation accuracy heavily influence both the detection and the refactoring activities. In [65], we assigned probability and effectiveness values to the detected antipattern occurrences and available refactoring actions, respectively, thus to more accurately study the impact of threshold values.

Complexity of performance antipatterns solution. The process of solving performance antipatterns is very complex because it may happen that several occurrences of antipatterns are detected and many architectural alternatives are defined for the solution of each occurrence. In [15], we introduced a ranking methodology to decide which antipatterns must be solved, among the detected ones, to quickly achieve requirements' satisfaction. We have also validated the ranking process as support to the antipatterns solution step. In particular, in [15], we found that the introduction of a ranking methodology greatly benefits the process, because between 45% and 64% of architectural alternatives are discarded without compromising the convergence towards the desired performance improvement. Furthermore, the introduction of a semantic factor refining the score of antipatterns additionally benefits the ranking methodology by reducing the number of architectural alternatives by up to 77%. However, the process of solving antipatterns can not be fully automated because it must be driven by analysts' expertise. In fact some decisions may not be machine-processable due to different reasons, such as legacy constraints, budget limits, etc.

7. Threats to validity

A threat to the internal validity is that our current implementation allows the detection of antipatterns that strictly conform to our interpretation of the literature [12]. Indeed several other feasible interpretations of antipatterns can be provided. This unavoidable gap is an open–issue in this domain and certainly requires a wider investigation to consolidate the antipatterns' definitions. However, different interpretations of antipatterns can be added in our tool by translating them in OCL rules.

A threat to the external validity is that the results have been obtained on a set of sample case studies modelled by graduate students. To increase the representativeness of input models to our approach, more expert software architects should be involved in a wider experimentation.

The results of our experimentation are repeatable because we use open–source programs that can be freely downloaded from the web. Our implementation and all its results are available in [54].

8. Conclusion

In this paper, we have introduced a model-driven approach to detect performance antipatterns in an ADL-based software architecture. To the best of our knowledge, this is the first paper that works on ADL like Æmilia to introduce automation in the investigation of the causes of poor performance and the results that we have reported are promising.

This experience has allowed us to widen the scope of our research that focused on UML-like languages [15, 16], because typical ADL specifications are very different from UML-like models. The major difference that we have found, in this case, is the fact that Æmilia lacks of information about component deployment. As an immediate consequence, several performance antipatterns cannot be detected on an Æmilia architecture. This is not a drawback, it highlights that an Æmilia architecture contains less information than an UML model, hence the performance analysis is carried out at a higher level of abstraction.

The experimentation conducted on three different case studies allowed us to consolidate: (i) the scalability of the approach because the operational steps do not take more than few seconds each, and (ii) the effectiveness of using performance antipatterns in removing performance flaws on software systems specified by means of ADL. In particular, we showed that the solution of performance antipatterns significantly improves the performance indices of the specified systems: two of the three case studies (i.e., BoA and LPS) achieved the fulfillment of performance requirements with the solution of one antipattern, whereas in one case study (i.e., VPS) two of the three performance requirements are satisfied. Hence, the antipattern-based analysis is actually useful to support software architects in the process of selecting design alternatives, because such design differences are quantified on the basis of numerical results. Moreover, the detection of performance antipatterns allows the observation of multiple architectural elements, thus to highlight the ones contributing to bad performance results. On the contrary, bottleneck analysis is a traditionally consolidated technique aimed at identifying a single critical element, thus often hiding the actual sources of performance flaws.

Six main tasks represent our future work agenda, that are: (i) studying the usability of our approach by exposing the developed tool [54] to users with different levels of experience; (ii) experimenting the approach on real systems coming from industrial experiences; (iii) introducing automation in the last activity of the round-trip process, by devising a technique to refactor Æmilia specifications upon antipattern solution; (iv) augumenting the Æmilia metamodel to include antipattern-based specific concepts to extend the set of detectable ones; (v) investigating how the support provided by Æmilia to security analysis opens to performance/security trade-off analyses in the same notation, and (vi) experiencing a similar approach on other ADLs (such as AADL, etc.) to compare their capabilities in this domain.

