Applying Performance Patterns for Requirements Analysis

Azadeh Alebrahim, paluno – The Ruhr Institute for Software Technology, Germany
Maritta Heisel, paluno – The Ruhr Institute for Software Technology, Germany

Performance as one of the critical quality requirements for the success of a software system must be integrated into software development from the beginning to prevent performance problems. Analyzing and modeling performance demands knowledge of performance experts and analysts. In order to integrate performance analysis into software analysis and design methods, performance-specific properties known as domain knowledge have to be identified, analyzed, and documented properly. In this paper, we propose the performance analysis method PoPeRA to guide the requirements engineer in dealing with performance problems as early as possible in requirements analysis. Our structured method provides support for identifying potential performance problems using performance-specific domain knowledge attached to the requirement models. To deal with identified performance problems, we make use of *performance analysis patterns* to be applied to the requirement models in the requirements engineering phase. To show the application of our approach, we illustrate it with the case study *CoCoME*, a trading system to be deployed in supermarkets for handling sales.

Categories and Subject Descriptors: I.5.2 [**Pattern Recognition**]: Design Methodology—*pattern analysis*; D.2.1 [**Software Engineering**]: Requirements/Specifications—*Methodologies*; D.2.9 [**Software Engineering**]: Management—*Software quality assurance (SQA)*; D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*

General Terms: Design, Performance

Additional Key Words and Phrases: Performance patterns, problem frames, requirements engineering, software architecture, UML

ACM Reference Format:

Alebrahim, A. and Heisel, M. 2015. Applying Performance Patterns for Requirements Analysis jn 2, 3, Article 1 (May 2010), 15 pages.

1. INTRODUCTION

Problems such as loss of productivity, loss of customers, cost overruns, etc. arise when software systems are constructed without having performance in mind [Smith and Williams 2006]. Fixing such problems afterwards might be costly or even hardly possible. The software after fixing such problems might be erroneous or might not perform as well as software which has been constructed under performance considerations [Smith and Williams 1993]. Therefore, performance as one of the critical quality requirements to the success of a software system must be integrated from the beginning of the software development to prevent performance problems.

Architecture solutions provide a means to satisfy quality requirements. Decisions made in the architecture phase could constrain the achievement of initial requirements, and thus could change them. The sooner architectural knowledge is involved in the process of requirements analysis, the less costly the changes will be. In this paper, we aim at reusing such knowledge in the requirements engineering with regard to performance. There exist solutions to performance problems such as performance patterns [Smith and Williams 2001; Ford et al. 2008] to be applied during the design and implementation phases. We make use of performance analysis patterns [Alebrahim 2015],

Author's address: Azadeh Alebrahim, Oststrasse 99, 47057 Duisburg, Germany; email: azadeh.alebrahim@paluno.uni-due.de; Maritta Heisel, Oststrasse 99, 47057 Duisburg, Germany; email: maritta.heisel@paluno.uni-due.de

which reuse conventional performance patterns and mechanisms and adapt them in a way that they can be used in the requirements analysis.

In this paper, we propose a structured method for *problem-oriented performance requirements analysis (PoPeRA)* that guides the requirements engineer in identifying potential performance-critical resources and their utilization. It then provides a structured way for the treatment of the identified performance problems on the requirements level by using performance analysis patterns. The proposed method relies on the problem-oriented requirements engineering approach problem frames [Jackson 2001].

We use this approach, because 1) it allows decomposing the overall software problem into simple subproblems, thus reducing the complexity of the problem, 2) it makes it possible to annotate subproblems with quality requirements, such as performance requirements (e.g. [Alebrahim et al. 2011b]), 3) it enables various model checking techniques, such as requirements interaction analysis and reconciliation [Alebrahim et al. 2014] due to its semi-formal structure, and 4) it supports a seamless transition from requirements analysis to architectural design (e.g. [Alebrahim et al. 2011a]).

The benefit of the proposed PoPeRA method is manifold. 1) It supports requirements engineers in identifying potential performance problems. 2) It provides guidance for refining performance problems located in the problem space. 3) The elaborated performance requirement models can easily be transformed into a particular solution in the software architecture. Thus, it bridges the gap between two domains, namely the requirements domain and the software architecture & design domain. 4) It supports less experienced software engineers in applying solution approaches early in the requirements analysis in a systematic manner.

The remainder of this paper is organized as follows. We present the background on which our approach is built in Sect. 2. Our PoPeRA method and its application is described in Sect. 3. Related work is discussed in Sect. 4, and a conclusion is given in Sect. 5.

2. BACKGROUND

In this section, we describe the problem frames approach briefly in Sect. 2.1 and the concept of *performance analysis patterns* in Sect. 2.2.

2.1 Problem Frames

Problem frames proposed by Jackson [2001] are patterns for requirements analysis used to understand, describe, and analyze software development problems. In this approach, each problem is decomposed into simple subproblems that fit to a *problem frame*. An instantiated problem frame is a *problem diagram* which basically consists of one *submachine* representing one part of the software to be built, relevant *domains*, *interfaces* between them, and a *requirement* referring to and constraining problem domains. Domains represent parts of the environment which are relevant for the problem at hand. The task is to construct a *(sub-)machine* that improves the behavior of the environment (in which it is integrated) in accordance with the requirement.

We describe problem frames using UML class diagrams, extended by a specific UML profile for problem frames (UML4PF) proposed by Hatebur and Heisel [2010]. A class with the stereotype $\ll$`machine`$\gg$ represents the software to be developed. Jackson distinguishes the domain types biddable domains (represented by the stereotype $\ll$`BiddableDomain`$\gg$) that are usually people, causal domains ($\ll$`CausalDomain`$\gg$) that comply with some physical laws, and lexical domains ($\ll$`LexicalDomain`$\gg$) that are data representations. Figure 1 shows the problem diagram for the requirement *R3* (taken from the application example described in Sect. 3) which is concerned with showing product info. It describes that the machine domain *ShowProductInfoMachine* must present product info on the domain *CashBoxDisplay* through the domain *CashBox* using the domain *ItemId*.

