Making Sense of Runtime Architecture for Mobile Phone Software

Alexander Ran, Raimondas Lencevicius
Nokia Research Center
5 Wayside Road, Burlington, MA 01803, USA
Alexander.Ran@nokia.com Raimondas.Lencevicius@nokia.com

ABSTRACT
We present a metamodel for runtime architecture and demonstrate with experimental results how this metamodel can be used to recover, analyze and improve runtime architecture of mobile phone software.

Categories and Subject Descriptors
D.2.11 [**Software Architectures**]: Domain-specific architectures.
D.4.1 [**Process Management**]: Scheduling.

General Terms
Measurement, Performance.

Keywords
Software architecture, performance, architecture recovery

1. INTRODUCTION
Last year our research group was requested to study the runtime architecture of mobile phone software to understand whether some performance aspects of a phone could be improved. The size of mobile phone software is such that a study of detailed design would require significantly more time than could have been reasonably allocated for this project. Therefore we focused on the most essential design decisions that affect runtime operation of mobile phone software or, in other words, runtime software architecture.

In this paper we give an overall account of this experience, which demonstrates that understanding runtime software architecture in terms of featuresets, concurrent tasks and resource scheduling framework makes it possible to recover essential architectural decisions from execution traces and analyze the effect of architectural decisions on software schedulability and performance.

In the next section, we give a description of the application domain to motivate our selection of specific performance metrics and emphasize the special importance of runtime software architecture in this domain. We then explain our understanding of what constitutes runtime software architecture and define the metamodel we have used in this study. We follow with a description of experiments and architecture recovery techniques we used. Finally we analyze the findings and discuss how the recovered information about architecture can be used to recognize patterns for improvement.

ESEC/FSE'03, September 1–9, 2003, Helsinki, Finland

2. UNDERSTANDING APPLICATION DOMAIN
Today personal communication devices are more than voice call terminals. Mobile phones serve as platforms for a variety of mobile applications including text and picture messaging, personal information management, including data synchronization with remote servers and desktop computers. Phones host a range of communication-centered applications most of which have real-time constraints. During a voice call speech data should be processed in a timely fashion to avoid jitter. During a GPRS session lower layer packets arrive every 10 ms. GSM level 3 signaling standard requires 500 ms response to any GSM Layer 3 message. GSM level 2 performance requirements require response to commands within 50ms. GSM-WCDMA handover has 40 ms absolute constraint for completion. These are just few examples of system requirements that lead to tight software performance constraints.

Although it is possible to discuss performance metrics for separate applications, it is less interesting in practice. This is because in most practical cases even the initial implementation of each application performs fine in separation. We could discuss performance of individual applications in terms of hardware utilization figures. Unfortunately utilization figures cannot be easily composed due to possible resource conflicts and timeliness constraints between applications. Therefore such metrics have only limited usefulness when we want to understand system performance.

Many mobile phone applications may execute concurrently but not all. In fact, it is impossible to execute all applications concurrently due to conflict and contention over the use of specific hardware resources on one hand and timeliness and other quality constraints of the applications on the other. It is then essential to identify the sets of applications that are concurrently useful and investigate whether it is possible to execute these sets concurrently on a given hardware. More powerful hardware components cost more and consume more power shortening battery life—a very important metric of mobile phone performance. Thus the most appropriate performance metric for mobile phone software can be expressed as the **ratio between concurrently available functionality over the cost of required hardware resources**.

3. RUNTIME SOFTWARE ARCHITECTURE
In this section, we use a general definition of software architecture and our domain analysis to derive a metamodel for runtime software architecture.

It is quite commonly understood today that in practice **software architecture is a collection of the most essential design decisions** [1]. In [4] we defined software architecture as a set of concepts and design decisions about structure and texture of software that enable effective satisfaction of architecturally significant requirements.

This definition characterizes architectural decisions in three important ways:

- Decisions about structure. What are the parts, how they are related, how they interact and depend on each other? This is the most commonly recognized set of decisions and it is reflected in most common approaches to software architecture
- Decisions about texture. Texture is the recurring microstructure of a system. Crosscutting concerns that cannot be localized in a single component must be addressed across multiple components repeatedly. In a well-designed system this must be done in a uniform fashion, which creates recurring microstructure—the definite texture of a system. Examples of crosscutting concerns include exception handling, execution tracing, overload control, flow control, resource reservation policies to mention a few.
- Architectural decisions are made to enable satisfaction of requirements. Runtime architecture has to address runtime requirements.

