Performance Assertions for Mobile Devices

Raimondas Lencevicius
Nokia Research Center Cambridge
3 Cambridge Center, Cambridge,
MA 02142, USA
Raimondas.Lencevicius@nokia.com

Edu Metz
ATI Technologies Inc.
1 Commerce Valley Drive, Markham,
ON L3T 7X6, Canada
emetz@ati.com

ABSTRACT

Assertions have long been used to validate the functionality of software systems. Researchers and practitioners have extended them for validation of non-functional requirements, such as performance. This paper presents the implementation and application of the performance assertions in mobile device software. When applying performance assertions for such systems, we have discovered and resolved a number of issues in assertion specification, matching, and evaluation that were unresolved in previous research. The paper describes a simple, but effective framework geared towards mobile devices that allows specification and validation of real world performance requirements.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; C.4 [**Performance of Systems**].

General Terms

Measurement, Performance, Verification.

Keywords

Mobile devices, assertions, performance.

1. INTRODUCTION

Assertions have long been used to validate the functionality of software systems [3][5][7][13]. Because assertions became a well-known and easy to use tool, researchers and practitioners tried to extend them for validation of non-functional requirements, such as performance. Perl [9][10] proposed to use assertions for validation of performance requirements and implemented such performance assertion system. Perl's work concentrated on post-mortem assertion checking using trace logs. Vetter and Worley [16] extended this work by proposing online validation of performance assertions. However, the designs of performance assertions in previous work contain a number of unresolved issues that hamper performance assertion adoption in practice: there is no mechanism for gathering and matching performance related events across process boundaries; no easily readable and writable assertion formulas integrated in the same programming language as the system under test; no general way to obtain data used in assertion formulas. Our work suggests ways to address these issues.

Mobile devices contain complex multitasking software systems subject to numerous performance requirements. Without performance assertions, performance requirements in mobile devices are often checked by hand or with ad hoc tools that analyze enormous trace log files gathered during system execution. Performance assertions in mobile systems allow simpler, faster and more precise testing and validation. We have adapted and extended previously described assertion frameworks to software in mobile devices.

Performance assertions allow to specify performance requirements, map them to the source code and validate them during execution. Performance assertions do not provide guarantees that performance constraints will be never violated. They are tools for testing, not for the formal verification. If the software executes without triggering the assertion, it successfully passes this test case. However, the assertion may still fail in another test case. If it triggers the assertion, the test case fails. Failing assertions provide immediate feedback to test engineers with localized report on the detected violation, which is a large improvement over the post-mortem trace analysis that is usually used to check performance constraints.

Section 2 describes the mobile device domain, performance requirements in it and how they can be handled using performance assertions. Section 3 describes the issues with performance assertions that we encountered and proposes possible ways to resolve them. Section 4 presents our approach for performance assertions in mobile devices including assertion framework implementation. Section 5 discusses application of performance assertions to representative performance requirements. Section 6 discusses the use of performance assertions in mobile software development process and explores the application of performance assertions in general software. The paper concludes with related and future work and conclusions.

2. MOBILE DEVICE APPLICATION DOMAIN

This section describes the application domain motivating our use of performance assertions. Although the ideas explored in this paper may have wider application, we are primarily concerned with embedded systems such as mobile phones or personal communication devices. Such systems have a number of characteristics that make them different from others and thus define an application domain.

Today's 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 as well as personal information management, including data synchronization with remote servers and desktop computers. Many mobile phones today are equipped with imaging devices and are capable of taking still images and video clips. The images may be sent over wireless networks to other phones or may be transferred to a remote server or a desktop computer for storage or forwarding. Mobile phones also have a number of local connectivity interfaces such as USB, IrDA, and Bluetooth that can be used for a variety of applications involving local data transfer, remote execution, and other types of interaction with surrounding computing resources. For example, a phone can serve as a wireless modem for a laptop computer over Bluetooth connecting it to wide area network over circuit switched data call or GPRS (*General Packet Radio Service*) packet data connection.

The above description shows that mobile phones host a range of communication-centered applications most of which have performance constraints at various levels. Some of these performance constraints are hard real-time, some of them are soft real-time, i.e., they affect the quality of operation perceived by users.

Lower layers of software implementing device drivers or protocol stacks have to satisfy constraints set in standards or hardware specifications. During GPRS session lower layer packets arrive every 10 ms. GSM (*Global System for Mobile Communications*) [4] level 3 signaling standard requires 500 ms response to any GSM Layer 3 message. GSM level 2 performance requirements require response to commands within 50 ms. GSM-WCDMA (*Wideband Code Division Multiple Access*) handover has 40 ms absolute constraint for completion.

Middleware and application layers need to satisfy usability constraints to optimize UI, minimize audio and video jitter, and so on. 10 or 15 frames per second of video should be rendered on the device. Opening a scheduled meeting in a calendar application should take less than 1 second.