In problem diagrams, *interfaces* connect domains and they contain *shared phenomena*. Shared phenomena may, e.g., be events, operation calls or messages. They are observable by at least two domains, but controlled by only one domain, as indicated by "!". The notation $SPIM!\{presentProductInfo\}$ (between the domains *ShowProductInfoMachine* and *CashBox*) in Fig. 1 means that the phenomenon *presentProductInfo* is controlled by

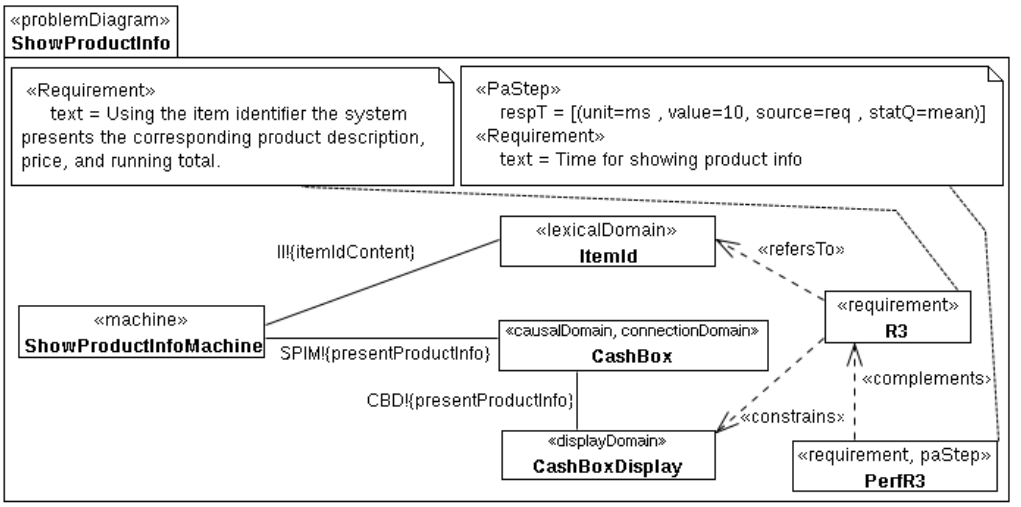

Fig. 1. Problem diagram for showing product info *CoCoME*

the domain *ShowProductInfoMachine*. When we state a requirement we want to change something in the world with the machine to be developed. Therefore, each requirement expressed by the stereotype ≪`requirement`≫ constrains at least one domain. This is expressed by a dependency from the requirement to a domain with the stereotype ≪`constrains`≫. A requirement may refer to several domains in the environment of the machine. This is expressed by a dependency from the requirement to a domain with the stereotype ≪`refersTo`≫. The requirement *R3* in Fig. 1 constrains the domain *CashBoxDisplay*. It refers to the domain *ItemId*.

In the original problem frames approach the focus is on functional requirements. We extended the UML-based problem frames approach by providing a way to attach quality requirements such as security and performance requirements to problem diagrams [Alebrahim et al. 2011b]. This is achieved by providing a dependency from the quality requirement to the functional one with the stereotype ≪`complements`≫. We represent performance requirements as annotations in problem diagrams using the UML profile MARTE [UML Revision Task Force 2009]. Note that it is possible to have more than one performance requirement in one problem diagram. In some cases, if more than one quality requirement is annotated in one problem diagram, conflicts among such requirements might emerge. Therefore, such potential conflicts should be detected and resolved. We discuss this in our previous work [Alebrahim et al. 2014]. In Fig. 1, the performance requirement *PerfR3* complements the functional requirement *R3*. It states that the time for showing product info should be 10 ms in average.

We believe that requirements engineering and architectural design must be integrated in such a way that the knowledge gained in requirements analysis is used in a systematic way when developing a software architecture. Incorporating solution approaches right from the beginning in requirements analysis facilitates the seamless transition from requirements to architectural design. Hence, we extended the original problem frames approach with solution approaches for quality requirements [Alebrahim et al. 2011a]. We not only provide annotations for modeling quality requirements in the problem diagrams, but also incorporate solution approaches for the annotated quality requirements.

2.2 Performance Analysis Patterns

Architectural patterns provide a means to fulfill quality requirements. Performance patterns convey essential performance-specific information and principles for facilitating the reuse of performance knowledge. In our previous

Table I. Template for performance analysis patterns

1) Name	Name of the pattern
2) Description	Brief description of the pattern
3) Also known as	Other well-known names for the pattern, if any
4) Problem	Situation and structure of the problem
5) Applicability	Conditions under which the pattern can be applied
6) Solution	Structure of the solution using stereotypes from UML4PF and MARTE
7) Collaboration	Behavior Description of solution elements
8) Benefits	Benefits of applying the pattern
9) Consequences	Consequences and hints to be considered when applying the pattern
10) Related patterns	Another pattern related to the pattern

work [Alebrahim 2015], we proposed an adaptation of existing performance patterns from the literature [Smith and Williams 2001; Ford et al. 2008] to deal with performance problems in the requirements engineering phase. We called the adapted performance patterns *performance analysis patterns*, which we use in this paper in the PoPeRA method.

The original performance patterns only describe the principle of the solution. They do not provide any structure of the problem. Our adaptation allows the use of such performance patterns in the requirements analysis for analyzing performance problems and providing solution approaches for avoiding such problems. We adapted four existing performance patterns, namely *First Things First* [Smith and Williams 2001], *Flex Time* [Smith and Williams 2001], *Master-Worker* [Ford et al. 2008], and *Load Balancer* [Ford et al. 2008] for requirements analysis [Alebrahim 2015]. These patterns provide solutions for the problem situation where an overload of the system is expected.

Performance analysis patterns encompass a generic template for describing performance patterns textually. The template has to be instantiated for each performance pattern explicitly (see Table I). It is inspired by the GOF template [Gamma et al. 1995]. It, however, contains additional information for representing performance-specific information. The fields *problem* and *applicability* describe when the pattern can be applied. They represent the pre-conditions for the pattern at hand. The fields *solution*, *collaboration*, *benefits*, and *consequences* describe the solution including its elements, their relationships, and their behavior. They represent the post-conditions for the pattern at hand. The proposed template allows software engineers to define new performance analysis patterns according to this structure as well. We describe each pattern as one instance of the template given in Table I. The instantiations of the template for the First Things First (FTF), Flex Time (FT), Master Worker (MW), and Load Balancer (LB) are shown in Tables II, III, IV, and V.

Table II. First Things First Pattern

1) Name	First Things First (FTF)
2) Description	FTF ensures that the most important tasks will be processed if not every task can be processed.
3) Also known as	-
4) Problem	A temporary overload of inbound requests is expected. This situation may overwhelm the processing capacity of a specific resource (see the generic problem frame in Fig. 2).
5) Applicability	FTF pattern is only applicable when there is a temporary overload. That is, the attribute *overloadType* of the stereotype ≪bottleneck≫ in Fig. 2 should have the value *temporary*.
6) Solution	The solution uses the strategy of prioritizing tasks and performing the important tasks with high-priority first. A new machine is introduced that takes the responsibility for prioritizing the tasks and assigning them to corresponding domains.
7) Collaboration	When requests are issued, they arrive through *Domain1* at the newly introduced machine domain *FTF/FT/MW/LB*, which takes the responsibility to prioritize the requests and forward them to the corresponding machine that performs the requests using the domain *Domain2* (see Fig.3). Note that there exists only one machine domain *Machine*.
8) Benefits	FTF reduces the contention delay for high-priority tasks.
9) Consequences	In the case of a permanent overload, applying this pattern would cause the starving of low-priority tasks.
10) Related patterns	LB pattern can be used to improve the processing capacity if the overload is not temporary.