As we explained earlier, in our domain, improved software performance provides more functionality or features concurrently available to users. Concurrent availability of software features is determined by what is schedulable on a given hardware. Thus we need to identify decisions about runtime structure and texture that affect schedulability. Structure is created through partition of software into units (elements, components) that stand in a specific relation to each other. The most common units or elements of partition that play an important role at run time are:

- Units of execution. Executable or loadable components are often also units of download and upgrade. These are application executables, dynamically loadable libraries, agents, and dynamically loadable classes.
- Units of protection. These are processes in systems that support virtual address spaces.
- Units of concurrency with respect to resource allocation. These are tasks and threads.

Schedulability is not dependent on the structure of executables and processes, but is affected by the structure of threads or tasks. Thus the structure of runtime software architecture can be expressed as a partition into threads and their relationships. Relationships between threads are expressed by direct dependence in terms of provided and required services and indirect dependence through the use of shared resources.

What are the decisions that determine runtime texture? These are decisions that address crosscutting concerns with respect to the task structure. These are concerns that cannot be localized to one or a small number of tasks whichever task structure we create. The concerns that crosscut the task partition and create runtime texture are related to intertask communication, and resource management policies and mechanisms. These concerns are addressed by choices of messaging infrastructure and resource scheduling framework. Thus the most common decisions about runtime texture are routing, queuing, scheduling and dispatch of messages, policies for reservation, release, and pre-emption of resources.

Thus **runtime software architecture is a partition of all software functions into concurrent units and a scheduling policy that together deliver the best possible service to users with available resources**.

As we discussed in the previous section many software applications running on a mobile phone have real-time constraints. Some applications must be executed concurrently and thus have to be schedulable together. However, if we tried to execute all applications at the same time, the system would not be schedulable. Thus an essential concept in the design of runtime architecture is a *featureset*. A featureset is a set of concurrently available features. Specification of useful featuresets for a given system is an architecturally significant requirement. Runtime architecture should enable schedulability of all featuresets. A featureset can be defined by a collection of use cases that can overlap in time. The scenarios that correspond to these use cases require that objects participating in the scenario can be concurrently active. A featureset determines which objects must be executed concurrently. All objects are allocated to some task. The tasks that contain objects from the same featureset have to be collectively schedulable. Tasks that do not belong to the same featureset do not have to be collectively schedulable. If tasks that belong to the same featureset are not schedulable, some architectural decisions regarding allocation of objects to tasks or resource scheduling policies must be revised.

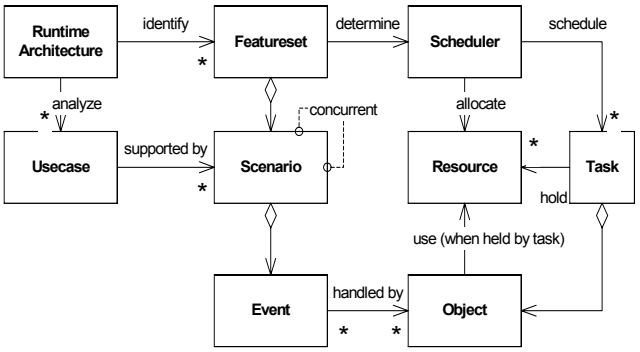

Figure 1. Runtime Architecture Meta Model

Figure 1 presents our metamodel for runtime architecture in the UML notation. Thus the main decisions of software architect regarding runtime architecture are

- Partition into featuresets
- Identification of resources
- Partition into tasks
- Allocation of objects to tasks
- Resource scheduling framework

Since these decisions are made to enable the satisfaction of performance requirements, we need to be able to analyze the runtime architecture in terms of its performance characteristics. This imposes some further constraints on tasks based on the properties of selected resource-scheduling framework. The next

section will discuss the task model expected by most commonly used scheduling frameworks.

4. SCHEDULING FRAMEWORK

Most industrial real-time systems today still rely on fixed priority scheduling. We used in this study the framework of rate monotonic analysis and scheduling (RMA) [2][3]. In its simplest form, RMA requires tasks to be periodic, independent, to have a deadline equal to period, and to have a constant response time in each period. RMA assigns priorities to tasks in reverse of their period (according to their rate). If a system satisfying RMA constraints is not schedulable using RMA, it is not schedulable using any other scheduling discipline. This fact by itself could be a sufficient motivation to use RMA as real-time scheduling framework.

