2010 15th IEEE International Conference on Engineering of Complex Computer Systems

Performance Antipatterns as Logical Predicates

Vittorio Cortellessa, Antinisca Di Marco, Catia Trubiani
Dipartimento di Informatica, Università dell'Aquila
Via Vetoio, 67010 Coppito (AQ)
L'Aquila, Italy
{*vittorio.cortellessa, antinisca.dimarco, catia.trubiani*}*@univaq.it*

***Abstract*—The problem of interpreting the results of performance analysis is quite critical in the software performance domain. Mean values, variances, probability distributions are hard to interpret for providing feedback to software architects. Instead, what architects expect are solutions to performance problems, possibly in the form of architectural alternatives (e.g. split a software component in two components and re-deploy one of them). In a software performance engineering approach this path from analysis results to software alternatives still lacks of automation and is based on the skills and experience of analysts. In this paper we propose an automated approach for the performance feedback generation process based on performance antipatterns. To this aim, we model performance antipatterns as logical predicates and we provide a java engine, based on such predicates, that is able to detect performance antipatterns in an XML representation of the software system. Finally, we show the approach at work on a simple case study.**

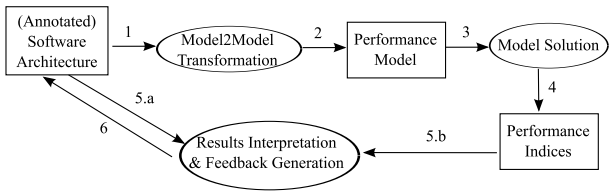
***Keywords*-Software Performance Engineering, Performance Analysis, Antipatterns.**

I. Introduction

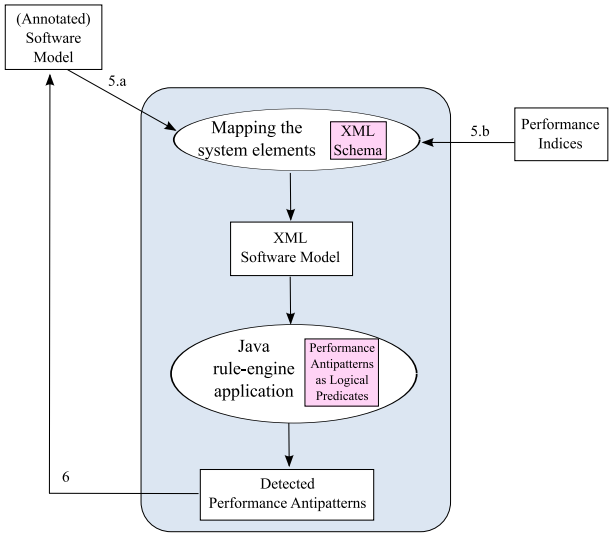
Since more than a decade the problem of modeling and analyzing software performance from the beginning of the life cycle has been tackled with a new type of approaches. The need of automation in the generation of performance models from software artifacts has gained a core role in the whole domain, because automation emerged as a key factor to overcome traditional problems in the field that are: short time to market, and too specific skills required to build trustable models.

Figure 1(a) schematically represents the typical steps of a complete performance modeling and analysis process. In the figure rounded boxes represent operational steps whereas square boxes represent input/output data. Arrows numbered from 1 through 4 in Figure 1(a) represent the typical forward path from an (annotated) software model through the production of performance indices of interest [1].

While in this path quite well-founded approaches have been introduced for inducing automation in all steps [2], [3], there is a clear lack of automation in the backward path that shall bring the analysis results back to the software model.



(a) The overall process.



(b) *Antipattern-based Results Interpretation & Feedback Generation.*

Figure 1. Automated software performance modeling and analysis.

The backward path is represented in the figure as a macro-step of result interpretation and feedback generation that we call *core step* in the following. In this step the performance indices obtained from the model solution, that are typically represented by average values and/or distribution functions, have to be interpreted in order to detect, if any, performance problems. Once performance problems have been detected (with a certain accuracy) somewhere in the model, solutions have to be applied to remove those problems[2]. Typical solutions consist in architectural alternatives, namely

[1]We do not detail this part of the process, as the focus of this paper is not on the forward path. However, readers interested to this path can refer to [1].

[2]Note that this task very closely corresponds to the work of a physician: observing a sick patient (the model), studying the symptoms (some bad values of performance indices), making a diagnosis (an architectural problem), prescribing a therapy (problem treatment).

978-0-7695-4015-3/10 $26.00 © 2010 IEEE
DOI 10.1109/ICECCS.2010.24

feedback, that modify the original software architecture to achieve better performance. Obviously, if all performance requirements are satisfied then the feedback will suggest no changes to the software architecture.

In Figure 1(a), the (annotated) architectural model (label 5.a) and the performance indices (label 5.b) are inputs to the core step that searches problems in the model.

This type of search may be quite complex and needs to be smartly driven towards the problematic areas of the model. The complexity of this step stems from several factors: (i) performance indices are basically numbers associated to model entities and often they have to be compared each other to let problems emerge; (ii) performance indices can be estimated at different levels of granularity and, as it is unrealistic to keep under control all indices at all levels of abstraction, incomplete information often results from the model evaluation; (iii) architectural models can be quite complex, they involve different characteristics of a software system (such as static structure, dynamic behavior, etc.), and performance problems sometimes appear only if those characteristics are cross-checked.

Therefore the need of guidance in this searching process is well clear. Strategies to drive the search can make use of quite different elements that may depend on the adopted model notation, on the application domain, on environmental constraints, etc. In this context we believe that the most promising elements that can drive this search are *performance antipatterns*. Briefly, they represent typical patterns that, if occurring in a model, may induce performance problems. An antipattern definition includes: the pattern specification in terms of model elements (i.e. the problem), the actions to take in order to solve the problem (i.e. the solution). Hence it may be beneficial to steer the searching process by looking for common performance problems, like antipatterns, that have well-known solutions to be adopted.

The backward path in Figure 1 proceeds with providing feedback to the architectural model (label 6) in the form of alternative architectural solution(s) that removes the original performance problems.

Goal of this paper is to provide an implementation of the core step based on performance antipatterns, as described in Figure 1(b).

Performance antipatterns have only been defined, up to now, in natural language [4], hence to automate their detection we formally define the *Performance antipatterns as logical predicates* (see Section IV). Such predicates define conditions on specific system elements (e.g., number of interactions among components, resource throughput) that we organize in an *XML Schema* introduced in Section III.

