ABSTRACT

Identifying performance problems is critical in the software design, mostly because the results of performance analysis (i.e., mean values, variances, and probability distributions) are difficult to be interpreted for providing feedback to software designers. Performance antipatterns support the interpretation of performance analysis results and help to fill the gap between numbers and design alternatives.

In this paper, we present a model-driven framework that enables an early detection of performance antipatterns, i.e., without generating performance models. Specific design features (e.g., the number of sent messages) are monitored while simulating the specified software model, in order to point out the model elements that most likely contribute for performance flaws. To this end, we propose to use fUML models instrumented with a reusable library that provides data structures (as Classes) and algorithms (as Activities) to detect performance antipatterns while simulating the fUML model itself. A case study is provided to show our framework at work, its current capabilities and future challenges.

Categories and Subject Descriptors
C.4 [Performance of Systems]: Design studies, Modeling techniques, Performance attributes; D.2.8 [Software Engineering]: Metrics—Performance measures

Keywords
Design Feedback; Performance Antipatterns; Foundational UML

1. INTRODUCTION
In order to evaluate the performance of software systems, it is important to introduce approaches that work in the early phases of the software life-cycle, even before the code is developed. In fact, if performance requirements are not met, there may be negative consequences on significant parts of the project.

Some work has been done in recent years to tackle the problem of automatically interpreting model-based performance analysis results and translating them into design feedback [1, 2, 3]. In this context, software models are typical artifacts involved in the interpretation, and new design(s) alternatives are carried out, as showing better performance. To this end, performance antipatterns [4] provide a valid support, as they characterize well-known bad design practices that lead to software products suffering by poor performance, and they include solutions that let the software architects devise new design(s) showing better performance.

In this paper, we adopt Foundational UML (fUML) [5] to design, from scratch, a model-based framework, namely Performance Antipatterns Modeling and Detection (PAMD), including a reusable library that enables the detection of performance antipatterns through the simulation of the fUML model itself.

The choice of fUML as software modeling notation has been supported in a recent survey on industrial needs on architectural languages [6], where the authors document the still growing and predominant adoption of UML but also the need for more powerful analysis functionalities and tool support for UML-based approaches. Furthermore, the need for a precise UML semantics is also stated in [7].

The benefit of PAMD is twofold. On one hand, it avoids to introduce model transformation approaches that produce different modeling notations and related technological spaces by integrating algorithms and results directly within the modeling language used in systems development, as suggested by France et al. [8]. On the other hand, it embeds a reusable model library that can be directly exploited for analysis purposes.

To the best of our knowledge, PAMD represents the first attempt to detect antipatterns by capturing the activities performed during the simulation of the software model.

The rest of the paper is organized as follows. Section 2 introduces an illustrative example used throughout the paper to show PAMD at work. Section 3 illustrates our framework and the performance antipattern model library, and Section 4 discusses challenges concerning of our approach.

1fUML is a standard recently introduced by OMG. It defines the operational semantics of a strict UML subset: structural and behavioral specifications are modelled by Classes and Activities, respectively.
Finally, Section 5 reviews related works and Section 6 concludes the paper outlining future research directions.

2. ILLUSTRATIVE EXAMPLE

In this section, we provide an illustrative example that we use through the rest of the paper. The example represents an enhanced version of the one provided by Smith and Williams [3] to describe the God class problem, also known as Blob performance antipattern since its presence in a design has negative impact on performance. In particular, we refer to a Multi-Valve System (MVS), i.e., an industrial process control application composed by controllers which, in turn, manage the opening and closing of several valves.

Figure 1 shows the initial design of the considered MVS. A Workload Generator, i.e., WG, originates the workload of the application, which is equally distributed (see probabilities \( p_1 = p_2 = 0.5 \)) between the two controllers, i.e., \( C_1 \) and \( C_2 \). The selected controller has the task to change the status of their own valves, i.e., \( V_1, ..., V_8 \) in case of \( C_1 \), \( V_4, ..., V_8 \) in case of \( C_2 \).

