A Basis for Performance Property Prediction of Ubiquitous Self-Adapting Systems

Gunnar Brataas,
Jacqueline Floch
SINTEF ICT
NO-7465 Trondheim
Norway
(+47 735) 92945 / 93012
Gunnar.Brataas@sintef.no
Jacqueline.Floch@sintef.no

Romain Rouvoy
University of Oslo
0316 Oslo
Norway
+47 22852869
Rouvoy@ifi.uio.no

Pyrros Bratskas,
George A. Papadopoulos
University of Cyprus
Nicosia
Cyprus
+357-22892684
Bratskas@cs.ucy.ac.cy
George@cs.ucy.ac.cy

ABSTRACT

Utility-based adaptation approaches permit to determine the “best” suited variant of an application at run-time. Utility policies are usually specified in terms of resources and QoS dimensions. Although utility policies provide a precise formulation for adaptation decision, they are difficult to specify. The developer especially needs assistance in the specification of performance properties. In this paper, we investigate using a performance engineering framework, for the specification of such properties and identify several open research issues.

Categories and Subject Descriptors

C.4 [**Performance of systems**]: Modelling techniques,
D.2.4 [**Software/Program Verification**]: Validation

General Terms

Verification, Performance.

Keywords

Self-adaptation, Performance property specification, Analytical Modelling

1. INTRODUCTION

Ubiquitous computing environments are characterized by frequent changes. To retain usability, usefulness, and reliability in such environments, systems should adapt to changing conditions [1]. The aim of the MUSIC project is to facilitate the development of self-adapting component-based applications for mobile users in ubiquitous computing environments [2]. We follow an architecture-centric approach where architecture models are represented at runtime to allow generic middleware components to reason about and to control adaptation of applications [3]. These architecture models describe the application structure and variability, and adaptation information. Rather than describing explicitly the adaptation actions to take place in particular situations, we adopt a utility-based approach. Utility policies are extended goal policies that ascribe a real-value scalar desirability to system states [4] (in our case system variants). The adaptation middleware thus can compute the utilities of system variants and reasons about the actions required to facilitate an adaptation.

Utility policies express the rationale of an adaptation decision in a precise way, and are therefore more appropriate than action or goal policies when adaptation triggers and effects interfere, or when goals conflict is the case in mobile and ubiquitous environments [5]. A drawback though is that they require the developer to specify the properties of the system variants, and this might be a difficult task. Especially system resource needs (*e.g.*, CPU or network) and the impact of resources on performance are hard to deal with. To facilitate the developer task in specifying such properties, we are investigating a performance engineering (PE) framework. In this position paper, we present an existing research approach that we propose to refine and extend for MUSIC needs and discuss open research issues.

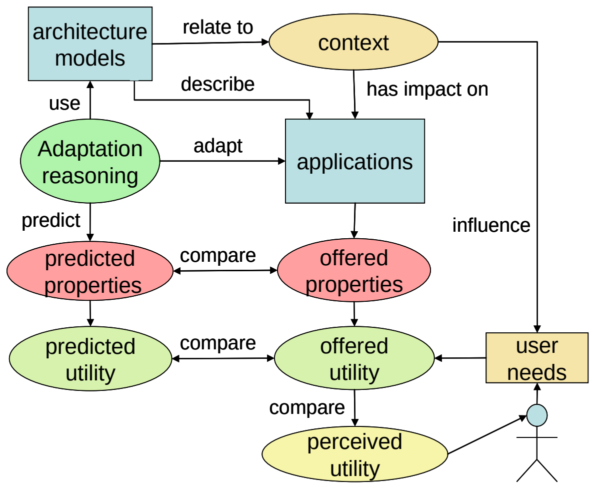

Figure 1. Utility-based adaptation.

2. PROBLEM DESCRIPTION

Figure 1 illustrates adaptation in a utility-based approach. The architecture models specify property predictors allowing the adaptation middleware to predict properties of application variants in a given context. A utility function is a kind of property predictor: it aggregates several properties and take user preferences into account. Examples of property predictors and utility functions are provided in [3]. The property predictors should be correctly specified so that the right adaptation decision is applied in response to context changes. This means that predicted properties should faithfully approximate the "real" offered properties and utilities of application variants.