From the annotated software architecture (label 5.a) and the performance indices (label 5.b), the *Mapping the system elements* step generates an *XML Software Model* conforming to the XML schema and containing all and only the specific software system information needed to detect performance antipatterns.

The *java rule-engine application* is the operational counterpart of the antipatterns declarative definitions as logical predicates. It takes as input the XML representation of the software system (see Section V) and gives in output a list of *detected performance antipatterns* (i.e., the description of the detected performance problems as well as their solutions). Such list contains the feedback, i.e., the alternative architectural solutions, suggested to software architects in the backward path (label 6).

It is worth to notice that the formalization of performance antipatterns we propose in this work is not coupled with the rule-engine. In this paper the engine is implemented in java but other implementations can be introduced on the basis of the logical predicates here defined.

Being the process unavoidably based on heuristic evaluations and decisions, there is no guarantee that the architectural feedback actually solves the problems. Hence the forward path has to be taken again, starting from the updated architectural model and ending up with new performance indices that can confirm the performance problems have been actually removed. Only after this validation the software lifecycle can proceed. If the validation is unsuccessful the backward path has to run again and the whole process restarts[3].

The paper is organized as follows: Section II presents performance antipatterns using an informal representation, Section III introduces the system elements (that have been organized in an XML Schema) on which the performance antipatterns apply, Section IV illustrates our approach based on logical predicates to formally represent antipatterns, Section V describes the java implementation of the antipattern detection process based on such predicates, in Section VI we propose an example as a proof of concept of our approach, in Section VII related work is presented, and finally in Section VIII conclusions and future work are provided.

II. Performance antipatterns definition

Patterns are common solutions to problems that occur in many different contexts [5]. They provide general solutions that may be specialized for a given context. Patterns capture expert knowledge about "best practices" in software design in a form that allows knowledge to be reused and applied in the design of many different types of software.

Antipatterns are conceptually similar to patterns in that they document recurring solutions to common design problems [6]. They are known as antipatterns because their use (or misuse) may produce negative consequences. Antipatterns document common mistakes (i.e. the "bad practices") made during software development as well as their solutions: what to avoid and how to solve the problems.

[3]In a perfect analogy with what a (good) physician makes to check if the prescribed therapy works.

	Antipattern		Problem	Solution
Single-value	Unbalanced Processing	Concurrent Processing Systems	Occurs when processing cannot make use of available processors.	Restructure software or change scheduling algorithms to enable concurrent execution.
		"Pipe and Filter" Architectures	Occurs when the slowest filter in a pipe and filter architecture causes the system to have unacceptable throughput.	Break large filters into more stages and combine very small ones to reduce overhead.
		Extensive Processing	Occurs when extensive processing in general impedes overall response time.	Move extensive processing so that it doesn't impede high traffic or more important work.
	Circuitous Treasure Hunt		Occurs when an object must look in several places to find the information that it needs. If a large amount of processing is required for each look (i.e. a database access), performance will suffer.	Refactor the design to provide alternative access paths that do not require a Circuitous Treasure Hunt (or to reduce the cost of each look).
	...		...	...
Multiple-values	Traffic Jam		Occurs when one problem causes a backlog of jobs that produces wide variability in response time which persists long after the problem has disappeared.	Begin by eliminating the original cause of the backlog. If this is not possible, provide sufficient processing power to handle the worst-case load.
	The Ramp		Occurs when processing time increases as the system is used.	Select algorithms or data structures based on maximum size or use algorithms that adapt to the size.
	...		...	...

Table I
PERFORMANCE ANTIPATTERNS: PROBLEM AND SOLUTION [4].

This paper focuses on *Performance Antipatterns* that, as the name suggests, define bad practices that induce performance problems and their solutions. In particular, we are interested on antipatterns that are independent on the notations used to represent architectural and performance models. This apparent restriction allows to work on the most common antipatterns that can occur in any model because they are not related to any specific characteristic of a modeling notation. For a similar reason performance antipatterns of our interest have not to be specific of any application domain.

The main source of performance antipatterns is the work done across years by Smith and Williams [4] that have ultimately defined a number of 14 notation- and domain-independent antipatterns. They describe settings where a sub-optimal performance design choice has been made. Some other papers present instances of the antipatterns that occur throughout different technologies, but they are not as general as the ones defined by Smith and Williams (see Section VII).

In Table I some of the performance antipatterns defined in [4] are listed. Each row of Table I represents a specific antipattern as characterized by three fields (one per column), that are: antipattern name, problem description, solution description.

This list of performance antipatterns defined by Smith and Williams has been here enriched with an additional attribute. As shown in the leftmost part of Table I, we have partitioned antipatterns in two different categories: one collects antipatterns detectable by single values of performance indices (such as mean, max or min values) and they are referred in this paper as *Single-value* Performance Antipatterns; the other category collects those antipatterns requiring the trend (or evolution) of the performance indices during the time (i.e., multiple values) to capture the performance problems induced in the software system. For these antipatterns, the mean, max or min values are not sufficient unless these values are referred to several observation time frames. Due to these characteristics, the performance indices needed to detect such antipatterns must be obtained via system simulation or monitoring. We name these antipatterns *Multiple-values* Performance Antipatterns.

Up today, antipatterns have been either graphically or textually represented through informal language syntaxes. In both cases an antipattern definition includes the problem (i.e. model properties that characterize the antipattern) and the solution (i.e. actions to take for removing the problem).

The core question tackled in this paper is: how can a performance antipattern be represented to be automatically processed? To this aim, we introduce a (notation-independent) representation of performance antipatterns based on logical predicates (see Section IV).

An antipattern problem definition identifies system properties characterizing the antipattern occurrence. Such properties refer to some software and/or hardware characteristics (i.e., the original design plus the performance indices obtained by the analysis). Thus an antipattern occurs if a certain condition (or predicate) on some system model elements is true. After all, such a predicate represents the formal definition of an antipattern.

III. Identifying the system elements of antipatterns

In this section we provide a structured description of the system elements that occur in the definitions of antipatterns [4] (such as software component, hardware utilization, throughput, etc.), which is meant to be the basis for a formal definition of antipatterns.

Since the (annotated) software architectural model contains details that are not relevant for the antipattern definition, as a first step we have filtered the system model elements useful for the antipatterns characterization.