In this paper we selected some performance requirements from lower layers, middleware and applications to use as motivating examples. These requirements are close to the requirements used for real products. We simplified the requirements and slightly changed their time constraints to avoid tying them to a concrete product:

1. *During GPRS session lower layer packets arrive every 10 ms.*
2. *GSM level 3 signaling standard requires 500 ms response to any GSM Layer 3 message.*
3. *GSM level 2 performance requirements require response to commands within 50 ms.*
4. *GSM-WCDMA handover has 40 ms absolute constraint for completion.*
5. *Screen redraw should take no more than 10% of the time needed to insert an appointment into a calendar application*
6. *Opening a scheduled meeting in a calendar application should take less than 1 second*
7. *Opening the calendar application should take less than 2 seconds plus 5 ms per each appointment in current month*
8. *Reading of a file should take at most 10ms multiplied by the number of blocks read and multiplied by the ratio of total and consecutive blocks in the file*
9. *Processing delay of audio frame through voice-over-IP (VoIP) stack should be less than 20 ms*
10. *Voice over IP call setup should be less than X ms*
11. *Deleting contacts in the phone book application should take less than 1 second plus 5 ms per each deleted contact*
12. *File copy rate from internal flash memory disk to flash card should be at least* MinRate *kb/s*

The requirements from this list are referred to in other sections of this paper by italicized numbers in parentheses: *(1,3)*.

Although some of the requirements listed are soft, catching their violations is still important in the testing process, since this allows improving the product before it is released to customers. Therefore notifications about failed performance assertions are very useful for the product development team. On the other hand, performance requirements on such complex systems as mobile device software are practically impossible to verify formally. Even if the verification were possible, it would likely show that the system violates the constraints in the worst case. For example, it is possible that the applications may fail to start in 1 second if a lot of system operations are scheduled at the same time. However, such worst-case examples usually involve highly unlikely scheduling of different processes and threads that occurs very infrequently in practice. Therefore performance assertions may be a good alternative to check the likely cases of execution during testing process.

3. EXTENDING PERFORMANCE ASSERTION SPECIFICATIONS

Performance assertions have been proposed and described in previous work [9][10][16]. We have attempted to use these assertions to validate performance requirements in mobile devices. During our analysis of performance assertion applicability to mobile devices, we discovered a number of issues with previous proposals. We extend the performance assertions to resolve these issues.

For assertions on mobile devices we considered *online* performance assertions, i.e. the assertions that are checked during the execution of the software system. Online assertion processing allows to halt the system or to notify the test engineer as soon as an assertion is violated. This shortens the testing time in case of assertion violations and provides additional clues for the causes of violations. In contrast, offline (post-mortem) assertion processing happens after the test. Even if assertion violations are found then, the test engineer may not remember some significant details that happened during the test, which were related to the failed assertion.

Programming languages with assertions (e.g. C [5], Eiffel [7]) usually provide online functional property assertions, which is another reason to follow the same model with performance assertions. Online performance assertions may have a higher overhead than offline assertions if the online processing of the assertion constraints costs more than storing the events into non-volatile memory. Therefore the costs of online processing and storing events need to be evaluated. This is possibly a more significant issue for performance assertions than for functional assertions, since online performance assertions change the performance of the same system that they are measuring. However, if performance assertions are not triggered in the system that is running slower because of the assertion overhead, there is additional assurance that the constraints will not be violated in a system with assertions removed. If assertions fail, there is a chance of a false positive due to assertion overhead, but the failure site should be investigated anyway, since it may have a too narrow margin to failure. Having said that, performance test engineers should be aware of the assertion overhead [8] and try to minimize it in testing.

Another important observation is that mobile device software often uses a multitasking client-server model (for example, Symbian OS [15] model). Performance requirements and therefore assertions in such model often cross process boundaries. This means that we have to allow programmers to specify cross-process performance constraints.

Keeping in mind that we are using online performance assertions and that cross-process assertions should be available, the following subsections describe in detail the issues that have to be solved to have a full-fledged performance assertion framework in complex software systems. We suggest a number of solutions to the issues raised.

3.1. Assertion specification

Although discussed in previous work [9][10][16], performance assertion specification is very important and needs to be revisited to provide a powerful and easy to use mechanism.