Table III. Flex Time Pattern

1) Name	Flex Time (FT)
2) Description	FT moves the load to a different period of time where the inbound requests do not exceed the processing capacity of the resource.
3) Also known as	-
4) Problem	An overload of the system is expected. The inbound requests exceed the processing capacity of a specific resource (see the generic problem frame in Fig. 2).
5) Applicability	FT is only applicable when some tasks can be performed at a different period of time. That is, the attributes *loadDistributionType* and *overloadType* of the stereotype ≪`bottleneck`≫ in Fig. 2 have the values *temporally* and *permanent*.
6) Solution	The solution uses the strategy of spreading the load at a different period of time. A new machine is introduced that takes the responsibility for modifying the processing time of the tasks and assigning them to corresponding domains for processing in the specified time.
7) Collaboration	When requests are issued, they arrive through *Domain1* at the newly introduced machine *FTF/FT/MW/LB*, which takes the responsibility to spread the requests at a different period of time to be processed by the corresponding machine using the domain *Domain2* (see Fig.3). Note that there exists only one machine domain *Machine*.
8) Benefits	FT pattern reduces the load of the system by spreading it temporally.
9) Consequences	The order of satisfying requirements will be changed. It has to be checked that this modification does not cause new bottlenecks.
10) Related patterns	LB pattern can be used to reduce the load if the tasks cannot be performed at a different period of time.

Table IV. Master Worker Pattern

1) Name	Master-Worker (MW)
2) Description	MW pattern makes it possible to serve requests in parallel. It distributes the load over two or more software resources.
3) Also known as	Computation replicating
4) Problem	An overload of the system is expected. The inbound requests exceed the processing capacity of a specific resource (see the generic problem frame in Figure 2).
5) Applicability	MW pattern is only applicable when the resource which is the bottleneck is a software resource, the overload is permanent, and the load can be spread spatially. That is, the attributes *loadDistributionType* and *overloadType*, and *resourceType* of the stereotype ≪`bottleneck`≫ in Figure 2 have the values *spatially*, *permanent*, and *software*.
6) Solution	The solution uses the strategy of spreading the load over several software resources.
7) Collaboration	When requests are issued, they arrive through *Domain1* at the newly introduced machine *FTF/FT/MW/LB*, which takes the responsibility to forward the request to one corresponding machine which is free (see Fig.3). The selected machine processes the request, creates a response using the domain *Domain2*, and sends the response. Note that there exist at least two machine domains of the same type.
8) Benefits	MW pattern reduces the load of the system by spreading it spatially.
9) Consequences	Efficient algorithm for allocating the requests to responders is required to ensure that the newly introduced machine does not become the new bottleneck.
10) Related patterns	LB pattern can be used to reduce the load if the bottleneck is a hardware resource. FT pattern can be used if the tasks can be performed at a different period of time. FTF pattern can be used when there is a temporary overload.

Table V. Load Balancer Pattern

1) Name	Load Balancer (LB)
2) Description	LB pattern is used to distribute computational load evenly over two or more hardware resources. In contrast to MW pattern that uses software resources, this pattern provides a hardware solution.
3) Also known as	-
4) Problem	An overload of the system is expected. The inbound requests exceed the processing capacity of a specific hardware resource (see the generic problem frame in Fig. 2).
5) Applicability	LB pattern is only applicable when the resource which is the bottleneck is a hardware resource, the overload is permanent, and the load can be spread spatially. That is, the attributes *loadDistributionType* and *overloadType*, and *resourceType* of the stereotype ≪`bottleneck`≫ in Fig. 2 have the values *spatially*, *permanent*, and *hardware*.
6) Solution	The solution uses the strategy of spreading the load over several hardware resources.
7) Collaboration	When requests are issued, they arrive through *Domain1* at the newly introduced machine *FTF/FT/MW/LB*, which takes the responsibility to forward the request to one corresponding machine which is free (see Fig.3). The selected machine processes the request, creates a response, and sends the response using the domain *Domain1*.
8) Benefits	LB pattern reduces the load of the system by spreading it spatially.
9) Consequences	Efficient algorithm for allocating the requests to responders is required to ensure that the newly introduced *LBMachine* does not become the new bottleneck.
10) Related patterns	MW pattern can be used to reduce the load if the bottleneck is a software resource. FT pattern can be used if the tasks can be performed at a different period of time. FTF pattern can be used when there is a temporary overload.

In addition to the template, we provide two problem diagrams describing the *generic problem structure* as well as the *generic solution structure*. To describe the problem situation, we provide only one *generic problem structure*, which fits all performance analysis patterns. The reason is that the lack of resources is the essence of most performance problems. This is the case when more requests have to be processed at the same time than the resources can process. Hence, there is only one problem diagram describing the generic problem structure. Nevertheless, the conditions under which we apply the performance analysis patterns are different. This is captured in the field *applicability* in the template as well [Alebrahim 2015]. The same holds for the solution. That is, the structure of the solution is similar for all performance analysis patterns. Nevertheless, they behave differently to solve problems that have the same structure but different applicability conditions.

Figure 2 shows the problem diagram describing the *generic problem structure*. Domains contained in this problem frame are:

(1) **One domain *Machine* as a machine domain**, which represents a resource expressed by the stereotype ≪resource≫. The resource is expected to be the bottleneck which cannot complete all inbound requests (see the stereotype ≪bottleneck≫)

(2) **Two domains *Domain1* and *Domain2***, which might be required for the subproblem at hand. Note that these domains must not be instantiated necessarily.

(3) **One requirement *Requirement***, which describes the functional requirement to be satisfied. It requires the processing of the requests.

(4) **One *PerformanceReq***, which describes the performance requirement. It requires the satisfaction of the functional requirement *Requirement* within a specific time.