We assume that \( C_1 \) and \( C_2 \) implementations are different. In particular: \( C_1 \) only sends a \( \text{switchValue()} \) signal to its valves, resulting in 1 message per valve; \( C_2 \) first requests the status of its valves, then it checks their status, and finally it sends an \( \text{open}() \) or \( \text{close}() \) signal to them, accordingly to their current status, resulting in 2 messages per valve (see problem in Figure 1). With this assumption, \( C_2 \) behaves like a Blob. In fact, differently from \( V_1, ..., V_3 \), valves \( V_4, ..., V_8 \) have no intelligence; they simply report their own status and respond to \( \text{open}() \) and \( \text{close}() \) signals. Hence, \( C_2 \) does all of the work, by requesting information from its valves, making decisions, and telling to its valves what to do.

The solution to the Blob performance antipattern (see solution in Figure 1) consists of reducing the number of messages required to perform an operation, by moving the status check from \( C_2 \) to each of its valves. As a result of this solution, the implementations of \( C_2 \) and \( V_4, ..., V_8 \) become identical to the ones of \( C_1 \) and \( V_1, ..., V_3 \), respectively.

3. THE APPROACH

Figure 2 sketches the fUML-driven approach for early detection of performance antipatterns. Artifacts and functionalities are depicted as rectangles and rounded boxes, respectively, whereas dashed arrows connect functionalities and related artifacts with labels detailing their relationships. PAMD is the core of our fUML-driven approach, and details are provided hereafter.

3.1 System Design

The approach starts with an user designing the software system (e.g., the MVS) using fUML, by means of any standard UML modeling tool. Indeed, fUML [5] defines the operational semantics of a strict UML subset that includes Classes, Common Behaviors, Activities, and Actions. Nor heavy (e.g., metamodel changes) neither lightweight extensions (e.g., UML profiles) are required. fUML enables the execution of UML models where structural elements (i.e., features) are Classes with their own Properties, Operations, Associations, and behavioral specifications (e.g., operation body) are modeled through Activities. For example, the right side of Figure 3 shows how to model a valve in fUML, in accordance with the UML metamodel (on the left side).

3.2 Model Simulation

The fUML standard goes along with a Java-based reference implementation of an fUML virtual machine (fUML VM) to simulate fUML models. Free open source and commercial UML modeling tools exist that embed this reference implementation within their modeling environments, like Papyrus [4] and MagicDraw [7]. We adopted the latter, in conjunction with its plug-in Cameo Simulation Toolkit, as modeling and simulation environment (i.e., mS tool in Figure 2).

At simulation time, the fUML VM generates a so-called instance model and ignores the non-executable part (e.g., Sequences, StateMachines, Deployments or Annotations [9]). InstanceSpecifications, Links, and Slots elements are generated within the instance model as counterparts of Classes, Associations and Properties, respectively. Note that the simulation time is regulated by the events triggered in the instance model, on the basis of dynamics specified in the System Model. Figure 1 can be seen as the graphical representation of the MVS instance model, where controllers and valves exchange messages (i.e., operation calls) through...
bidirectional links. In this example, WG triggers the events occurring in the MVS.

The execution of fUML Activities, like the valve’s getter and setter of Figure 4, reads (readStructuralFeature), adds (AddStructuralFeature), deletes, or modifies elements of the instance model.

### 3.3 PAMD Library

User-defined fUML models can be extended with model libraries, whose elements are meant to be reused/shared across different UML models, as it happens for libraries in object-oriented programming languages.

PAMD is an executable model library which relies on the native model simulation capability of fUML to provide additional functionalities. In particular, PAMD provides Metrics Calculation and Performance Antipattern Detection, whose data structures and algorithms are partially shown in the top-most part of Figure 4. PAMD_System and PAMD_Component represent the structural elements of the System Model for which metrics are calculated, once suitably instrumented. The combination of the mēs tool with PAMD provides a fUML-based modeling and simulation environment to detect antipatterns.

Model instrumentation consists in extending the System Model with elements from our PAMD library. The model instrumentation process consists of: i) establishing the proper generalization relationships among System Model’s and PAMD’s Classes, e.g., the generalizations in bold in Figure 4 and ii) extending System Model’s operations with the proper PAMD’s inherited operations, e.g., the switchValve() activity of GenericController, where the msg_sent_increment has been added to calculate the number of messages sent by a controller. PAMD’s inherited operations are invoked to collect metrics at simulation time, which are used to detect performance antipatterns.