Acknowledgements

We would like to thank the anonymous reviewers for their constructive feedback that helped us to improve the paper quality.

References

[1] D. C. Petriu, Challenges in integrating the analysis of multiple non-functional properties in model-driven software engineering, in: Proceedings of the Workshop on Challenges in Performance Methods for Software Development, (WOSP-C), 2015, pp. 41–46.

[2] P. Mohan, A. U. Shankar, K. JayaSriDevi, Quality flaws: Issues and challenges in software development, Computer Engineering and Intelligent Systems 3 (12) (2012) 40–48.

[3] A. Jansen, J. Bosch, Software architecture as a set of architectural design decisions, in: Working IEEE / IFIP Conference on Software Architecture (WICSA, 2005, pp. 109–120.

[4] A. Alebrahim, D. Hatebur, M. Heisel, Towards systematic integration of quality requirements into software architecture, in: European Conference on Software Architecture ECSA, 2011, pp. 17–25.

[5] B. Tekinerdogan, H. Sözer, Defining architectural viewpoints for quality concerns, in: European Conference on Software Architecture ECSA, 2011, pp. 26–34.

[6] M. Bernardo, P. Ciancarini, L. Donatiello, Architecting families of software systems with process algebras, ACM Trans. Softw. Eng. Methodol. 11 (2002) 386–426.

[7] V. Cortellessa, A. D. Marco, P. Inverardi, Model-Based Software Performance Analysis, Springer, 2011. doi:10.1007/978-3-642-13621-4. URL `http://dx.doi.org/10.1007/978-3-642-13621-4`

[8] H. Koziolek, Performance evaluation of component-based software systems: A survey, Perform. Eval. 67 (8) (2010) 634–658.

[9] S. Balsamo, M. Bernardo, M. Simeoni, Performance evaluation at the software architecture level, in: Formal Methods for Software Architectures, Springer, 2003, pp. 207–258.

[10] G. Franks, D. Petriu, M. Woodside, J. Xu, P. Tregunno, Layered bottlenecks and their mitigation, in: Third International Conference on Quantitative Evaluation of Systems (QEST), 2006, pp. 103–114.

[11] C. U. Smith, L. G. Williams, Software performance antipatterns for identifying and correcting performance problems, in: International Computer Measurement Group Conference, 2012.

[12] C. U. Smith, L. G. Williams, More new software antipatterns: Even more ways to shoot yourself in the foot, in: Int. CMG Conference, 2003, pp. 717–725.

[13] C. U. Smith, L. G. Williams, Performance and scalability of distributed software architectures: An SPE approach, Scalable Computing: Practice and Experience 3 (4) (2000).

[14] V. Cortellessa, A. Di Marco, C. Trubiani, An approach for modeling and detecting software performance antipatterns based on first-order logics, Software and System Modeling 13 (1) (2014) 391–432.

[15] C. Trubiani, A. Koziolek, V. Cortellessa, R. H. Reussner, Guilt-based handling of software performance antipatterns in palladio architectural models, Journal of Systems and Software 95 (2014) 141–165.

[16] V. Cortellessa, A. Di Marco, R. Eramo, A. Pierantonio, C. Trubiani, Digging into UML models to remove performance antipatterns, in: ICSE Workshop Quovadis, 2010, pp. 9–16.

[17] C. Trubiani, A. Koziolek, Detection and solution of software performance antipatterns in palladio architectural models, in: ICPE, 2011, pp. 19–30.

[18] K. Dunsire, T. O'Neill, M. Denford, J. Leaney, The ABACUS Architectural Approach to Computer-Based System and Enterprise Evolution, in: ECBS, 2005, pp. 62–69.

[19] A. Kamandi, J. Habibi, Toward a New Analyzable Architectural Description Language based on OSAN, in: ICSEA, 2007, p. 20.

[20] ATESST2, EAST-ADL Domain Model Specification (2010).
URL `http://www.east-adl.info`

[21] B. A. Lewis, P. H. Feiler, Multi-dimensional Model Based Engineering Using AADL, in: IEEE International Workshop on Rapid System Prototyping, 2008.