These concepts have been organized in an XML Schema. The choice of XML as representation language comes from its primary nature of interchange format supported by many tools. We intend to define notation-independent antipatterns, and XML is one of the most straightforward practical means to define generic structured data [4]. Besides, XML technologies support the immediate generation of code that can parse XML files compliant with a certain Schema.

Obviously our XML Schema shares many concepts with existing software system modeling languages (such as UML [7] and ADLs [8]). However, it is not meant to be another software modeling language, rather it is oriented to specify the basic elements of performance antipatterns. Moreover, since we only address performance antipatterns that can be defined independently of the notation adopted for software modeling, in our XML schema we use elements that are independent from any particular notation.

We organize the model elements in views, each capturing a different aspect of the system. We consider three different views representing three sources of information: the *Static View* that captures the elements involved in the software system and the static relationships among them (e.g. classes, components, etc.); the *Dynamic View* that represents the interaction that occurs between the elements to provide the system functionalities (e.g. messages); and finally the *Deployment View* that describes the mapping of the software components onto the hardware resources. This organization is similar to the Three-View Model that was introduced in [9] for performance engineering of software systems.

In general, intersections among views exist, for example the elements interacting in the dynamics of a system are also part of the static view. To avoid redundancy and consistency problems, concepts shared by multiple views are defined once in a view, and simply referred in the other ones.

A primary advantage of introducing views is that the Performance Antipattern specification can be partitioned on their basis. As we will discuss in Section IV, the predicate expressing a performance antipattern is the conjunction of sub-predicates, each referring to a different view.

However, to specify an antipattern it might not be necessary information coming from all the three views, because certain antipatterns involve only elements of a single view. Moreover, different views may be available at different phases of the development process. Such view-based specification of an antipattern allows to reason on antipatterns early in the software life cycle, when some views could not yet be defined. The antipattern detection process could act on the basis of the information currently available and incrementally enhanced with different levels of accuracy as more information becomes available.

It is also worth to notice that an antipattern specification is a heuristic formula: even if the system features match the predicate, it might happen that the antipattern does not actually occur in the system. In general, a confidence value should be associated to an antipattern to quantify the probability that the formula occurrence corresponds to the antipattern presence, but this is matter of our future work.

Our XML Schema for performance antipatterns is synthetically shown in Figure 2 [5]: a *System* is composed of three different Views and of the set of services it provides. The *Static* View groups the elements needed to specify structural aspects of the software system; the *Dynamic* View deals with the behavior of the system; and finally the *Deployment* View captures the elements of the deployment configuration.

A *service* has an identifier (*serviceID* attribute) and it is described by static, dynamic and deployment elements: the *serviceStaticView* refers to a subset of the *Static View*, and similarly for the other two views.

In order to better characterize performance antipatterns, system services must be associated to their *serviceDemand* that can be *open* (specified by the *workload*) or *closed* (specified by the *numUsers* and the *thinkingTime*).

The *Static View* is composed by two different subviews: the *componentView* and the *classView*. Such views contain elements to describe the static aspects of the system at different level of details. The former (referring to components) can be provided early in the software life cycle when the architecture of the system is designed, while the latter can be provided in a detailed design phase, when the system is decomposed in classes. The *Dynamic View* is a set of *behavior*s having an identifier (*behaviorID* attribute), and the probability of execution, *behaviorProb*. A *behavior* contains a set of *entities*, *messages* or an *operator*. The *deploymentView* has a set of processing nodes, *procesNodes*, and optionally some *networkLinks* between the *procesNodes*. More details on the XML Schema are given in [10].

IV. Performance antipatterns as predicates

In this Section we formalize the representation of performance antipatterns as logical predicates. For sake of

[4] As mentioned in Section VIII, as a future work we intend to introduce a metamodel-based approach to the antipattern definition.

[5] For sake of space this Section contains the strictly necessary information about the XML Schema. For more details refer to [10].

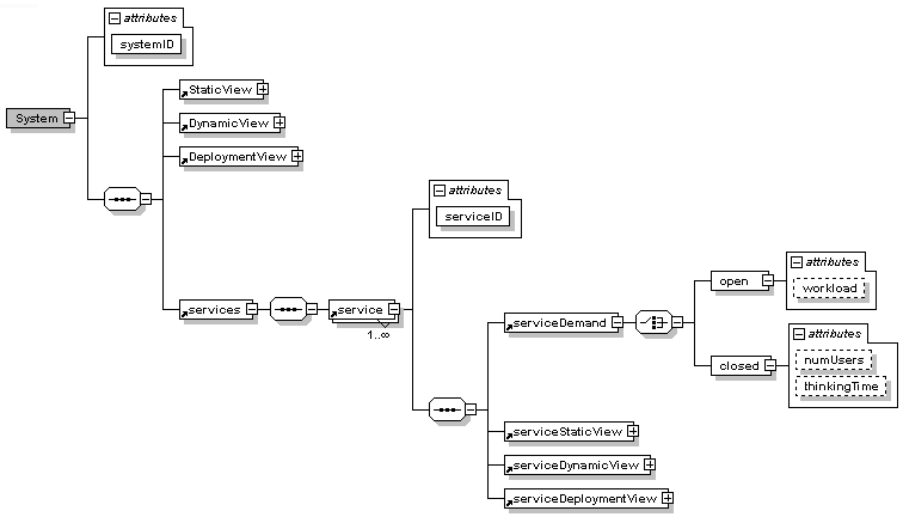


Figure 2. An excerpt of the XML Schema.

illustration, we have selected from Table I the following antipatterns: "Concurrent Processing Systems", "Circuitous Treasure Hunt", and "The Ramp".

One sub-section is dedicated to each antipattern and is organized as follows. From the informal representation of the *problem* (as reported in Table I), a set of *basic predicates* (BP_i) is built, where each BP_i addresses part of the antipattern specification. The basic predicates are first described in a semi-formal natural language and then formalized by means of first-order logics. The operands of basic predicates are those system elements defined in the XML Schema, denoted as underlined words in the following.

In the formalization process we also define supporting *functions* that elaborate certain system elements (represented in the predicates as $F_{funcName}$), and a set of *thresholds* for comparison with observed properties of the software system (represented in the predicates as $Th_{thresholdName}$). Threshold values can be assigned by software architects basing on heuristic evaluations, or they can be obtained by monitoring the system.