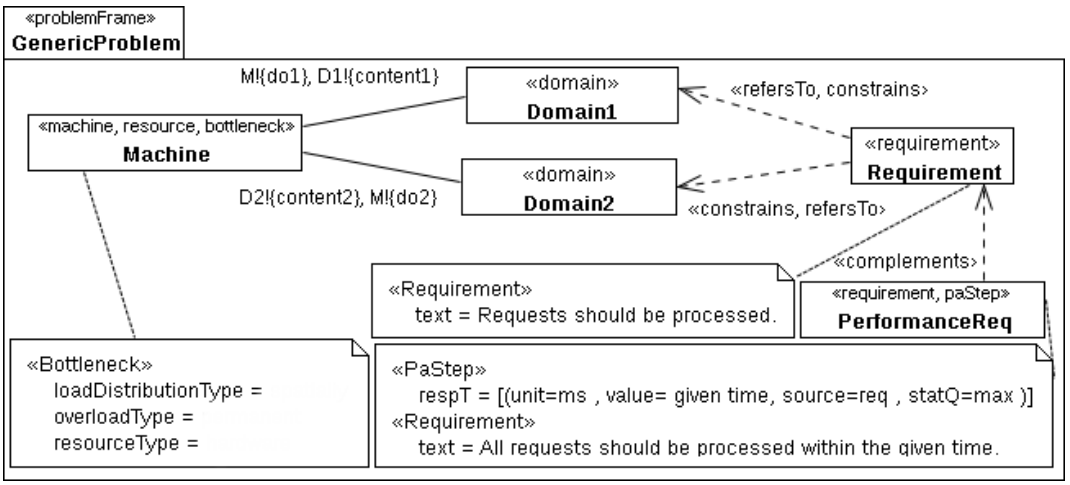

Fig. 2. Problem diagram describing *generic problem structure*

Figure 3 shows the problem frame describing the *generic solution structure*. We introduce the new machine domain *FTF/FT/MW/LB* to compose several machine domains that are bottlenecks (see machine domain *Machine* in Fig. 2) in order to prevent the overload for each single machine domain. Domains contained in this problem diagram are:

(1) **One domain *FTF/FT/MW/LB*** as a machine domain and as a resource.

(2) **At least one domain** as machine domain and as resource responsible for responding to the requests.

(3) **One domain *Domain1***, which transmits the requests to the machine domain Machine.

(4) **One domain *Domain2*** required for processing the requests.

(5) **One functional requirement *Requirement***, to be satisfied by the machine domains *Machine* (at least one machine domain).

(6) **One performance requirement *PerformanceReq***, to be satisfied by the machine domains *Machine* (at least one machine domain).

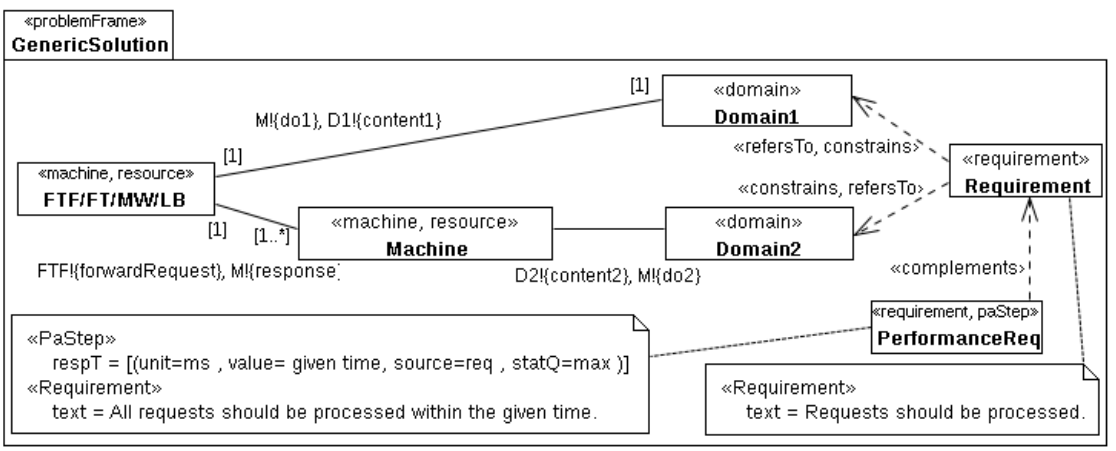

Fig. 3. Problem frame describing generic solution structure

3. THE POPERA METHOD AND ITS APPLICATION

In this section, we present our method for *problem-oriented performance requirements analysis (PoPeRA)*. It is concerned with identifying performance-specific resources, their capacity and utilization, how resources are shared, where performance problems are located, and how they can be treated by applying appropriate performance analysis patterns. Our proposed method is visualized in Fig. 4. The requirement models, namely the problem diagrams (PD), are assumed to be available.

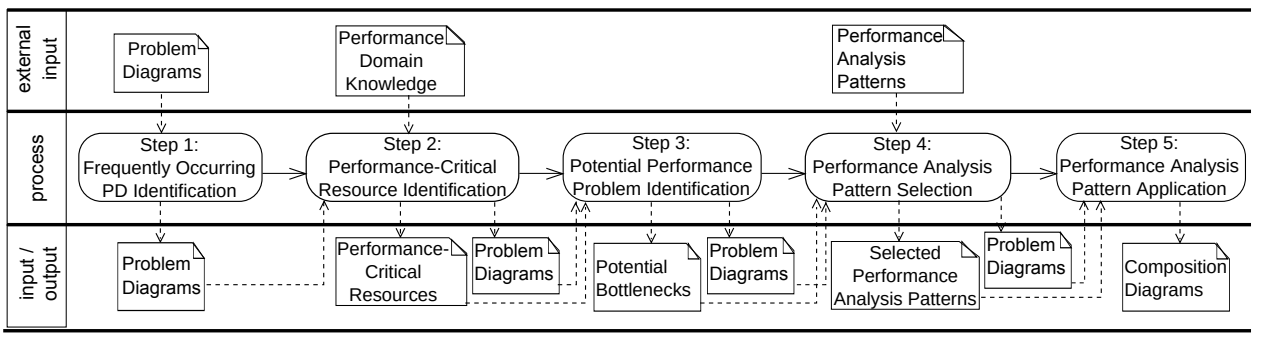

Fig. 4. Overview of the PoPeRA method

To illustrate the application of our method, we apply it to the trading system *Common Component Modeling Example (CoCoME)* [Rausch et al. 2008], a common example including properties of a real world system. Different tasks can be performed using this trading system, such as scanning the products using a *bar code scanner*, paying by cash or credit card taking place at a single *cash desk* as well as administrative tasks such as ordering products. The cash desk is operated by the *cashier* who scans the products the *customer* wants to buy. The corresponding product description, price, and running total are displayed on the *cash box display*. A *store* contains several cash desks, called a *cash desk line*. Each cash desk is connected to a *store server*. The manager of each store (*store manager*) can order goods, change the prices, view, and generate reports using the store client. All store servers are connected to an *enterprise server*.

We consider the use case *process sale* containing several activities. We defined six functional requirements for this use case. By means of the requirement *R3*, we illustrate the applicability of our method. The requirement *R3* requires showing of product info (description, price, and running total) using the item identifier. In the following, we describe the *PoPeRa* method followed by its application to the CoCoME example.