[22] Carnegie mellon software engineering institute (sei), architecture analysis and design language (aadl) - latency analysis.
URL `https://wiki.sei.cmu.edu/aadl/index.php/Latency_Analysis`

[23] V. Cortellessa, M. De Sanctis, A. Di Marco, C. Trubiani, Enabling performance antipatterns to arise from an adl-based software architecture, in: Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA), 2012, pp. 310–314.

[24] M. Bernardo, Twotowers 5.1 user manual (2006).
URL `http://www.sti.uniurb.it/bernardo/twotowers`

[25] L. Mandel, M. V. Cengarle, On the expressive power of ocl, in: FM'99—Formal Methods, Springer, 1999, pp. 854–874.

[26] N. Rozanski, E. Woods, Software systems architecture: working with stakeholders using viewpoints and perspectives, Addison-Wesley, 2012.

[27] R. Kazman, L. Bass, M. Webb, G. Abowd, Saam: A method for analyzing the properties of software architectures, in: International Conference on Software Engineering (ICSE), IEEE Computer Society Press, 1994, pp. 81–90.

[28] L. Dobrica, E. Niemelä, A survey on software architecture analysis methods, IEEE Transactions on Software Engineering 28 (7) (2002) 638–653.

[29] B. Tekinerdogan, Asaam: Aspectual software architecture analysis method, in: Working IEEE/IFIP Conference on Software Architecture (WICSA), IEEE, 2004, pp. 5–14.

[30] R. Kazman, G. Abowd, L. Bass, P. Clements, Scenario-based analysis of software architecture, Software, IEEE 13 (6) (1996) 47–55.

[31] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere, The architecture tradeoff analysis method, in: IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), IEEE, 1998, pp. 68–78.

[32] R. Kazman, J. Asundi, M. Klein, Quantifying the costs and benefits of architectural decisions, in: International Conference on Software Engineering (ICSE), 2001, pp. 297–306.

[33] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, I. Meedeniya, Software architecture optimization methods: A systematic literature review, IEEE Trans. Software Eng. 39 (5) (2013) 658–683.

[34] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, M. Roveri, Safety, Dependability and Performance Analysis of Extended AADL Models, Comput. J. 54 (5) (2011) 754–775.

[35] V. Gaudel, F. Singhoff, A. Plantec, S. Rubini, P. Dissaux, J. Legrand, An Ada design pattern recognition tool for AADL performance analysis, in: ACM SIGAda International Conference on Ada, 2011, pp. 61–68.

[36] W. J. Brown, R. C. Malveau, H. W. McCormick III, T. J. Mowbray, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, 1998.

[37] P. A. Laplante, C. J. Neill, AntiPatterns: Identification, Refactoring and Management, 2005.

[38] C. Trubiani, A. D. Marco, V. Cortellessa, N. Mani, D. C. Petriu, Exploring synergies between bottleneck analysis and performance antipatterns, in: ACM/SPEC International Conference on Performance Engineering (ICPE), 2014, pp. 75–86.

[39] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, P. Flora, Detecting performance anti-patterns for applications developed using object-relational mapping, in: Proceedings of the International Conference on Software Engineering, ACM, 2014, pp. 1001–1012.

[40] V. S. Sharma, S. Anwer, Detecting performance antipatterns before migrating to the cloud, in: IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Vol. 1, IEEE, 2013, pp. 148–151.

[41] B. Dudney, S. Asbury, J. K. Krozak, K. Wittkopf, J2EE Antipatterns, John Wiley and Sons, 2003.

[42] B. Tate, M. Clark, B. Lee, P. Linskey, Bitter EJB, Manning, 2003.

[43] T. Parsons, J. Murphy, Detecting performance antipatterns in component based enterprise systems, Journal of Object Technology 7 (3) (2008) 55–90. doi:http://www.jot.fm/issues/issue 2008 03/article1/.

[44] J. Xu, Rule-based automatic software performance diagnosis and improvement, in: WOSP, 2008, pp. 1–12.

[45] A. Koziolek, H. Koziolek, R. Reussner, Peropteryx: automated application of tactics in multi-objective software architecture optimization, in: QoSA, 2011, pp. 33–42.