Note that in this process we provide our formal interpretation of the informal definitions in Table I. Such interpretation is fully described in the semi-formal description of the basic predicates. Thereafter, the formalization that we propose obviously reflects our interpretation. Different formalizations of antipatterns can be originated by laying on different interpretations of their informal definitions.

A. "Concurrent Processing Systems"

In Figure 3 we report an example of a typical situation for the Concurrent Processing Systems antipattern to occur: there are two types of requests entering the system (i.e. *classA* and *classB*), the requests are assigned either to the processor P_1 or to processor P_2. There is an unbalanced processing since processor P_1 is less used than P_2, and processor P_3 is not employed at all. To improve the system performance we need to ensure that the software is able to use the available resources, without overloading some of them.

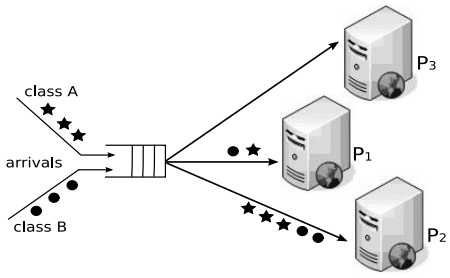


Figure 3. A scenario of the *Concurrent Processing Systems* antipattern.

Concurrent Processing Systems is a Single-value performance antipattern fully explained in [4], whose problem informal definition is (see Table I): *"it occurs when processing cannot make use of available processors"*.

This antipattern is formalized through three basic predicates (i.e. BP_1, BP_2, and BP_3) whose system elements belong to the Deployment View. In the following, we denote with $\mathbb{P}$ the set of the procesNode instances in the system.

BP_1 - There is at least one procesNode instance, P_x, among the procesNode instances in $\mathbb{P}$, having a large queueLength, that is its queueLength is greater than a threshold:

$$F_{queueLength}(P_x) \geq Th_{queue}{}^{6} \quad (1)$$

[6]This threshold value can be estimated, for example, as the average number of all the queue length values, with reference to the entire set of hardware machines in the software system, plus the corresponding variance.

BP_2 - P_x has also a heavy computation. This means that the utilization of some hardware resources in P_x (e.g., CPU, disk,etc.) exceeds a predefined limit. Let us define $F_{maxUtil}$ that is the function giving in output the maximum value of hwUtilization for the hardware resources in P_x. Such function has two input parameters: the processing node, and the type of hardware resource (i.e. 'cpu', 'disk', or 'all' to denote no distinction between them). Since in this case, the important thing is to identify if there exists an overloaded hardware resource, we call the function with the 'all' option. BP_2 can be formalized as:

$$F_{maxUtil}(P_x, all) \geq Th_{hwRes}{}^{7} \quad (2)$$

BP_3- processing nodes are not used in a well-balanced way, namely there is at least another instance of procesNode, P_y, among the procesNode instances in $\mathbb{P}$, whose hwUtilization of the hardware resources is smaller than the P_x one. That is:

$$F_{maxUtil}(P_y, all) < F_{maxUtil}(P_x, all) \quad (3)$$

Hence, the *Concurrent Processing Systems* antipattern occurs when: $\exists P_x, P_y \in \mathbb{P} \mid \boxed{(1) \wedge (2) \wedge (3)}$, where $\mathbb{P}$ represents the set of all the procesNode instances in the software system. The set of ProcesNode instances satisfying such predicate must be pointed out to the designer for a deeper analysis.

B. "Circuitous Treasure Hunt"

Circuitous Treasure Hunt is a Single-value performance antipattern fully discussed in [4], whose problem informal definition (see Table I) can be partitioned in two parts.

The first part, *"it occurs when an object must look in several places to find the information that it needs"*, is formalized through one basic predicate BP_1 whose elements belong to the three Views:

BP_1- There are at least two swResources, swR_x and swR_y, instances of two components or classes such that: *i)* they are both involved in a service S; *ii)* one, the sender (i.e., swR_x), sends an excessive number of messages to the other one, the receiver (namely, swR_y); *ii)* the receiver is a database (as captured by the isDB attribute in the component/class definition of the static view). Let us define the function $F_{numDBmsgs}$ that provides the number of messages sent to the swR_y database by the swR_x in a service S. The basic predicates coming from this analysis are:

$$F_{numDBmsgs}(swR_x, swR_y, S) \geq Th_{DBmsgs}{}^{8} \quad (4)$$

The second part of the antipattern problem definition claims that *"If a large amount of processing is required for each "look" (i.e. a database access), performance will suffer"*.

We formalized this part of the definition with two basic predicates (i.e. BP_2, and BP_3), whose elements belong to the Deployment View:

BP_2- The procesNode P_{swR_y} on which the swResource swR_y is deployed has a heavy computation. That is, the hwUtilization of a hardware resource of the procesNode P_{swR_y} exceeds a certain threshold. For the formalization of this characteristic, we use the $F_{maxUtil}$ function, with the 'all' option, that returns the maximum hwUtilization among the ones of the hardware resources composing the processing node. In formula:

$$F_{maxUtil}(P_{swR_y}, all) \geq Th_{hwRes} \quad (5)$$

BP_3- Since a database access utilizes more disk than cpu, we compare the maximum disk(s) hwUtilization with the one of the cpu(s) of the procesNode P_{swR_y} that can be modeled as:

$$F_{maxUtil}(P_{swR_y}, disk) \geq F_{maxUtil}(P_{swR_y}, cpu) \quad (6)$$

Summarizing, the *Circuitous Treasure Hunt* antipattern occurs when: $\exists swR_x, swR_y \in \text{swR} \mid \boxed{(4) \wedge (5) \wedge (6)}$, where swR represents the set of all the swResources. Here the information to point out to the designer is the set of swResource pairs satisfying the predicate.

C. "The Ramp"

The Ramp is a Multiple-values performance antipattern fully explained in [4], whose problem informal definition is (see Table I): *"it occurs when processing time increases as the system is used"*.

This antipattern is formalized through two basic predicates (i.e. BP_1, BP_2), whose elements belong to the Static and Deployment View:

BP_1- There is at least one instance of operation Op that has an increasing value in the response time along different observation time slots. Let us define the function $F_{RT}(Op, i)$ that returns the mean responseTime of the operation Op observed in the time slot i. We consider the average response time increase of the operation in N consecutive time slots. Then we can write:

[7]This threshold value can be estimated, for example, as the mean value of all the hardware utilization values, with reference to the entire set of hardware resources in the software system, adding the corresponding variance.