### 3.4 Metrics Calculation

Once the System Model has been instrumented, several metrics can be calculated by PAMD by means of the calculateMetrics() operation belonging to the PAMD_System. In the following, we only focus on metrics involved in the Blob definition. With respect to Figure 4 these metrics are:

- message_sent is the number of messages sent by a PAMD_Component.
- message_received is the number of messages received by a PAMD_Component.
- percentage_msg_sent is the percentage of messages sent by a PAMD_Component among all the ones circulating in the whole system.
- percentage_msg_received is the percentage of messages received by a PAMD_Component among all the ones circulating in the whole system.

To calculate the first two metrics, we instrument behaviors of the System Model in order to count the number of CallOperation actions executed at simulation time by each PAMD_Component. For example, the instrumented switchValve() ends with a call to msg_sent_increment() for incrementing the number of messages sent by the controller. In the MVS, we monitor messages sent and received by both Controller and Valve instances, and store such values on the corresponding instance (see message_sent and message_received properties of PAMD_Component).

Note that metrics are updated as far as an event occurs in the instance model.

### 3.5 Performance Antipattern Detection

PAMD is currently able perform an early detection of the Blob antipattern, by monitoring PAMD_Components. Early detection means that we perform detect antipatterns without generating performance models. We consider two cases of Blob, namely sender and receiver: the former case occurs when a PAMD_Component sends an excessive number of messages, whereas the latter case occurs when a PAMD_Component receives an excessive number of messages. With respect to the illustrative example, for the sake of space, we only consider here Blob sender cases, whose occurrences involve MVS controllers. Due to fact that each controller switches the status of all its valves when requested by the WG, we can omit Blob receiver cases, since they occur on valves if and only if the sender case occurs on the corresponding controller. On the contrary, if a controller selectively changed the status of its valves, then Blob receiver cases would not strictly depend from Blob sender cases.

With respect to the instrumented ValveSystem of Figure 4 Controller3V and Controller5V classes are all singletons, i.e., only one instance for each of these classes exists at simulation time. At each simulation step, the ValveSystem instance continuously invokes its activateValues() operation which, in turn, invokes the operations of the same name on the two controller objects, in accordance with the value of a uniformly distributed random variable.

Blob occurrences are determined by the switchValue() Activity of Controller5V. In fact, such Activity is composed by an invocation of the checkStatus() operation, followed by an invocation of open() or close() operations, thus generating 2 messages for each Valve (see the corresponding instrumented activity in Figure 4).

According to Figure 4, the design solution MVS’ consists of: i) moving the logics within the blob dashed box of the switchValue() behavior within a new operation on Valve class, and ii) replacing it with a call to such new operation on Valve instances. The resulting MVS’ will halve the number of messages sent by Controller5V.

Table 1 shows an excerpt of a 40-steps simulation scenario of both MVS designs of our illustrative example, i.e., MVS and MVS’, along with involved metrics and Blob detection results, calculated during each simulation step. $C_1$ and $C_2$ are instances of Controller3V and Controller5V, respectively. At each step $i$ (see the first column), for each design, values of metrics defined in Section 3.4 are reported.

The set of entries of the isBlob columns corresponds to the Detection Results box of Figure 4. Only steps where at least one Blob occurs are shown, i.e., the ones having isBlob equal to 1. Non-consecutive steps are devise by a horizontal line. Note that isBlob is modeled as a boolean UML property of PAMD_Component, hence it is stored as Slot value within the fUML model at simulation time.

Given a simulation step $i$, isBlob equals to 1 if $\% \text{msgs sent}$ is greater or equal to a certain threshold, i.e., 0.75, in our illustrative example. Such value is a threshold parameter of the PAMD_System, namely percentage_msg_sent_UPPER (see Figure 4), and it can be configured by the user.

Two horizontal lines separate steps 1-8 from the subsequent ones. This is because, in those steps, only $C_1$ sends messages in the MVS. However, it is reasonable to wait

---

5We embed the Javascript Math object to model random variables.
Figure 4: Instrumented MVS model.