3.1 Step 1: Frequently Occurring PD Identification

To analyze the performance of the system-to-be, the performance analyst must know about performance critical scenarios which are represented by those subproblems that occur frequently. Frequently occurring subproblems could more likely cause a bottleneck in the system than those ones occurring rarely. Hence, the performance analyst has to identify the frequency of occurrence of each subproblem in this step.

First, we specify the allocation of basic functions (subproblems) to the architectural elements. There is no precise deployment possible in such an early phase. However, we provide a mapping of subproblems to the physical environments and decompose the subproblems according to this allocation. For example, in a distributed client-server system, we decompose the subproblems so that each subproblem is assigned only to the client or to the server. Specifying such an allocation is essential when identifying the frequency of occurrence of each subproblem. According to this decision, we split the subproblems. This supports us in eliciting and modeling hardware resources as domain knowledge and subsequently determining the mapping of software parts to hardware resources later on in step 3. As result of this step, we obtain a subset of problem diagrams which will be executed more frequently than the others (see the output of step 1 in Fig. 4).

Application of Step 1: Frequently Occurring PD Identification

Figure 1 describes the overall problem *showing product info*. Figure 5 illustrates how this overall problem is split into the three subproblems *sending itemId to the server*, *requesting product info from the server*, and *presenting product info*. The first and third subproblems are allocated to the *cash desk* and the second one to the *store server*. Functional requirement *R3* is split into the functional requirements *R3-1*, *R3-2*, and *R3-3*. Performance requirement *PerfR3* is decomposed into the three performance requirements *PerfR3-1*, *PerfR3-2*, and *PerfR3-3*. Performance requirement *PerfR3* requiring 10 ms for showing product info must be achieved through these three performance requirements. For more information regarding the decomposition of problem diagrams, we refer to our previous work [Alebrahim et al. 2011b].

Frequently occurring subproblems could more likely cause a bottleneck in the system than those occurring rarely. In this step, we identify such subproblems. Considering all the subproblems related to the use case *process sale*, we determine that the subproblems related to the requirements *R3* and *R5* occur more often than the others. The reason is that these subproblems are concerned with selling products and logging product sales. Considering the requirement *R3*, we identify the subproblems *ShowProductInfoClientRequest*, *ShowProductInfoServer*, and *ShowProductInfoClientResp* as frequently occurring problem diagrams. This information supports the performance analyst later on in identifying potential bottlenecks in the system.

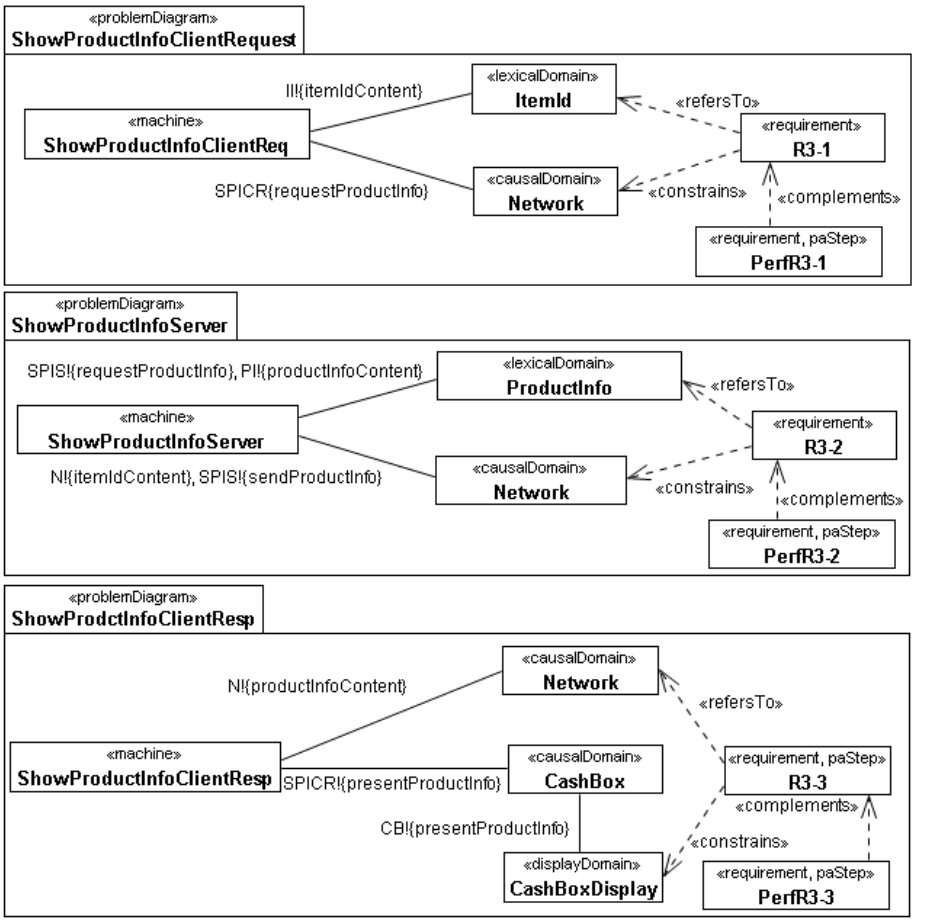

Fig. 5. Allocation of the subproblem for showing product info *CoCoME*

3.2 Step 2: Performance-Critical Resource Identification

Performance is concerned with the *workload* of the system and the available *resources* to process the workload [Bass et al. 2003]. Each resource is expressed by its type in the system (such as CPU, memory, I/O device, or network), its utilization, and its capacity (such as the transmission speed for a network). The system has to process the requests (or satisfy the requirements) caused by the workload using available resources. If many requests have to be processed at the same time by the same resource and the inbound requests exceed the processing capacity of the resource, the resource will be in contention that leads to delays for some requests and subsequently to not achieving the performance requirements. Typically, such resources are referred to as *bottlenecks*. Identifying the location of bottlenecks is critical for the performance of a system.

Hence, in this step, we look for those problem diagrams whose corresponding machines consume the same resource at the same time. Such problem diagrams might cause contention for a specific resource. We identify resources that are used in several problem diagrams. We call such resources *performance-critical resources*. In order to identify resources as critical, we have to elicit and model them in the problem diagrams as *domain knowledge* [Alebrahim et al. 2014]. Domain knowledge for performance represents the important information about

the system and the environment that affects the achievement of performance requirements. In order to identify relevant resources required for the performance analysis systematically, we iterate over the domains in each problem diagram. For each domain we have to check if it represents or contains any hardware device that the machine is executed on or any resource that can be consumed by the corresponding performance requirement. In this case, the domain is a performance-specific resource and has to be annotated as such a resource. To model domain knowledge regarding resources (*Memory, Network, Processor*), we apply the stereotypes ≪HwMemory≫, ≪HwMedia≫, and ≪HwProcessor≫ [UML Revision Task Force 2009]. As a result, we obtain problem diagrams that are annotated with performance domain knowledge.