[8]This threshold value can be estimated as the average number of database requests, with reference to the entire set of software resources in the software system, plus the corresponding variance.

$$\frac{\sum_{1\leq i\leq N}(F_{RT}(Op,i)-F_{RT}(Op,i-1))}{N} > Th_{rtVar}{}^{9} \quad (7)$$

BP_2- The operation Op shows a decreasing throughput over time. Let us define the function $F_T(Op,i)$ that gives the mean throughput of the operation Op observed in the time slot i. We consider the average throughput decrease in N consecutive time slots. Then we can write:

$$\frac{\sum_{1\leq i\leq N}(F_T(Op,i)-F_T(Op,i+1))}{N} > Th_{tVar}{}^{10} \quad (8)$$

Hence, *The Ramp* antipattern occurs when: $\exists Op \in \mathbb{O} \mid \boxed{(7) \wedge (8)}$, where Op is an instance of operation belonging to $\mathbb{O}$. The set of the operation satisfying such a predicate has to be reported to the designer as feedback.

V. Java Rule-Engine Application

For the sake of validation, we implemented the defined predicates in a java rule-engine application. Such application is designed to parse any XML document compliant with our Schema and produces as output the detected antipatterns.

It uses the Java Application Programming Interface (API) for XML processing: Document Object Model (DOM) [11] is a cross-platform and language-independent convention for representing and interacting with objects in XML documents.

For sake of illustration, we report in Listing 1 the fragment of Java code that implements the boolean function (i.e., *checkCPS_PA*) able to match the three conditions under which the *Concurrent Processing Systems* antipattern occurs (see Section IV-A).

In the function two supporting arrays are defined: the first one (i.e., *nodeQLs*) records the mean queue length for each processing node; the second one (i.e., *nodeHwResUtil*) stores the maximum utilization of the hardware resources again for each processing node. The two arrays play a key role and they are used to be compared with threshold values.

The first condition of *Concurrent Processing Systems* consists in the existence of at least one instance of *procesNode* that has an average queue length larger or equal to a threshold (i.e., Th_{queue}). Th_{queue} is approximated to the average number of all the queue length values, with reference to the entire set of hardware machines in the software system and it is calculated by the function *getTh_queue(doc)*.

The second condition of *Concurrent Processing Systems* consist in the existence of at least one instance (among the ones that satisfy the first condition) of *procesNode* that has a maximum hardware utilization greater or equal to a threshold (i.e., Th_{hwRes}). Th_{hwRes} is approximated to the average number of all the hardware utilization values, with reference to the entire set of hardware resources in the software system. This threshold is calculated by the function *getTh_hwRes(doc)*.

The third condition is that processing nodes are not used in a well-balanced way, so there is at least one instance of *procesNode* (among all system nodes) that has an average maximum utilization consistently lower than the one of the *procesNode* instances that satisfy the first and second conditions. Any of these instances can be targeted to lead a *Concurrent Processing Systems* antipattern. Listing 1 checks all previous conditions and reports whether the antipattern occurs or not.

```
boolean checkCPS_PA(Document doc) {
    boolean PA_cps = false;

    int nodeQLs[] = new int[numProcNodes(doc)];
    float nodeHwResUtil[] = new float[numProcNodes(doc)];

    NodeList listOfProcesNodes = doc.getElementsByTagName("procesNode");

    for (int s = 0; s < listOfProcesNodes.getLength(); s++) {
        float maxUtilCurrentNode = 0;
        Node procesNode = listOfProcesNodes.item(s);
        if (procesNode.getNodeType() == Node.ELEMENT_NODE) {
            Element procesElement = (Element) procesNode;

            NodeList innerListOfHwResources = procesElement.
                  getElementsByTagName("hwResource");
            for (int i = 0; i < innerListOfHwResources.getLength(); i++) {
                Node hwResNode = innerListOfHwResources.item(i);
                float tempUtilCurrentNode = Float.parseFloat(hwResNode.
                      getAttributes().item(1).getNodeValue());
                if (tempUtilCurrentNode > maxUtilCurrentNode) {
                    maxUtilCurrentNode = tempUtilCurrentNode;
                }
            }
            nodeHwResUtil[s] = maxUtilCurrentNode;

            NodeList queueLengthList = procesElement.getElementsByTagName("
                  queueLength");
            Element queueLengthElement = (Element) queueLengthList.item(0);
            nodeQLs[s] = Integer.parseInt(((Node) queueLengthElement.
                  getChildNodes().item(0)).getNodeValue().trim());
        }
    }

    for (int i = 0; i < numProcNodes(doc); i++) {
        if (nodeQLs[i] > getTh_queue(doc)) {
            if (nodeHwResUtil[i] > getTh_hwRes(doc)) {
                for (int j = 0; j < numProcNodes(doc); j++) {
                    if (nodeHwResUtil[i] > nodeHwResUtil[j]) {
                        PA_cps = true;
                    }
                }
            }
        }
    }
    return PA_cps;
}
```

Listing 1. Fragment for the evaluation of the *Concurrent Processing Systems* antipattern.

VI. Case study

In this section we apply the proposed approach of the core step to the Electronic Commerce System (ECS) case study. With this example we demonstrate the feasibility of our approach illustrating how the predicates introduced in Section IV can be used to detect, if any, antipatterns.

ECS is a highly web-based system to manage business data: customers are able to browse catalogs and make selections of items that need to be purchased. At the same time, agents can upload their catalogs, change the prices, the availability of products and so on.

The performance requirement defined for the ECS system is that each hardware resource has not to be used more

[9]This threshold value represents the maximum feasible slope of the response time observed in N consecutive time slots.

[10]This threshold value represents the maximum feasible slope of the throughput observed in N consecutive time slots.

than 80% under the mean workload of 70 requests/second concurrently in execution in the system.

The ECS system could reveal some critical services, such as browsing catalogs and make a purchase. The former one is critical because it is required by a large number of (registered and not registered) customers, whereas the latter one requires several database accesses that can drop the system performance.

We adopt a revised version of Prima-UML methodology [12] as forward path from an (annotated) software model through the production of performance indices of interest. PrimaUML requires modeling the system requirements through an UML Use Case Diagram, modeling the software dynamics through UML Sequence Diagrams, and modeling the software-to-hardware mapping through an UML Deployment Diagram. The use case diagram should be annotated with the system operational profile, the sequence with service demands and message size of each operation, and the deployment with the characteristics of hardware nodes.