MUSIC uses component frameworks to design applications and supports compositional variability, allowing application reconfiguration at the component level. This requires specifying property predictors at the component level. Further, a user often runs multiple applications at the same time. This requires adaptation reasoning to take into account the combined resource needs and utility of the set of active applications.

Figure 2 illustrates the different composition levels for a set of self-adapting applications. At each level, the properties and resource requirements should be specified correctly and validated. The component level deals with individual or elementary components. Resource requirements and properties offered by components should be specified correctly. The application level deals with application and possibly composite components. The aggregation of resource requirements and properties from components to applications may introduce discrepancies and must be validated. In addition, the utility of each application model must be checked: is the "real" utility similar? The portfolio level deals with a set of applications. The "aggregation" of resource requirements, properties and utilities from several applications must be validated. The system-and-user level deals with a set of applications deployed on a specific platform and made available to end-users. When all applications are running on the top of the adaptation middleware, subject to context changes and experienced by the user, the offered utility should be checked against the utility perceived by the user.

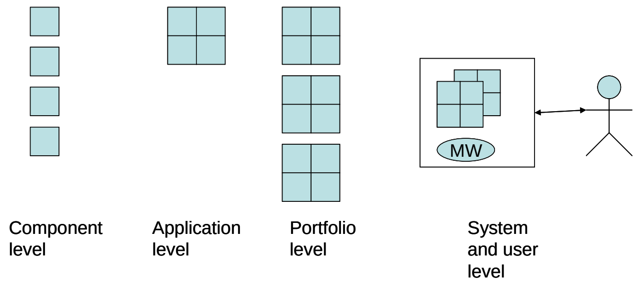

Figure 2. Composition levels.

3. MAIN REQUIREMENTS

Validation and tuning of system properties are generally associated to a fine-grained analytical modelling of the system performances. However, the description of such an analytical model is often time-consuming and error prone for the developer. The objective of our work is to leverage the task of the developer by providing him/her tools and methods to describe these analytical models quickly and efficiently. This means that the PE framework should be integrated in the development process. This integration consists in extending existing tools with support for the particularities of self-adapting systems.

Thus, the PE framework should provide an integrated support to the developer for *(1)* inferring the analytic model of a component, and *(2)* validating the analytic model using component test-beds and context simulations. Finally, this approach should support different types of execution platforms due to the large heterogeneity of ubiquitous devices.

4. ANALYTICAL MODELLING

As a basis for our performance property models we will use analytical performance models where equations are used to derive properties of interest from observable quantities. We propose to use static analytical models for software components and dynamic analytical models for hardware resources. We could have selected other modelling paradigms [6]. More powerful approaches, *e.g.*, layered queuing models [7], allow for contention also at the software levels, but then also require getting more parameters and become more complex. Approaches based on SPE [8] (Software performance engineering) will not have the same component focus as our static model. Nevertheless, it would be interesting to also explore other modelling paradigms in our component-based mobile setting.

As shown in Figure 3, user workload is separated into *work* (*what* is done?) which drives a *static model* and *load* (*how often* is it done?) which drives the *dynamic* model. For dynamic performance models, resource demands, service times and response times are well-defined concepts, *e.g.*, see [9]. The *resource demands* needed by the dynamic model are the product of service times and the devolved work from the static model. *Devolved work* is the number of elementary hardware operations for the top level work. *Service time* is the execution time for each elementary operation on a hardware resource, excluding queueing. *Response times* for top-level operations are finally derived from the dynamic model. A multi-class queuing network [9] can be used for the dynamic model, with one class for each top-level operation. For an open queuing network model, load is represented by arrival rates.