To identify *performance-critical resources*, we iterate over the domains marked as resources in the problem diagrams. Each resource has to be checked for its occurrence in other problem diagrams as well. If this is the case, it has to be marked as a *performance-critical resource*.

Application of Step 2: Performance-Critical Resource Identification

For this step, we consider problem diagrams allocated to one physical environment. The server part of the system has to process many requests at the same time, whereas the client part does not provide a critical physical environment from the performance perspective. Therefore, we consider problem diagrams allocated to the server part for the rest of this paper. We consider the problem diagram *ShowProductInfoServer* related to the requirement *R3-2* shown in Fig. 6.

The causal domain *Network* has to be annotated as a resource. We apply the stereotype ≪hwMedia≫ for this domain. We use the stereotype ≪storageResource≫ for the lexical domain *ProductInfo*. The stereotype ≪resource≫ is used for annotating the machine domain *ShowProductInfoServer*. The hardware resource *CPU* has to be modeled explicitly as a causal domain. It has to be annotated with the stereotypes ≪hwProcessor≫ and ≪hwMemory≫. Figure 6 shows the problem diagram *ShowProductInfoServer* annotated with performance-specific domain knowledge.

For each domain annotated as a resource, we check if it is used to achieve more than one requirement at the same time. This is the case when the domain occurs in more than one problem diagram. This condition holds for the causal domain *CPU*, as it is used by all the machines on the server side. *CPU* therefore represents a *performance-critical resource*. The causal domain *Network* represents a *performance-critical resource* as well, as it is used by several subproblems.

3.3 Step 3: Potential Performance Problem Identification

This step has to be supported by a performance analyst to analyze whether the processing capacity of existing resources (modeled as domain knowledge) suffices to satisfy performance requirements for each subproblem with regard to the existing workload and frequency of occurring problem diagrams (identified in step 1). The workload has to be modeled as domain knowledge as well. To annotate the workload, we make use of the stereotype ≪gaWorkloadEvent≫ (see the MARTE profile for more information [UML Revision Task Force 2009]).

As a result of this step, those resources, in which the inbound requests might exceed the processing capacity of the resource, are identified. That is, from the set of *performance-critical resources* we retain those resources whose problem diagrams exhibit a high usage. We mark such resources as *bottlenecks* using the stereotype ≪bottleneck≫.

Application of Step 3: Potential Performance Problem Identification

Using the results obtained from the previous steps, we have to identify potential bottlenecks. To this end, we have to model workload and behavior scenarios. According to the CoCoME description, 8 customers are at the same time in the system for the subproblem *ShowProductInfoServer*. The workload for this subproblem is shown in Fig. 6.

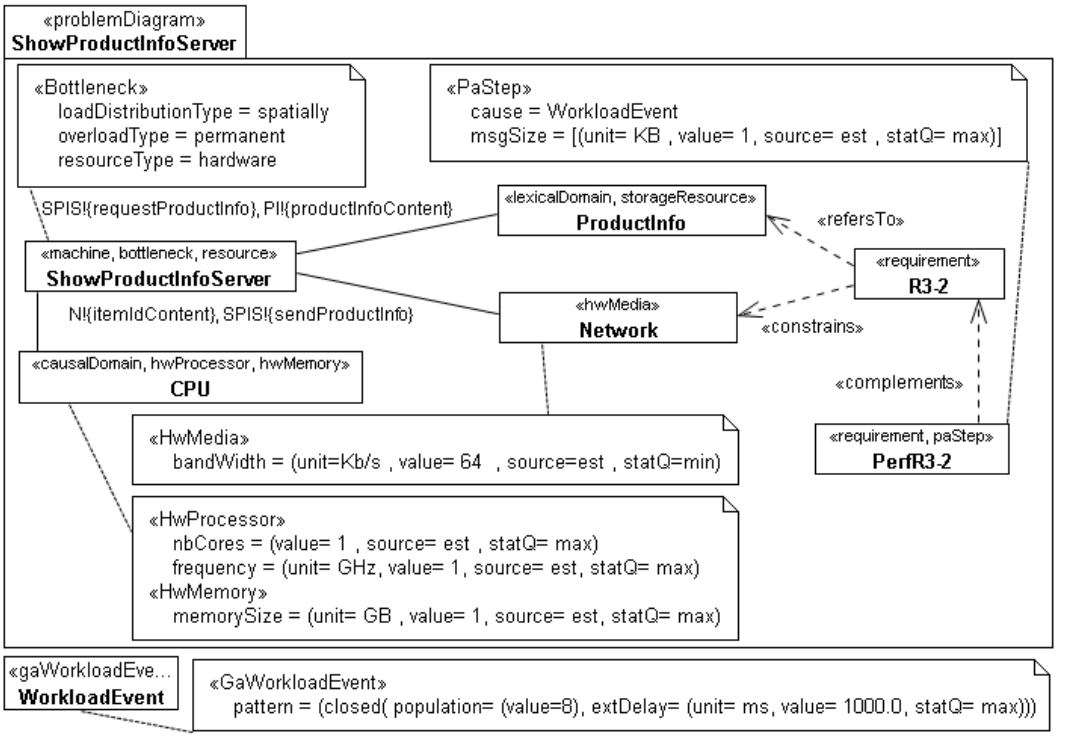

Fig. 6. PD *ShowProductInfoServer* annotated with performance-specific domain knowledge

The subproblem *ShowProductInfoServer* (see Fig. 6) is the most critical. It exhibits high usage as identified in step 1 and high workload (see *WorkloadEvent* in Fig. 6). It might not be able to satisfy its performance requirement. Therefore, we mark it as potential bottleneck. We defined the three attributes *loadDistributionType*, *overloadType*, and *resourceType* for the stereotype ≪`bottleneck`≫ to further specify the characteristics of the bottleneck. This supports us in the next step in selecting the appropriate performance pattern:

—the type of the overload for the subproblem *ShowProductInfoServer* is permanent (*overloadType=permanent*),

—the subproblem cannot be moved to a different period of time (*loadDistributionType=spatially*), and

—the resource (*CPU*) in contention is a hardware resource (*resourceType=hardware*) (see Fig. 6).