In Figure 4, we report an excerpt of the ECS (annotated) Software Model. In the modeling, we use UML 2.0 [7] as modeling language and MARTE Profile [13] to annotate additional information needed to execute performance analysis (such as workload to the system, service demands, hardware characteristics). In particular, the UML component diagram of Figure 4(a) describes the details of the software components and their interconnection through provided/required interfaces, whereas the UML deployment diagram of Figure 4(b) shows the deployment of the software artifacts over the hardware platform. The deployment is annotated with the characteristics of the hardware nodes by means of MARTE stereotypes to specify CPU characteristics (speedFactor and schedPolicy tags) and network delay (blockT tag).

The performance indices for the ECS system are obtained by solving the queueing network model produced applying PrimaUML.

We consider two scenarios: *browseCatalog* (invoked with a probability of 99%) and *makePurchase* (invoked with a probability of 1%). In Table II we report input parameters and output indices of the ECS queueing model: the first column contains the service demand of each service center of the queueing network; the second column shows their corresponding utilization. As it can be noticed, the *libraryNode* has an utilization of 96% that is larger than the required one (i.e., 80%). The requirement is not satisfied thus we apply our approach to detect possible performance antipatterns.

As first step, the approach joins the Architecture Model and the Performance Indices in an XML representation [14] for the ECS system case study. After that, we run the Java rule-engine application that detects the antipatterns collected in Table III.

We denote with ECS_*antipattern* the new ECS system in which the corresponding antipattern solution is applied.

	Service Demand (input parameters)	Utilization (output indices)
webServerNode	2.02 msec	27%
libraryNode	7.05 msec	**96%**
controlNode	3 msec	41%
databaseNode_cpu	15 msec	20%
databaseNode_disk	30 msec	41%

Table II
PERFORMANCE PARAMETERS AND INDICES OF THE ECS SYSTEM.

Antipattern	Problem	Solution
Concurrent Processing Systems	Processing cannot make use of the processor *webServerNode*.	Restructure software or change scheduling algorithms between processors *libraryNode* and *webServerNode*.
Blob	*libraryController* performs most of the work, it generates excessive message traffic.	Refactor the design to keep related data and behavior together. Delegate some work from *libraryController* to *filmLibrary* and *bookLibrary*.
Circuitous Treasure Hunt	A large amount of processing to find information in *database*.	Refactor the design of the *database* to reduce the cost of each access.

Table III
ECS PERFORMANCE ANTIPATTERNS: PROBLEM AND SOLUTION.

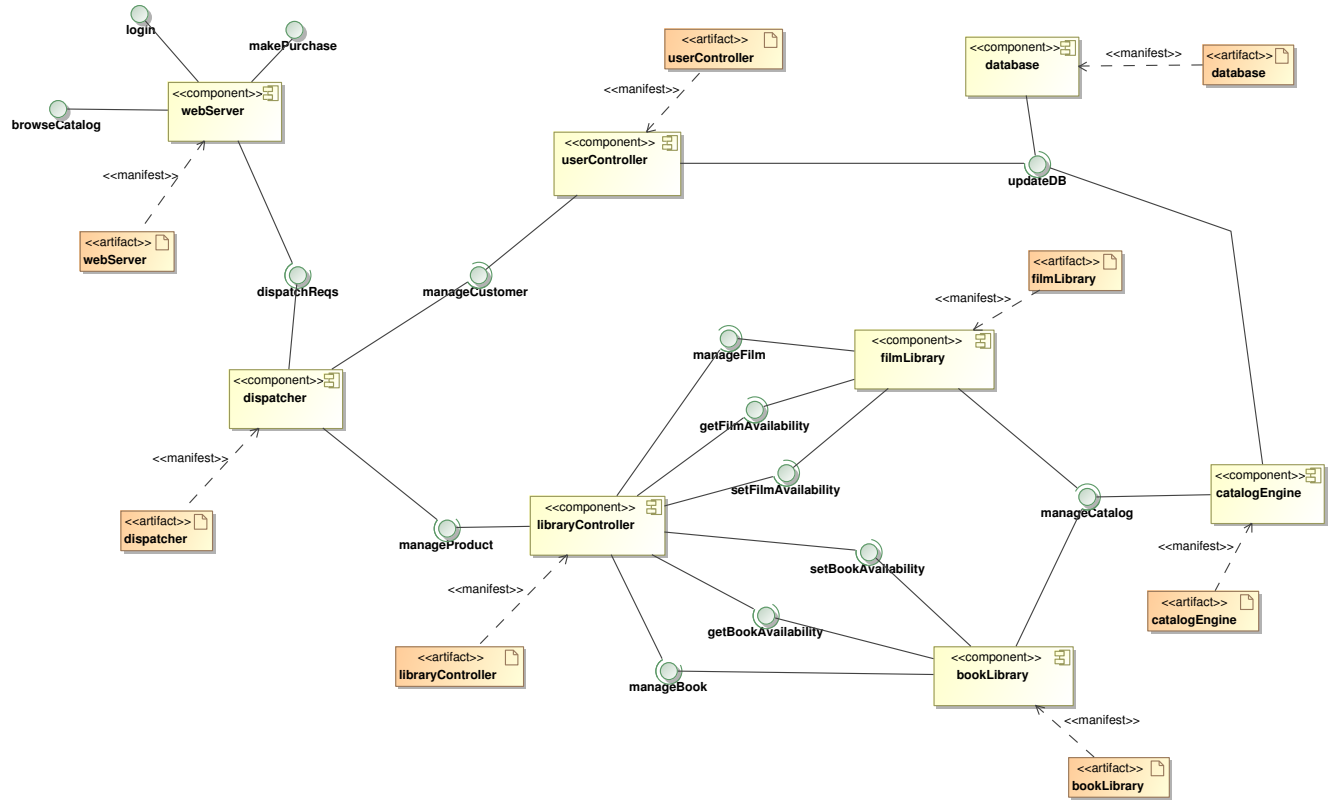
For example, in our experimentation we decide to solve the Concurrent Processing Systems Antipattern, obtaining a new system: ECS_cps. According to the antipattern solution we re-deploy *dispatcher* and *userController* components from *libraryNode* to the *webServerNode*, thus to obtain the ECS_cps system.