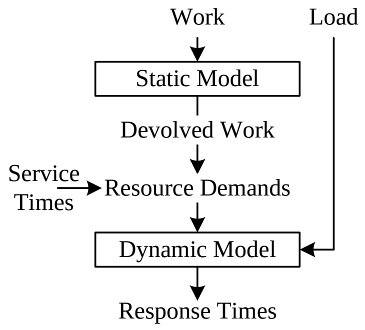

Figure 3. Overall framework.

Figure 4 shows the components in the static model for the service technician example from [3]. Technicians are responsible for inspecting geographically spread technical installations. They use PDAs for status and fault report. When changes occur in the work environment, applications adapt. For example, the application structure might adapt from a thin to a medium or self-reliant client; the UI switches between keyboard- or voice-based.

The static model defined in [10] and validated by [11] describes how work is devolved from user level work through several software components and finally onto work on primary hardware components. The model example shown in Figure 4 has five primary hardware resources, one CPU for the handheld, one CPU (or several if needed) for the server, flash memory on the handheld, one disk (or several disks) on the server and the Internet, which acts as the communication medium. Components are linked together in a directed acyclic graph. The links have three types. Solid thin links represent transformation and processing of information, solid thick links represent persistent storage and dotted thin links represent communication.

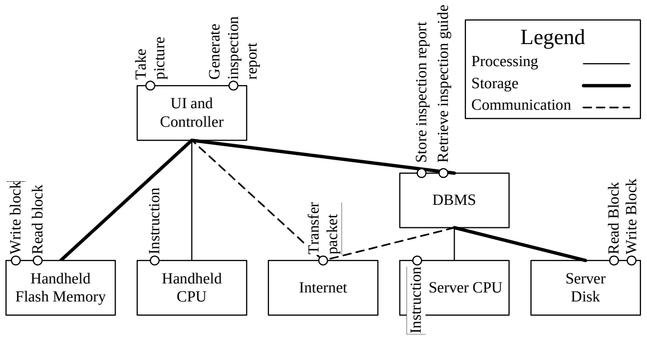

Figure 4 . Static model of Service Technician example.

The first crucial step in making a static model is to decide on which components and links to model. In our model example the `UI` component and the `Controller` component are combined into the `UI and Controller` component to decrease the complexity of the model, and trading off accuracy for cost of measurements. In our example, storage of inspection reports could be done *locally* in the `Handheld Flash Memory` or *remotely* in the `Server Disk`. Each component offers operations, *e.g.*, both the `DBMS` and the `Handheld CPU` will have operations. For the `DBMS, the two operations Store inspection report` and `Retrieve inspection guide` are specified, and for both CPUs we specify on average `Instruction` only. The accuracy of the model increases with the number of operations, but so will also the measurement cost.

Operations on components which are linked together are related with a Complexity Specification Matrix (CSM). Each element in the CSM is a *function* describing the average number of a particular low-level operation which is used by a particular high-level operation. This function will often be simple numbers like 0 (the low-level operation is not used) and 1 (exactly one low-level operation is used for each high-level operation). A more sophisticated function may depend on load and data sizes. An access to a DBMS will for example depend on the *size* of the requested data. A garbage collection component depends on the system *load*, as there is more garbage collection when the load on the system is high, than when it is low.

To compute the devolved work for an application consisting of several components, we use two simple rules. If two links devolve work on the same component, their respective CSMs are *added* together. *Multiplication* is used for combining links coming into a component with the links going out from a component [10] [11].

In our model example the component `UI and Controller` has the two operations `Take picture` and `Generate inspection report`. The complexity matrix showing the relationship between the top-level operations `Take picture` and `Generate inspection report` and the low-level `CPU Instructions` will in this case have two elements:

$$C_{CPU}^{UI\_\&\_Cntrl} = \begin{array}{l} \\ Take\_picture \\ Generate\_report \end{array} \begin{array}{c} Instr. \\ \left(\begin{array}{c} 10^{7} \\ 10^{8} \end{array}\right) \end{array}$$

In this CSM the element 10^8 means that the each `Generate report` operation on the average requires 10^8 CPU instructions.