[46] A. Wert, J. Happe, L. Happe, Supporting swift reaction: automatically uncovering performance problems by systematic experiments, in: International Conference on Software Engineering, ICSE, 2013, pp. 552–561.

[47] N. Moha, Y. Guéhéneuc, L. Duchien, A. L. Meur, DECOR: A method for the specification and detection of code and design smells, IEEE Trans. Software Eng. 36 (1) (2010) 20–36.

[48] F. Palma, J. Dubois, N. Moha, Y. Guéhéneuc, Detection of REST patterns and antipatterns: A heuristics-based approach, in: Service-Oriented Computing - 12th International Conference, ICSOC 2014, Paris, France, November 3-6, 2014. Proceedings, 2014, pp. 230–244.

[49] F. Palma, J. Gonzalez-Huerta, N. Moha, Y. Guéhéneuc, G. Tremblay, Are restful apis well-designed? detection of their linguistic

(anti)patterns, in: Service-Oriented Computing - 13th International Conference, ICSOC 2015, Goa, India, November 16-19, 2015, Proceedings, 2015, pp. 171–187.

[50] M. Bernardo, M. Bravetti, Performance measure sensitive congruences for markovian process algebras, Theor. Comput. Sci. 290 (1) (2003) 117–160.

[51] W. J. Stewart, Introduction to the numerical solution of Markov Chains, Princeton University Press, 1994.

[52] P. Welch, The Statistical Analysis of Simulation Results, Academic Press, 1983.

[53] R. Howard, Dynamic Probabilistic Systems, John Wiley & Sons, 1971.

[54] M. De Sanctis, C. Trubiani, V. Cortellessa, A. Di Marco, M. Flamminj, PANDA-AEmilia open source project.
URL `https://github.com/CatiaTrubiani/panda-aemilia`

[55] C. Trubiani, A model-based framework for software performance feedback, in: MoDELS Workshops, 2010, pp. 19–34.

[56] J. L. C. Izquierdo, J. S. Cuadrado, J. G. Molina, Gra2MoL: A domain specific transformation language for bridging grammarware to modelware in software modernization, in: MODSE Workshop, 2008.

[57] Api for ocl syntax, package org.eclipse.emf.ocl.parser.
URL `http://archive.eclipse.org/modeling/mdt/ocl/javadoc/1.1.0/org/eclipse/emf/ocl/parser/package-summary.html`

[58] D. Arcelli, V. Cortellessa, C. Trubiani, Experimenting the influence of numerical thresholds on model-based detection and refactoring of performance antipatterns, ECEASST 59 (2013).

[59] I. Malavolta, M. Di Marcello, F. Gallo, L. Iovino, S. Pace, Bus on Air, Business-Plan Competition (2010).
URL `http://www.busonair.eu`

[60] D. Troegner, Combination of fuzzy sets with the object constraint language (OCL), in: Informatik 2010: Service Science - Neue Perspektiven

für die Informatik, Beiträge der 40. Jahrestagung der Gesellschaft für Informatik e.V. (GI), Band 2, 27.09. - 1.10.2010, Leipzig, Deutschland, 2010, pp. 705–710.

[61] R. Jelliffe, The Schematron Assertion Language 1.6, Academia Sinica Computing Centre (2002).
URL `http://xml.ascc.net/resource/schematron`

[62] D. Jackson, Alloy: a lightweight object modelling notation, ACM Trans. Softw. Eng. Methodol. 11 (2) (2002) 256–290.

[63] R. Hähnle, Many-valued logic, partiality, and abstraction in formal specification languages, Logic Journal of the IGPL 13 (4) (2005) 415–433.

[64] E. Damiani, S. Marrara, G. Pasi, Fuzzyxpath: Using fuzzy logic an IR features to approximately query XML documents, in: Foundations of Fuzzy Logic and Soft Computing, 12th International Fuzzy Systems Association World Congress, IFSA 2007, Cancun, Mexico, June 18-21, 2007, Proceedings, 2007, pp. 199–208.

[65] D. Arcelli, V. Cortellessa, C. Trubiani, Performance-based software model refactoring in fuzzy contexts, in: Fundamental Approaches to Software Engineering FASE, 2015, pp. 149–164.