We re-apply the PrimaUML methodology. Table IV reports the performance parameters and indices obtained by solving the queueing network model corresponding to the ECS_cps system. Note that the antipattern has provided a relevant information to the designer because its removal consistently improves the utilization of hardware resources and the system requirement (i.e., each hardware utilization lesser than 80%) is not any more violated.

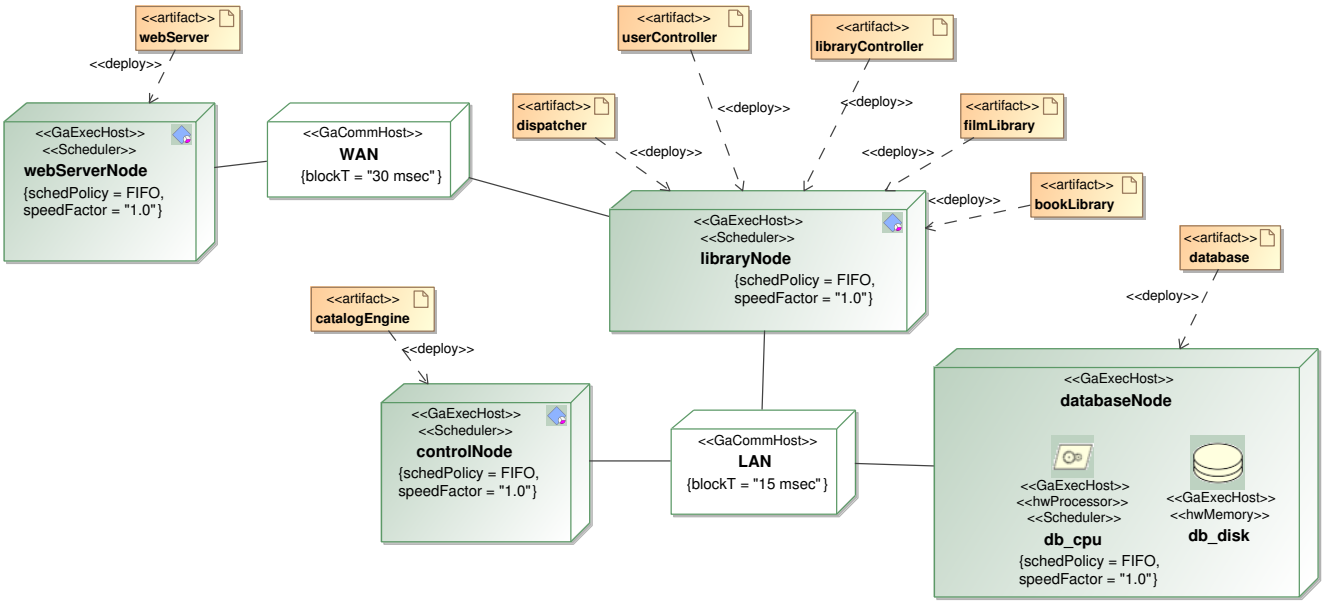
VII. RELATED WORK

The term *Antipattern* appeared for the first time in [6] in contrast to the trend of focus on positive and constructive solutions. Differently from patterns, antipatterns look at the negative features of a software system and describe commonly occurring solutions to problems that generate negative consequences.

Antipatterns have been applied in different domains. For example, in [15] data-flow antipatterns help to discover errors in workflows and are formalized through the CTL*



(a) UML component diagram



(b) UML deployment diagram

Figure 4. An excerpt of the ECS (annotated) Software Architecture Model.

	Service Demand (input parameters)	Utilization (output indices)
webServerNode	4.07 msec	61%
libraryNode	5 msec	75%
controlNode	3 msec	45%
databaseNode_cpu	15 msec	22%
databaseNode_disk	30 msec	45%

Table IV
PERFORMANCE PARAMETERS AND INDICES OF THE ECS_CPS SYSTEM.

temporal logic. As another example, in [16] antipatterns help to discover multi threading problems of java applications and are specified through the LTL temporal logic.

Performance Antipatterns, as the same name suggests, deal with performance issues of the software systems. They have been previously documented and discussed in different works: *technology independent* performance antipatterns have been defined in [4] and they represent the main references in our work; *technology specific* antipatterns have been specified in [17] [18].

Enterprise technologies and EJB antipatterns are analyzed in [19]: antipatterns are represented as a set of rules loaded into the JESS [20] rule engine. The monitoring of the software application leads to reconstruct its run-time design and to obtain JESS facts. The matching between pre-defined rules and application facts is performed in order to carry out the detection of antipatterns.

Another recent work on the software performance diagnosis and improvements is proposed in [21]. Rules able to identify patterns of interaction between resources are defined, and again specified as JESS rules. Performance problems are identified before the implementation of the software system, even if they are based only on bottlenecks (e.g. the "One-Lane Bridge" antipattern) and long paths. Layered resource architectures are considered and the performance analysis is conducted with Layered Queueing Network (LQN) models. Such approach applies only to LQN models, hence its portability to other notations is yet to be proven and it may be quite complex. Our intent is instead to address a much wider set of modeling notations. This is the reason why we propose to represent performance antipatterns with basic constructs such as logical predicates.

Antipattern representation is a more recent research topic, whereas there has already been a significant effort in the area of representing software design *patterns*.

In [22] the authors propose a pattern specification language that is aimed to specify static and dynamic features of design patterns. Each pattern is analyzed from two complementary views: the structural and the behavioral views. In the context of performance antipatterns we need an additional view, as the deployment view is necessary to represent platform properties.

In [23] design patterns are represented by logic predicates through which it is possible to identify roles and subsequently candidates for the patterns. Starting from the UML class diagram of the design pattern, it is possible to identify the role elements of the pattern and their relationship: each role is translated into a logical predicate, and finally the design pattern is a logical predicate that manages the interplay of all the different roles involved in its specification.

In [24] a set of modeling patterns are used to explore the design space of soft real-time systems. Patterns are proposed as parametric templates that can be applied by setting appropriate values according to different deployments. A modeling language formalizes patterns representing design solutions that may have a different impact on the system performance intended as the amount of fulfilled real-time deadlines.

VIII. CONCLUSIONS

This work is a contribution to the automation of the backward path from performance results to architectural feedback. Specifically, we have tackled the problem of using performance antipatterns to provide a first implementation of the *Results Interpretation & Feedback Generation* step.