If we assume the handheld CPU is running on average 10^7 instructions per second, then the service time for each CPU instruction is 10^{-7} seconds. Note a main advantage of this framework: it is easy to adjust for differences in CPU speed between different processors. Only service times have to be adjusted, whereas the static model itself remains unchanged.

The resource demand for each high-level `Generate report` operation is the product of the number of instructions for each `Generate report` operation and the resource demand, and is $10^{8} \cdot 10^{-7} s = 10\,s$, *i.e.*, it takes 10 seconds of CPU time to execute one `Generate report` operation.

To compute the resource requirement on the handheld device, we also need the CSM between `UI and Controller` and `Handheld Flash Memory. The Handheld Flash Memory` has the two operations `RB` (Read block) and `WB` (Write block). We assume a block size of 1 MB. The CSM is defined as:

$$C_{Flash\_mem}^{UI\_\&\_Cntrl} = \begin{array}{l} \\ Take\_picture \\ Generate\_report \end{array} \begin{array}{c} \begin{array}{cc} RB & WB \end{array} \\ \left(\begin{array}{cc} 0 & 10 \\ 100 & 1000 \end{array}\right) \end{array}$$

We assume each read access requires 0.02 s, and each write access requires 0.05 s, therefore each `Take picture` operation will take 0.5 s, while each `Generate report` operation will require $100 \cdot 0.02\,s + 1000 \cdot 0.05\,s = 52\,s$.

During peak load the user generates 1 report every 10 minute, and will in the same time interval take around 20 pictures. For simplicity, and since they are heavily intertwined, we consider the `Handheld CPU` and `Handheld Flash memory` as *one* resource. The utilisation, *U*, of this resource becomes:

$$U = \frac{20\,(1\,s + 0.5\,s) + 1(10\,s + 52\,s)}{10 \cdot 60\,s} = 0.153$$

A utilisation of 0.153 means that the handheld is busy 15.3 % of the time. This is not a significant load, but nonetheless small queues will contribute to the response time. We assume steady state and a random distribution of the load (in practice, the time between individual top-level operations will then be exponentially distributed). Using a multi-class open queuing network model as the dynamic model, then the steady-state, average response time for each `Generate report` operation will be [9]:

$$R = \frac{D}{1-U} = \frac{10\,s + 52\,s}{1-0.153} = 73.2\,s$$

If the storage is distributed to a server, both the load and the response times on the handheld will be smaller, but we must in addition also add the network and the response time contributions from the server (which would be considerably less than for a handheld, given much faster processing and storage devices).

In summary, there are three types of "parameters" or inputs to the models in this framework: 1) the load 2) the service times and 3) the elements in the CSM. Based on the response times it is possible to derive property predictors and utility functions.

5. TEST-BEDS

In our PE framework, the objective of using test-beds is twofold:

1. Test-beds should capture resource demands for the different parameters of the static model based on service time responses.
2. Test-beds should also validate the property predictors defined in the static model (and computed in the dynamic model) by comparing it to measured performances.

If the former objective can be achieved using existing profiling approaches and tools (such as the Eclipse TPTP framework [12]), the latter needs to extend current approaches to take into account domain-specific property predictors. Thus, our PE framework includes test-beds for *(1)* resource demands evaluation, and *(2)* property predictors validation of components.

This first type of test-bed gathers data about the observable performance related to the execution of a component. The resulting information (execution time, list of methods calls, etc.) becomes useful to refine the definition of the static model. In [13], the authors introduce a workbench and a repository dedicated to the gathering of resource demand data. Scenarios are defined for identifying resource demands as both operating system dependencies (*e.g.*, memory, CPU), and invocation dependencies (*e.g.*, libraries). Performance is then evaluated by the test environment and stored in a repository, which contains the demands per operation and the total demands per execution of each scenario. To measure resource demands and eventually to specify the CSMs for primary hardware resources could be time-consuming, but one eventually gets the information one look after. However, to characterise the CSMs for intermediate components could be a more frustrating experience, because enough information may not be readily available making educated guessing necessary [13].