3.4 Step 4: Performance Analysis Pattern Selection

After identifying potential bottlenecks, we select appropriate performance analysis patterns that we introduced in Sect. 2.2 in order to prevent such potential performance problems. The field *applicability* in the template for performance analysis patterns represents the pre-conditions each subproblem has to fulfill before applying the specific pattern. To this end, we consider the identified potential bottlenecks. For each subproblem containing a resource marked as bottleneck, we have to determine whether the specific subproblem fulfills the required pre-conditions by answering the questions 1) *Does the subproblem exhibit a permanent or a temporary high usage?*, 2) *Can the subproblem be satisfied at a different time (spatially or temporally)?*, and 3) *Does the resource causing the bottleneck provide a software or a hardware resource?*

Table VI shows the conditions under which the performance analysis patterns can be applied. It provides support in selecting the appropriate analysis pattern. As an example, the *Load Balancer* pattern can be applied if the

Table VI. Performance analysis patterns and their selection criteria

	selection criteria		
	type of load distribution	type of overload	type of resource
First Things First (FTF)	spatially / temporally	temporary	software / hardware
Flex Time (FT)	temporally	permanent	software / hardware
Load Balancer (LB)	spatially	permanent	hardware
Master-Worker (MW)	spatially	permanent	software

resource causing the bottleneck is a hardware resource, the subproblem cannot be satisfied at a different period of time (spatially), and a permanent overload of the subproblem is expected.

Application of Step 4: Performance Analysis Pattern Selection

In this step, we select appropriate performance analysis patterns for those subproblems containing bottlenecks. We can apply patterns to the subproblems only when subproblems are valid instances of the problem frame describing the *generic problem structure* given in Fig. 2. We can apply patterns to the subproblem *ShowProductInfoServer* (see Fig. 6), as it represents a valid instance of the problem frame describing the *generic problem structure*. The instance contains the following elements:

(1) **One machine domain *ShowProductInfoServer*** responsible for responding to the requests. The bottleneck is a hardware resource that is modeled explicitly (*CPU*).

(2) **One domain *Network*** transmitting the requests to the machine domain *ShowProductInfoServer* and **one domain ProductInfo** required for the subproblem.

(3) **One requirement *R3-2***, which describes the functional requirement.

(4) **One performance requirement *PerfR3-2***, which describes the performance requirement corresponding to the functional requirement *R3-2*.

Furthermore, each specific pattern can be applied to a subproblem if the subproblem fulfills all pre-conditions of the specific pattern given in the field *applicability* of the pattern template. In the previous step, we specified the characteristics of each bottleneck using the attributes *loadDistributionType*, *overloadType*, and *resourceType*, which correspond to the pre-conditions for the application of performance analysis patterns. This helps us to find the appropriate pattern by using the Table VI given in Sect. 3. According to the table, we select *Load Balancer* for the subproblem *ShowProductInfoServer*, as

—the hardware resource *CPU* is the bottleneck,

—the load is spatial, and

—the overload is permanent.

That is, we have to replicate the subproblem and its related hardware, which is the bottleneck, namely the *CPU*.

3.5 Step 5: Performance Analysis Pattern Application

In this step, selected performance analysis patterns are applied to the subproblems containing a resource marked as bottleneck. Such subproblems are instances of the problem frame describing the *generic problem structure* introduced in Sect. 2.2 (see Fig. 2). The fields *solution* and *collaboration* in the template for performance analysis patterns describe how the selected pattern can be applied. Figure 3 exemplifies the composition of subproblems as a problem frame describing the *generic solution structure*. Several patterns can be applied to a subproblem if the subproblem and its related resource fulfill the required pre-conditions shown in Table VI.

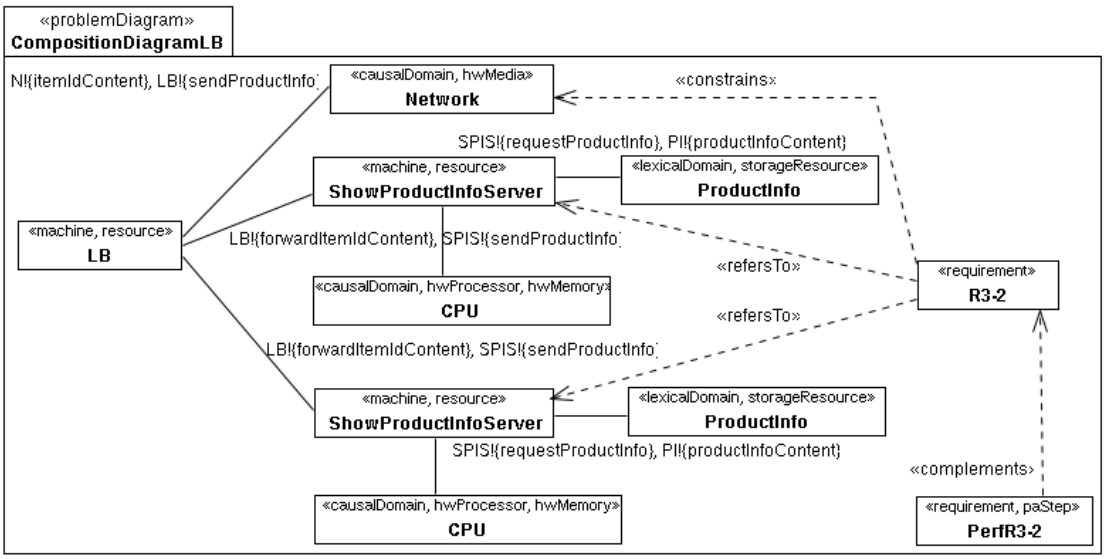

Fig. 7. Load Balancer application to the subproblem *ShowProductInfoServer*

Application of Step 5: Performance Analysis Pattern Application

In this step, we apply the selected patterns to the subproblems. For the subproblem *ShowProductInfoServer*, we selected the *Load Balancer* pattern. Figure 7 shows the application of this pattern to the subproblem *ShowProductInfoServer*.

It is a valid instance of the composition frame describing the *generic solution structure* illustrated in Fig. 3. It contains the following elements:

(1) **One domain *LB* as a machine domain** and as a resource.

(2) **Several domains *ShowProductInfoServer*** from the same type as machine domains and as resources.

(3) **Several domains *ProductInfo***, one for each machine domain. Such domains are used by the machine domain for processing the requests.

(4) **One domain *Network***, which is responsible for transmitting the requests.

(5) **One functional requirement *R3-2*** to be satisfied by several machines *ShowProductInfoServer*.

(6) **One performance requirement *PerfR3-2***, which represents the performance requirement to be satisfied by several machines *ShowProductInfoServer*.

In this way, we performed a performance requirements analysis by applying our PoPeRA method to the trading system CoCoME. We identified locations of potential bottlenecks in the system right from the beginning of the software development. Furthermore, we applied performance analysis patterns to the problematic subproblems to deal with such potential bottlenecks. The PoPeRA method helps requirements engineers and performance analysts finding potential performance problems already in the requirements engineering phase. Complementary performance analysis methods can be used in the further phases of software development when more design details are available.