Performance assertions have some properties that make them different from functional assertions. Therefore it is impossible to just adopt the functional assertion specification model. Differently from functional assertions (e.g. in C programming language [5]) that are textually located in a single place of the source code and executed in a single "place" during program execution, performance assertions most of the time consist of multiple textual entities that correspond to multiple temporal events. Any assertions that have a timing constraint between two points of execution in a program need at least two entities: the beginning of the assertion and the end of the assertion. Vetter and Worley [16] use pa_start and pa_end constructs to indicate where assertions start and end. For more complex assertions, such as: *(5) "Screen redraw should take no more than 10% of the time needed to insert an appointment into a calendar application*", there needs to be more than two different events in the program execution. In this example, the assertion needs events that outline the beginning and end of each screen redraw procedure as well as the whole appointment insertion code. Only using these events the assertion can calculate whether the constraint was satisfied. Let us consider what information is needed for performance assertion specification. Based on these needs, we propose a way to specify assertions in section 4.

3.1.1 Event matching

pa_start and pa_end constructs by itself are only sufficient for simple assertions contained in a single non-recursive function. However, real-world performance requirements can span function and even process boundaries. Consider a performance assertion to validate the requirement *(6) "Opening a scheduled meeting in a calendar application should take less than 1 second*". The constructs for the beginning and the end of this assertion would belong to different processes. To be precise, the beginning of the assertion would be in a keyboard interrupt handler while the end would be in the calendar application. Alternatively, the beginning and end event markers could be placed into the window manager code, but even then they would belong to different functions and possibly threads.

Furthermore, the program may contain more than one performance assertion at the same time. Because of this, there has to be a way to match the correct pa_start construct with corresponding pa_end construct. A simplest solution is to match using a static performance assertion IDs:

```
int someCalendarFunction (…)
{
        …
        pa_start(CALENDAR_EVENT_OPEN, …);
        …
}

int someOtherCalendarFunction (…)
{
        …
        pa_end(CALENDAR_EVENT_OPEN, …);
        …
}
```

This approach provides a unique static assertion name (or identification). However, a unique static name is not sufficient at all times. There are assertions that need dynamic identification. For example, to assert that the time from a GUI object creation to its rendering should be limited, we cannot just use static assertion IDs:

```
int GUIObject::GUIObject (…)
{
        …
        pa_start(GUIOBJECT, …);
        …
}
```

```
int GUIObject::Render (…)
{
        …
        pa_end(GUIOBJECT, …);
        …
}
```

Multiple GUI objects may be created before rendering any single one of them, so the execution may have a number of pa_start(GUIOBJECT, …) events before a pa_end(GUIOBJECT, …) event. Since there is no identification of the object created, it would be impossible to match the correct pa_start event to the correct pa_end event.

Adding dynamic assertion IDs solves this problem:

```
int GUIObject::GUIObject (…)
{
        …
        pa_start(GUIOBJECT, GUIObjectID, …);
        …
}

int GUIObject::Render (…)
{
        …
        pa_end(GUIOBJECT, GUIObjectID, …);
        …
}
```

In this case GUIObjectID can be a this pointer to the object itself. In general, dynamic IDs can be object IDs, hash values or any other unique dynamic identifiers. Dynamic IDs may solve the event matching for requirements *(1, 2, 3, 5, 6, 8, 11)*[1].

Unfortunately, this solution is not always sufficient either. Consider the example of a timing constraint on the opening of a calendar application (*(7) "Opening the calendar application should take less than 2 seconds plus 5 ms per each appointment in current month*"). The opening of the application via clicking on its icon in the device generates a software interrupt. The interrupt handler has the information about the key pressed, but it does not have the information about the application that was started. So the interrupt handler may contain the following beginning of an assertion:

```
void keyboardInterruptHandler (…)
{
        …
        pa_start(START_KEYPRESSED, keyID, …);
        …
}
```

This pa_start identifies only the key that was pressed. On the other hand the calendar application may contain the following assertion end statement:

```
int CalendarApplication::Initialize (…)
{
        …
        // Application started
        pa_end(CALENDAR_APPLICATION, …);
        …
}
```

This end statement only identifies the launched application. Now, how can the keyboard event be matched with the calendar application initialization when the keyboard event handler does not know which application it starts and the application does not know which keyboard event started it? In the system execution there may be many keyboard events and even multiple application initializations. How is it possible to match the keyboard event with the corresponding application initialization? There has to be an intermediate event that provides matching for the two events. This could be a statement in the window manager that processes the key presses and starts corresponding applications:

```
int WindowManager::KeyPressStartApp (…)
{
        …
        pa_match(START_KEYPRESSED, keyID,
                CALENDAR_APPLICATION, …);
        …
}
```

In the example above, the pa_match event matched the key pressed and the application started. It is also quite possible that the window manager first processes the key press in one function and then calls another function to start an application. In this case additional matching events may be needed.

Additional matching events solve the event-matching problem for all our example requirements.

Perl [9] solves the issue of assertion beginning and end matching in offline analysis by using an interval system. In her analysis any interval between two events in a log file can have some performance constraint. Even in the interval system, the events have to have enough information for matching to occur. For example, if there is no information which START_KEYPRESSED event starts the calendar application, it is impossible to check the timing constraint on the opening of a calendar application even using interval system.