The second type of test-bed addresses the validation of property predictors associated to a component. This includes general performance properties, such as response times and domain-specific QoS properties (*e.g.*, packet-loss rate for streaming applications). The latter cannot be modelled, observed and validated by traditional performance modelling approaches and testing environments. This means that the specification and validation of these predictors should be derived from concrete observations that are possibly abstracted to higher-level quality properties. Thus, it may be necessary to extend domain-specific data structure (*e.g.*, video streams) with instruments, such as high-level metadata, that reify the QoS property model at runtime. For example, the quality of a video stream may be converted to abstract QoS property levels (low, medium, high).

6. CONTEXT SIMULATION

Context simulation contributes to facilitate the capture of dynamic properties. We may simulate two kinds of context:

1. The user work load, allowing one to observe how the load influences the performance properties. A solution to this may be the CLIF framework (http://clif.objectweb.org/).
2. The system resources allowing one to observe the effects that either result from platform variations or from the competition from other applications.

As for the latter is concerned, we propose to exploit the ResourceManager component of the MUSIC Middleware [2] which aims at managing the resource instances present in a given adaptation domain. The ResourceManager provides uniform discovery and configuration interfaces to local lower level resources present in a node, such as memory, CPU, and network resources. Additionally, in a distributed environment, the ResourceManager running on a master node provides information about the global resource model and resource situation in its adaptation domain, including other available nodes. A node, which is also considered as a resource, represents a computational entity in the system that may host application components. The ResourceManager is designed not only for *Resource Discovery* but also for *Observability/Listenability* and *Configuration*. Resource's properties are observable in both pull and push schemes and the resource management should be able to manage the configuration of certain resources and guarantee a certain level of protection when accessing the underlying manageable resource.

The MUSIC resource model supports the modelling of resources in a uniform way. A resource must be generic, modelled uniformly and its range extensible, supporting adding new resource types to cover all resources identified in a given scenario.

7. RESEARCH CHALLENGES

The use of analytical modelling for self-adaptive computer systems is reported for example in [14]. The modelling of performance for a component-oriented software system has been a lively research issue for at least one decade (see for example [15]). However, to our knowledge, analytical modelling of performance properties for *mobile and ubiquitous* self-adapting component-based systems is not addressed in the literature. Some of the research challenges will therefore be remaining research challenges from component-based performance engineering of "vanilla" software, while other research challenges are original because of the new environment. In this section we identify some open research issues.

Component-based performance engineering is not normal practice in software engineering, both because of remaining research challenges and because of its cost in terms of manpower. The rigorous formal basis given in this paper should in practice therefore be simplified, decreasing its accuracy, but more importantly also decreasing its cost. It is however our underlying assumption that it is better to simplify a rigorous formal basis than to work with an ad hoc approach.

In the context of self-adapting systems, a main challenge is to provide the developers with tools that are easy to use and support the specification of accurate models. As our work focuses on everyday systems, we may relax the level of accuracy and detail.

The selected application variant does not need to be the best, but has to be good enough. Also too detailed architectural models would lead to frequent adaptation reasoning resulting in much overhead. Currently in MUSIC, *ports* are defined that combine several operations into one service, this to simplify the specification of property predictors in the architecture models. The static models as introduced in Section 4 require the description of individual operations. We will investigate whether the PE framework can be accommodated to the coarse-grained architecture models proposed in MUSIC.

As MUSIC uses component frameworks facilitating the deployment of new component variants and supporting the composition of new application variants at run-time, we may not be able to specify performance models for complete applications at design time. Rather, we need a modular and incremental approach supporting the instrumentation of components separately. This is also needed by the extended scope of MUSIC to service oriented architectures (SOA).

The cost of the proposed approach is by far dominated by the cost of measurements. For each component, we must define the platform on which the component executes and characterise the resource consumption in terms of this platform (this is done using the CSMs). In the case of the service technician example in Figure 4, the report may be stored either locally on the handheld or on a remote server. In both cases, the CSMs using this platform should be equal. What is changing is the CSM in the platform itself, *i.e.*, the resource consumption for the server on *its* resources will differ radically from the resource consumption for storing the same information on the handheld. However, in the worst case, variability will require new CSMs for each platform and thus also the need for new measurements. We will investigate this further.