4. RELATED WORK

A number of approaches that contributed to software performance development have focused on architectural solutions. Nevertheless, information and knowledge needed for dealing with performance issues have to be collected and analyzed early in the software development process. Similar to our approach, Williams and Smith [1995] explore the information needed to construct and evaluate performance models. They define a similar set of information required for early life cycle software performance engineering. They use the terms “execution environment” for resource capacity and “resource requirement” for resource utilization and resource type.

In a later work [Smith and Williams 2004] they present the process software performance engineering (SPE). It relies on use cases and scenarios that describe them. After identifying critical use cases and scenarios, execution graphs are used to determine performance requirements. In further steps, the constructed execution graphs are evaluated to identify performance problems. Although use cases and scenarios build the starting point of the SPE process, it takes in further steps an architectural perspective. While the SPE process uses use cases and scenarios, the PoPeRA method is based on problem frames. In contrast to our PoPeRA method that focuses on performance requirements analysis, the SPE process requires detailed information regarding system resources that are not available in the requirements engineering phase. Hence, the SPE process can be used as complementary to the PoPeRA method afterwards when performance requirements analysis is performed.

Bass et al. [2000] analyze how architectural mechanisms such as fixed priority scheduling and caching help achieving performance as one specific quality requirement. They introduce three strategies: resource allocation, resource arbitration, and resource use for the achievement of performance requirements. Each strategy provides a set of performance mechanisms. Fixed priority scheduling that prioritizes processes to a fixed priority uses the strategy resource arbitration. The authors only describe two mechanisms. This corresponds to our proposed performance analysis patterns. However, they do not provide a systematic method on how to identify performance problems and how to apply such mechanisms.

Approaches that deal with performance in the early software development use mostly use cases and scenarios for analyzing and understanding requirements. The original problem frames approach does not support quality requirements. Some work has been done on security requirements analysis [Schmidt et al. 2011; Hatebur et al. 2008] and dependability requirements analysis [Hatebur and Heisel 2009] based on problem frames. To the best of our knowledge, there has been no research regarding performance requirements analysis based on problem frames.

5. CONCLUSION

In this paper, we proposed our comprehensive method, which is based on problem frames for analysis performance requirements. The PoPeRA method helps the performance analyst identifying potential performance problems as early as possible in the software development process using performance-specific domain knowledge. Furthermore, it provides support for selecting appropriate performance analysis patterns to solve the identified potential performance problems. We showed the application of our method to the case study CoCoME.

To summarize, the PoPeRA method 1) uses problem diagrams to elicit and model performance-specific domain knowledge, 2) uses the modeled performance-specific domain knowledge to identify potential bottlenecks, 3) selects appropriate performance analysis patterns using annotated problem diagrams, 4) applies the selected performance analysis patterns to resolve identified performance problems, and 5) guides the requirements engineer in stepwise analysis of performance requirements.

UML4PF already supports creating requirement models and annotating them with performance information. We strive for extending the UML4PF tool in order to provide support for the PoPeRA method. UML4PF automatically checks the model for semantic errors as well. We extended the list of OCL validation conditions to ensure the integrity and coherency of the model regarding performance-related artifacts that we identified and used in the

PoPeRA method. In addition, we strive for validating our method with another case study to determine the effort spent for executing the method and to further improve it.

6. ACKNOWLEDGMENTS

We would like to thank our shepherd Lise Hvatum for her valuable feedback to improve this paper.

REFERENCES

ALEBRAHIM, A. 2015. Performance analysis patterns for requirements analysis. In *Proceedings of Student Research Forum Papers and Posters, the 41st International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*. CEUR Workshop Proceedings Series, vol. 1326. CEUR-WS.org, 54–66.

ALEBRAHIM, A., CHOPPY, C., FASSBENDER, S., AND HEISEL, M. 2014. Optimizing functional and quality requirements according to stakeholders' goals. In *System Quality and Software Architecture (SQSA)*. Elsevier, 75–120.

ALEBRAHIM, A., HATEBUR, D., AND HEISEL, M. 2011a. A method to derive software architectures from quality requirements. In *APSEC*. IEEE Computer Society, 322–330.

ALEBRAHIM, A., HATEBUR, D., AND HEISEL, M. 2011b. Towards systematic integration of quality requirements into software architecture. In *ECSA*. LNCS 6903. Springer, 17–25.

ALEBRAHIM, A., HEISEL, M., AND MEIS, R. 2014. A structured approach for eliciting, modeling, and using quality-related domain knowledge. In *ICCSA*. LNCS 8583. Springer, 370–386.

BASS, L., CLEMENS, P., AND KAZMAN, R. 2003. *Software architecture in practice* Second Ed. Addison-Wesley.

BASS, L., KLEIN, M., AND BACHMANN, F. 2000. Quality attributes design primitives. Tech. rep., Software Engineering Institute.

FORD, C., GILEADI, I., PURBA, S., AND MOERMAN, M. 2008. *Patterns for Performance and Operability*. Auerbach Publications.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.

HATEBUR, D. AND HEISEL, M. 2009. A foundation for requirements analysis of dependable software. In *SAFECOMP*. LNCS 5775. Springer, 311–325.

HATEBUR, D. AND HEISEL, M. 2010. Making Pattern- and Model-Based Software Development more Rigorous. In *ICFEM*. Springer, 253–269.

HATEBUR, D., HEISEL, M., AND SCHMIDT, H. 2008. Analysis and component-based realization of security requirements. In *AReS*. IEEE Computer Society, 195–203.

JACKSON, M. 2001. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley.

RAUSCH, A., REUSSNER, R., MIRANDOLA, R., AND PLASIL, F. 2008. *The Common Component Modeling Example: Comparing Software Component Models* 1st Ed. LNCS 5153. Springer.

SCHMIDT, H., HATEBUR, D., AND HEISEL, M. 2011. *Software Engineering for Secure Systems: Academic and Industrial Perspectives*. IGI Global, 32–74.

SMITH, C. AND WILLIAMS, L. G. 2004. Software Performance Engineering. In *UML for Real*. Springer, 343–365.

SMITH, C. U. AND WILLIAMS, L. 2001. *Performance solutions, a practical guide to creating responsive, scalable software*. ADDISON WESLEY.

SMITH, C. U. AND WILLIAMS, L. G. 1993. Software performance engineering: A case study including performance comparison with design alternatives. *IEEE Trans. Software Eng. 19,* 7, 720–741.

SMITH, C. U. AND WILLIAMS, L. G. 2006. Five steps to establish software performance engineering. In *Int. CMG Conf.* 507–516.

UML REVISION TASK FORCE. 2009. *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*. http://www.omg.org/spec/MARTE/1.0/PDF.

WILLIAMS, L. G. AND SMITH, C. U. 1995. Information requirements for software performance engineering. In *Computer Performance Evaluation*. Springer, 86–101.