To use RMA, we need to have a task model that specifies the task period, response time, and deadline or constraints on response time. In addition, if tasks have been assigned non-RMS priorities, such priorities need to be known too. This is the minimal information needed for making effective scheduling decisions. In order to apply RMA to our system, we had to identify concurrent tasks and for each task to determine its priority, period, response time, and deadline or constraint on response time.

We had full access to the design documentation, source code, and extensive execution traces produced from multiple use cases. Unfortunately design documentation often does not contain sufficient information regarding task structure. This is mainly because initial task structure is often changed at later stages of product integration and fine-tuning when design documentation is not actively maintained anymore.

Although source code contains task creation instructions and task priorities, it is not easy to determine from the source code which objects / functions are allocated to which task and how objects in a given task interact with objects in other tasks. Therefore we mainly focused our efforts on the analysis of execution traces.

5. RECOVERING RUNTIME ARCHITECTURE FROM EXECUTION TRACES

The raw data for architecture recovery consisted of multiple files of execution traces collected from different use cases. A trace file usually includes traces of different stages of a scenario, like connection setup and teardown, buffering of data, etc. To identify the segment of the use case that has all the relevant tasks active we used utilization maps (Figure 2).

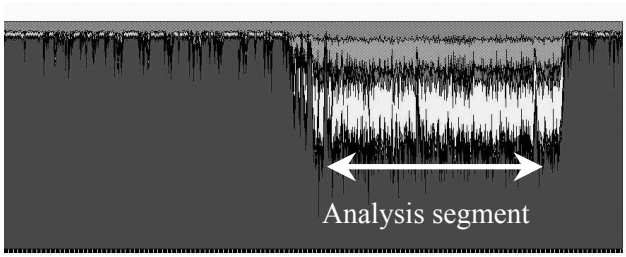

Figure 2 Finding the analysis segment from utilization map

A trace is produced every time a task is scheduled to run. To account for task preemptions, we had to recognize them in traces. The mere fact that a higher priority task T_{high} is executed right after a lower priority task T_{low} (Figure 3) does not indicate that the lower priority task was preempted. It is possible that the lower priority task T_{low} had completed its response before T_{high} was scheduled. To recognize preemption, our system produced an event at the end of each task's response. If such an event occurred at time T2, the task T_{high} was scheduled after T_{low} had completed its response. If such an event occurred only at the time T4, then the task T_{high} preempted T_{low}. This decision affects the calculation of response time and period of T_{low} task. If task T_{high} preempted T_{low}, then T_{low} response times [T1, T2] and [T3, T4] on the picture should be added to represent a single response time. If no preemption occurred, then [T1, T2] is one response time of T_{low}, [T3, T4]—another, and [T1, T3] is a period of T_{low}.

If a trace at the end of task's response is not available, task preemptions can be recognized by examining the sequence of tasks following T_{low}. If T_{low} is not rescheduled after it was preempted by T_{high} and before a lower priority task T_{lower} is scheduled then it can be assumed that T_{low} has completed its response before T_{high} was scheduled. It is also possible for a lower priority task T_{low} to invoke a higher priority task T_{high} as a part of T_{low}'s response. This situation can be recognized by studying message traces. If T_{low} sends a message to T_{high} before it was preempted and later T_{low} was rescheduled, we consider the action of T_{high} to be a part of T_{low} response.

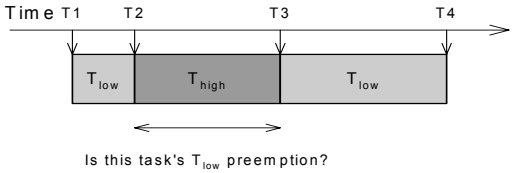

Figure 3. Task preemption

The analysis of task preemptions led us to recognize that the RMA model of what constitutes a response is limited in a number of ways: A response to an event may involve multiple tasks of different priority. However, the invocation of these tasks is deterministic in response to an event and thus has to be seen as a single activity. It is also possible that a response to an event may have multiple deadlines for different actions that constitute the response. In these cases, the task period cannot be considered the main deadline.

Once we have extracted from the traces the observations of each task's parameters such as invocations and response times, we had to decide how to process these data in order to arrive to a fair characterization of each task in terms of RMA model. Specifically, RMA expects each task to be characterized by a single period, deadline and response time possibly representing the worst case. In all the use cases we have analyzed such an approach is not practical because none of these use cases would be schedulable even though we knew that the system performed fine in reality. For example, in a simple voice call scenario where the average CPU load was lower than 30% and the system is clearly schedulable, the worst-case response time in a single task is larger than its worst-case period, which indicates that even this single task is not schedulable according to the analysis. Similarly, Table 1 shows the extracted data of a communication task for the use case of file transfer between a laptop and a server over USB

and GPRS. The worst-case period of the task takes 0.4 time units, while the worst-case response time takes 20 time units.