The more components, the more connections between components (and thus the more CSMs), the more elements in each CSM, the more complex and costly the approach is. Accuracy requires several measurements for each CSM element, and thus contributes to complexity. One may assume that mobile applications are simpler and that the cost of performance modelling is smaller than for “conventional” software.

Knowledge about the load is usually a critical parameter in PE frameworks. As our aim is to compare several variants, we can assume that all variants are subject to the same load. An approximation about the load is thus probably sufficient. Also in a mobile setting, using a handheld, the number of operations is probably limited. A new issue though is that the load may be influenced by the context changes: the user may use an application more often in a particular context.

To assess the feasibility of the approach and to get more experiences of how the simplifications can be done and on how accurate the models will be in practice, we need case studies. The pilot services developed in the MADAM project [16] that provides the foundation for MUSIC, are a good starting point. It is further interesting to seek for patterns of the relations betweens the components in the static component models.

Finally, we may consider more tricky issues: *(1)* Extensions to the static model may be required to handle the memory consumption for each component itself (*i.e.*, apart from the memory consumption for its operations). In addition, memory constraints in the primary memory (*i.e.*, to estimate the performance implications of using less memory than what is optimally required by and application) may be hard to handle analytically by the dynamic model. *(2)* The approach may be relevant for the modelling of battery consumption related to the modelling of performance, because each elementary operation will consume a given amount of energy (We must in addition take into account that the energy per elementary operation depends on the clock speed, i.e. running a given workload at a *high* clock speed consumes more energy than using a *lower* clock speed.). Energy may also be needed for keeping an idle resource standby.

ACKNOWLEDGMENTS

Thanks to Erlend Stav, SINTEF ICT, for valuable comments. The European project MUSIC (EU IST 035166) supports our work.

REFERENCES

[1] Satyanarayanan, M. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications*, vol. 8, no. 4 , pp. 10-17, 2001.

[2] http://www.ist-music.eu/

[3] Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., and Gjørven, E. Using architecture models for runtime adaptability. *IEEE Software*, vol. 23, no. 2, 2006, 62-70.

[4] Kephart, J.O., and Das, R. Achieving Self-Management via Utility Functions. *IEEE Internet Computing*, vol.11, no. 1 , pp. 40-48, 2007.

[5] Floch, J., Stav E., and Hallsteinsen, S. Interfering effects of adaptation: implications on self-adapting systems architecture. LNCS 4025, pp. 64-69, Springer, 2006.

[6] Woodside, M., Franks, G., Petriu, D.C., The Future of Software Performance Engineering, *Future of Software Engineering*, IEEE, May 2007.

[7] Rolia, J.A. and Sevcik, K.C. *The Method of Layers*, IEEE Trans. on SW Eng., 21(8): pp 689-700, August 1995.

[8] Smith, C.U., Williams, L.G., *Performance Solutions*, Addison Wesley, 2002.

[9] Menasce, D.A. Almeida, V.A.F, Dowdy, L.W. *Performance by Design*, Prentice Hall, 2004.

[10] Hughes, P. *SP Principles*, Technical report, STC Technology, o59/ICL226/0, July 1988.

[11] Brataas, G., Hughes, P.H., Sølvberg, A. Performance Engineering of Human and Computerised Workflows, LNCS 1250, Springer, 1997.

[12] http://www.eclipse.org/tptp/

[13] Woodside M., Vetland V., Courtois M., and Bayarov S. Resource Function Capture for Performance Aspects of Software Components and Sub-systems, LNCS 2047, Springer, 2001.

[14] Menasce, D.A., Bennani, M.N. and Ruan, H. On the Use of Online Performance Models in Self-managing and Self-organising Computer Systems, LNCS 3460, Springer, 2005.

[15] Gomaa, H., Menasce, D.A. Performance Engineering of Component-Based Distributed Software Systems, LNCS 2047, Springer, 2001

[16] http://www.ist-madam.org/