Table 1: MVS alternative designs Vs. Blob detection.

<table>
<thead>
<tr>
<th>step</th>
<th>C1 message_sent</th>
<th>C1 percentage_msg_sent</th>
<th>C2 message_sent</th>
<th>C2 percentage_msg_sent</th>
<th>isBlob C1</th>
<th>C1 message_sent</th>
<th>C1 percentage_msg_sent</th>
<th>C2 message_sent</th>
<th>C2 percentage_msg_sent</th>
<th>isBlob C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>25</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>29</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>33</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>34</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>35</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>36</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>37</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>38</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>39</td>
<td>0</td>
<td>0</td>
<td>70.39</td>
<td>29.41</td>
<td>0</td>
<td>0</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

that both the controllers have sent at least one message to their own valves before starting to detect Blob occurrences. As a consequence, steps 1-8 represent a warm-up time, i.e., the simulation interval in which detected antipattern occurrences are considered to be false-positives.

Our experimentation found out that MVS design shows $C_1$ as a Blob in steps 11-12, and $C_2$ in steps 21-40. The design alternative MVS' is evaluated, and we notice that $C_1$ is detected as a Blob for a longer simulation interval, i.e., 9-13, whereas $C_2$ is not a Blob anymore. Hence, MVS' results in a more balanced ValveSystem, in terms of messages sent by controllers, and it represents a valid alternative to the initial one, i.e., MVS.

4. CHALLENGES

Our approach highlights the complexity of the identification and removal of performance antipatterns, hence we discuss some key challenges that this work raises.

Validation. The first challenge related to this work is to study, by means of validation, to what extent the adoption of fUML is advantageous/disadvantageous with respect to approaches relying on performance models, e.g. [10]. Basing on the experience from this work, we think that even if the detection may be costly due to the continuous update of system parameters, then it pays off to system designers that get more knowledge and are aware of the actual system flaws and their duration. PAMD may provide a more powerful instrument to designers, since we can identify the system scenarios where performance flaws are more critical than others, through the fUML model simulation. We plan to validate the usability of our approach with respect to others by means of an empirical user survey, thus to study its actual benefits.

Interpretation of simulation results. It is very difficult in this context to interpret detection results and to evaluate the effectiveness of performance antipattern solutions, since no performance analysis is conducted. In fact, as far as different simulation intervals are analysed, it is worth to consider consecutive intervals and to detect antipattern occurrences if a certain number of such consecutive intervals show the same bad practice, i.e., it persists for a while. Concerning antipattern solutions, we cannot compare the improvement of performance indices after removing antipat-
tern occurrences since no performance analysis is conducted. As a consequence, there is no guarantee of performance improvements in advance, as the entire process is based on heuristics evaluations. Solving an antipattern results in a new fUML model that is obtained by applying a set of refactoring actions (suggested by the specification of antipattern solution), but we do not know if the design feedback has been actually beneficial for the system performance. However, basing on the experience from this work, we observed that antipattern occurrences can be delayed in the new model, instead of being removed at all. For example, in our experimentation we found that the solution of antipatterns most likely produces new software models where antipattern occurrences arise while considering a larger number of simulation steps. We plan to rank detected antipatterns by estimating the number of users interacting with the system (i.e., while considering system operational profile) in all the simulation intervals where antipatterns occur. In this way PAMD can be integrated with solution strategies aimed at delaying antipatterns especially in simulation intervals where there is a high number of incoming requests.

Parameter tuning. While dealing with the simulation of a software model there are several parameters that can be tuned. For example, the warm-up time (see Section 3.5) can be varied, and it is interesting to detect all antipattern occurrences across the different variations. Furthermore, PAMD currently calculates the metrics of interest (e.g. % of sent messages) while considering one simulation interval. However, when quantifying the metrics several strategies may be devised: (i) fix a simulation interval \((a, b)\), i.e., consider the simulation steps from \(a\) to \(b\) for metrics calculation, as the designer needs a feedback in that specific interval only; (ii) fix a duration for a certain number of simulation steps, i.e., consider \(n\) steps of duration \(d\), as the system requires observation windows of different sizes. We plan to extend PAMD in order to support these strategies in the near future.