After some analysis we concluded that either our tasks are aperiodic with a wide distribution of inter invocation and response times or what is more probable the initial assumption that we adopted from RMA of tasks having a single period and response time does not hold in our case. However, it may be possible to identify a small set of typical ("peak") periods and response times of a task.

Table 1. Communication task response times and periods in file transfer over GPRS and BT

Worst case (longest) response time	20	Worst case (shortest) period	0.4
Average response time	1.6	Average period	20.5
Largest cluster response time	1.3	Largest cluster period	1.6
Response time clusters		Period clusters	
Time	Size	Time	Size
1.3	4352	1.6	1479
3.1	906	4.3	657
		10.6	562
		20.6	1024
		41.9	1221
		100.9	280

We developed a data-clustering algorithm that identifies presence of multiple clusters ("peaks") in observations of task periods and response times. We have applied a modified K-means algorithm with criteria for establishing new clusters. We have then developed a technique to merge clusters that did not exhibit sufficient separation. We found that indeed most of the tasks had several characteristic periods and response times. For example, a communication task (Table 1) has 2 response time clusters and 6 period clusters in the given use case.

Response time and period clusters need to be analyzed further. In many cases, it is impossible to perform a schedulability analysis based on just the largest response-time and period clusters. For example, the largest period and response-time clusters of the task in Table 1 show that this task would have used the processor at $1.3/1.6 \approx 81\%$ utilization. Since the system has other tasks as well, such a high utilization by one task would indicate that the system is unschedulable. However, as we know from practice, this is not the case. It may be possible though to identify pairs of compatible period and response time values. It is also essential to determine which parameter values of one task correspond to which parameter values of another task from the same featureset.

6. CONCLUSIONS

Understanding runtime software architecture in terms of featuresets, concurrent tasks and resource scheduling framework seems very useful both in terms of making concrete what are essential architectural decisions and how to analyze the effect of architectural decisions on software schedulability and performance.

Using execution traces to recover runtime architecture is a practical approach in view of the fact that design documents often become outdated and runtime information is hard to extract from source code.

Since featuresets do not overlap at runtime, they could be incorporated into the runtime architecture. Different schedules can be used for different featuresets. This is possible even when using a fixed priority operating system by allowing the operating system to perform infrequent priority changes of all tasks before scheduling the first task when featureset changes. Such an approach allows using optimal schedule for each featureset and improves overall schedulability of the system.

Execution traces may uncover inefficiencies in the design of runtime architecture. Typical examples are tasks with very short and very long periods that can occur in the same featureset. Fixed priority scheduling cannot assign a consistent priority to such tasks. Functionality needs to be reallocated. Another example is a single task that serves multiple independent event streams. Such a task will have more than one period from the independent events superimposed on each other. As a result such tasks do not have any characteristic period and cannot be scheduled using RMA framework. Functionality for serving independent streams of events must be allocated to different tasks.

Execution traces demonstrate that event responses may include multiple actions spread over multiple tasks. Unless such response has to meet multiple deadlines for different actions, the priority of tasks that belong to the multi-action response should be non-decreasing since the priority of activity as a whole is that of its lowest priority task and correspondingly the activity as a whole can be blocked by tasks with priorities higher than the activity's lowest priority.

Activities that do not share actions with other activities can be reallocated so that the entire activity happens in the same task. This may not work when actions have independent deadlines and thus the tasks that carry the actions may need to be of different priority.

7. REFERENCES

[1] G. Booch, I. Jacobson, J. Rumbaugh, *"The Unified Modeling Language User Guide"*, Addison-Wesley 1999.

[2] M.H. Klein, T. Ralya, B. Pollak, R. Obenza, M.G. Harbour, *"A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems"*, Kluwer Academic Publishers, 1993.

[3] J.W.S. Liu, "Real-Time Systems", Prentice-Hall, 2000.

[4] A. Ran "ARES Conceptual Framework for Software Architecture" in M. Jazayeri, A. Ran, F. van der Linden (eds.), *"Software Architecture for Product Families Principles and Practice"*, Addison Wesley, 2000.