We have structured the system elements that define performance antipatterns as an XML Schema, which has been conceived as notation-independent. Besides, we have formalized the definition of performance antipatterns as logical predicates whose operands are elements of the Schema. A case study is illustrated to demonstrate the feasibility of the proposed approach.

The formalization of antipatterns proposed in this paper is the result of multiple formulations and checks. It is a first attempt to formally define antipattern and it may be subject to some refinements. However, the presented case study proves the potentiality of the formalization framework for performance antipattern detection.

However, we remark that the formalization we propose reflects our interpretation of the informal definition of the considered antipattern. Different formalizations of antipatterns can be originated by laying on different interpretations of such informal definition.

As a short-term future goal we intend to experiment our approach on complex case study to validate its scalability. In a longer term, we aim at introducing a metamodeling approach to the antipattern definition and to use model-driven engineering techniques to detect and solve performance antipatterns.

The proposed formalization is based on a certain number of thresholds that introduce in the framework a degree of uncertainty. We are working on defining a metric that quantifies such uncertainty as a function of the number of the thresholds an antipattern formalization requires.

Since an antipattern is made of a problem description as well as a solution description, we are working on the solution representation. Solutions of many antipatterns can be represented with the same instruments, because they are expressed as patterns that actually remove the original problems.

At the moment, with this formalization we are able to detect antipatterns. However, we are working to exploit the formalization provided here to automatically deduce antipattern solution (that is system re-design). Just to give a hint, being an antipattern expressed as a logical formula, the negation of such formula should provide some suggestion on the actions to take to solve the antipattern. This would represent a general approach to automatically solve any new type of antipattern that can be defined in future.

Finally, whenever a list of present antipatterns is extracted, heuristic approaches have to be identified to decide how many and which antipatterns to remove among the found ones. This solution process can be quite complex, and the antipattern formalization is only the first step. For example, metrics can be introduced to quantify the role of views in the definition of an antipattern. In fact, once defined, an antipattern must be searched, often with incomplete information available to analysts. Therefore the searching process is basically driven from heuristics, and metrics are crucial to effectively drive such process.

ACKNOWLEDGMENT

This work has been partly supported by the Project PACO (Performability-Aware Computing: Logics, Models, and Languages).

REFERENCES

[1] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: A survey," *IEEE Trans. Softw. Eng.*, vol. 30, no. 5, pp. 295–310, 2004.

[2] S. Bernardi, S. Donatelli, and J. Merseguer, "From uml sequence diagrams and statecharts to analysable petrinet models," in *WOSP*, 2002, pp. 35–45.

[3] C. M. Woodside, D. C. Petriu, D. B. Petriu, H. Shen, T. Israr, and J. Merseguer, "Performance by unified model analysis (puma)," in *WOSP*, 2005, pp. 1–12.

[4] C. U. Smith and L. G. Williams, "More new software performance antipatterns: Even more ways to shoot yourself in the foot," in *Computer Measurement Group Conference*, 2003.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Ed., 1995.

[6] W. J. Brown, R. C. Malveau, H. W. M. III, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, J. Wiley and Sons, Eds., 1998.

[7] "UML 2.0 Superstructure Specification, OMG document formal/05-07-04, Object Management Group, Inc. (2005), *http://www.omg.org/cgi-bin/doc?formal/05-07-04*."

[8] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Trans. Softw. Eng.*, vol. 26, no. 1, pp. 70–93, 2000.

[9] C. M. Woodside, "A three-view model for performance engineering of concurrent software," *IEEE Trans. Softw. Eng.*, vol. 21, no. 9, pp. 754–767, 1995.

[10] V. Cortellessa, A. Di Marco, and C. Trubiani, "Modeling and characterizing performance antipatterns," Dipartimento di Informatica, http://www.di.univaq.it/cortelle/docs/002-2009-report.pdf, Tech. Rep. TRCS 002/2009, 2009.

[11] "Package org.w3c.dom, *http://java.sun.com/j2se/1.4.2/docs/api/org/w3c/dom/package-summary.html*."

[12] V. Cortellessa and R. Mirandola, "Prima-uml: a performance validation incremental methodology on early uml diagrams," *Sci. Comput. Program.*, vol. 44, no. 1, pp. 101–129, 2002.

[13] *UML Profile for MARTE, ptc/08-06-09*, Object Management Group, Inc, 2008. [Online]. Available: *http://www.omgmarte.org/Documents/Specifications/08-06-09.pdf*

[14] "http://www.di.univaq.it/catia.trubiani/repository/ECSsystem.xml."

[15] N. Trcka, W. M. van der Aalst, and N. Sidorova, "Data-flow anti-patterns: Discovering dataflow errors in workflows," in *Conference on Advanced Information Systems (CAiSE)*, vol. 5565. LNCS Springer, 2009, pp. 425–439.

[16] S. Boroday, A. Petrenko, J. Singh, and H. Hallal, "Dynamic analysis of java applications for multithreaded antipatterns," in *Workshop on Dynamic Analysis (WODA)*, 2005, pp. 1–7.

[17] B. Dudney, S. Asbury, J. K. Krozak, and K. Wittkopf, *J2EE Antipatterns*, J. Wiley and Sons, Eds., 2003.

[18] B. Tate, M. Clark, B. Lee, and P. Linskey, *Bitter EJB*, Manning, Ed., 2003.

[19] T. Parsons and J. Murphy, "Detecting performance antipatterns in component based enterprise systems," *Journal of Object Technology*, vol. 7, no. 3, pp. 55–90, 2008.

[20] "http://www.jessrules.com/jess/index.shtml."

[21] J. Xu, "Rule-based automatic software performance diagnosis and improvement," in *WOSP*, 2008.

[22] T. Taibi and D. C. L. Ngo, "Formal specification of design patterns - a balanced approach," *Journal of Object Technology*, vol. 2, no. 4, pp. 127–140, 2003.

[23] G. Kniesel, J. Hannemann, and T. Rho, "A comparison of logic-based infrastructures for concern detection and extraction," in *Workshop LATE*, 2007.

[24] O. Florescu, J. Voeten, B. Theelen, and H. Corporaal, "Patterns for automatic generation of soft real-time system models," *SIMULATION*, vol. 85, no. 11/12, pp. 709–734, 2009.