In addition to simulation parameters, antipattern specifications build on a set of thresholds referring to either performance indices (e.g., a device utilization) or design features (e.g., number of messages sent by a software component). PAMD can analyze the initial fUML model with different threshold values, hence we plan to quantify their influence using precision and recall metrics, similarly to the methodology presented in [11].

Extensibility of PAMD library. PAMD library currently contains a subset of the functions that have been defined in our logic-based specification of performance antipatterns [12]. Other complex classes and/or activities may be added to calculate different metrics, e.g., the number of created and destroyed instances at simulation time. In addition, there might be the need to introduce algorithms that calculate performance indices (e.g., response time and utilization), as done with the MOSES (MOdeling Software and platform archiEcutre in UML for Simulation-based performance analYsis) model library [23]. Similarly to PAMD, MOSES provides Classes and Activities which enable the calculation of performance indices of the modeled system during the fUML model simulation, i.e., without the need of model transformation to external analysis notations (e.g., queuing networks). However, the current drawback is that, by using MOSES, we experienced scalability issues of the fUML VM in managing large workloads. For this reason, we plan to face this challenge in the near future.

5. RELATED WORK

In literature, several approaches have been introduced to specify and detect code smells and antipatterns [14, 15, 16]. They range from manual approaches, based on inspection techniques [17], to metric-based heuristics [18, 19], using rules and thresholds on various metrics [20] or Bayesian belief networks [21].

However, few model-based approaches prevent performance problems early in the software life-cycle. In [22], we compared the approaches working either on software or performance sides. On the software side, in [23], meta-heuristic search techniques are used for improving different non-functional properties of 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. On the performance side, in [2], performance problems are identified before the implementation of a software system, but they are based on bottlenecks (e.g., the “One-Lane Bridge” antipattern) and long paths only. The main limitation of such approach is that it only applies to Layered Queueing Network performance models, hence its portability to other notations is yet to be proven and it may be quite complex.

Differently from literature, our approach represents the first attempt to detect performance problems during the simulation of a software model. In fact, this work is part of an ongoing, broader research effort (9, 13, 24) that aims at integrating fUML and non-functional analysis methodologies by providing ad hoc model libraries.

The extension of simulation/analysis capabilities of fUML can be realized in three different ways: i) by providing fUML model libraries, as we did in this work and in [13, 24]; ii) by modifying the fUML VM [5], and iii) by translating approaches to external notations and tools, as done in [9]. In [13, 24] we explicitly avoid translational approaches by embedding the analysis algorithms within the fUML through executable model libraries. In [21] we proposed the Agilla Modeling Framework (AMF), i.e., an executable fUML model library to conveniently design agent-based software applications for WSN.

Regarding the second approach, i.e., modifying the fUML VM, [25] and [20] propose to redesign the fUML VM to support testing and debugging [25] as well as concurrency, synchronization, and scheduling capabilities [20]. The main drawback of these approaches is that they break the current fUML standard semantics. As a consequence, they may potentially affect the execution of any existing library.

Finally, some translational approaches exist that transform fUML models into executable code [27] and model execution traces. In [9] we devised an Eclipse-based translational approach that combines fUML with profiles for post-simulation performance analysis of fUML model execution traces. In [28], the same kind of model execution traces can be used to debug UML Activities. Indeed, even though translational approaches have the advantage that existing analysis techniques and tools for the target language can be directly exploited, a major drawback is that an additional level of indirection and complexity is inevitable. Implementing the translation of models from the source language (e.g., UML) into models of the target language used for the analysis requires a deep knowledge of the semantics and the meta-models of both source and target languages as well as of model transformation techniques.
6. CONCLUSION

In this paper we presented a model-driven framework that enables an early detection of performance antipatterns, i.e., without generating performance models. We proposed to use fUML models instrumented with a reusable library that monitors design features and enables the detection of antipatterns while simulating the fUML model itself.

As consequence, the interpretation of model simulation results makes both antipattern detection and solution challenging activities: many aspects need to be investigated while moving at model simulation time. As future work, we aim to consider more complex examples in different domains, thus to assess the usefulness of our approach.

7. REFERENCES