3.1.2 Assertion formulas

To calculate the timing constraints, the assertions have to have a way to specify them. The constraints, represented as expressions, may refer to data from assertion beginning events, intermediate events, as well as assertion end events. When these events take place in the same process, local and global variables, and object fields can be used to store the data. The assertion formula then can refer to the data in the same programming language in which the software is written. Consider the example:

[1] Whether a certain performance assertion solution is applicable to a certain example requirement is dependent on the way software is written, so we use expressions "may solve" or "may apply" throughout the paper.

```
int x;
int functionWithAssertion (…)
{
        …
        pa_start(ASSERTION1, …);
        …
        x = 10;
        …
        pa_end(ASSERTION1,
        (assertion_interval(ASSERTION1) < x * 5));
        …
}
```

In this case, the values for expression assertion_interval(ASSERTION1) < x * 5 are taken from variables, the assertion_interval helper function uses the timestamps obtained at pa_start and pa_end to calculate how long the code execution between the start and the end took, and the assertion formula can be compiled into the program code and calculated there. The timestamps can be obtained by calling a time function behind the scenes and storing the timer data into a global data structure.

This approach is very similar to functional assertions that use local and global variables, object fields, and constants available in the program spot where the assertion is placed. If additional information for functional assertions is needed, the assumption is that the programmer will provide it via helper variables and helper functions. This approach may work for performance requirements *(2, 11)*.

However, this approach does not work well when pa_start and pa_end occur in different processes, since the assertion formula in one process cannot refer to variables in another process. In this case, there are a couple of possible approaches. One possibility is to modify the language compiler or linker to correctly build the code to access data from another process via inter-process communication or shared memory. Another possibility is to explicitly encapsulate the access to the data from another process in wrappers, for example:

```
// process1:
int functionWithAssertionStart (…)
{
        …
        pa_start(ASSERTION1, …);
        …
        x = 10;
}

// process2:
int functionWithAssertionEnd(…)
{
        …
        pa_end(ASSERTION1,
          assertion_interval(ASSERTION1)
          < pa_access(PROCESS1, "x") * 5);
        …
}
```

Code 1: Data access with wrappers

Yet another approach would be to implement an assertion formula specification language independent from the underlying programming language and also implement some mechanism to extract data from processes that hold pa_start and pa_end events. This provides a nice separation of the assertion constraints and software.

The compiler/linker modification approach keeps the advantage of specifying the assertion simply in the underlying language. However, it is not feasible when compiler or linker cannot be modified. Language pre-processor could also be used for this approach if available.

The completely new assertion formula language provides the power to specify assertions that may not be easily specified in the underlying language. For example, adding such constructs as "always" or "there exists" may be difficult to express without breaking the underlying language rules. Even referring to the old value of a variable cannot be done without programmer help (adding an extra variable) unless already supported by the underlying programming language [7]. The drawback of the new assertion language approach is that the formula becomes different from the underlying language expressions and programmers may have trouble specifying assertions.

The wrapper approach is intermediate between two extremes above. Vetter and Worley [16] use it even in their single process assertions by specifying assertion formulas in the C/C++ printf format.

3.2. Data for assertions

Performance assertions usually correspond not to a single performance constraint, but to a parameterized performance constraint model. For example, a performance assertion for a file read response time is usually not specified as "*File should be read in less than 100 ms*", but rather as *(8)* "*Reading of the file should take at most 10ms multiplied by the number of blocks read and multiplied by the ratio of total and consecutive blocks in the file*". The file-read constraint may be dependent on the size of the data read, on the file block size and so on. These parameters have to be available for assertion validation. Some assertions need even more information. Assume that we need to check whether a program deletes all the files from a directory within certain time. The number of files in the directory may not be counted in the process that contains pa_start event or in the process that contains pa_end event. This number may be calculated only somewhere inside the file manager system. In assertions that occur in the same process, such information can be stored in global variables or objects. In assertions that cross process boundaries, we propose an assertion specific mechanism that sets values needed by assertions. Therefore we introduce a pa_set construct that allows setting a named parameter:

```
pa_set ("NumberFilesinDirectory",
        OptionalAssertionID, 20);
```

The variable can be set for a single assertion by specifying an assertion ID, or for all assertions. It is likely that assertions for requirements *(7, 8, 11, 12)* may need such pa_set constructs.

3.3. Time and assertions

The assertion model of pa_start, pa_match, pa_set, and pa_end events is tightly connected to the passage of time. The assertion is only checked when the pa_end event is reached. This has a number of important implications.

First, what happens if a certain pa_end event is never executed, even though the corresponding (via ID) pa_start event was executed? Since there is some constraint that specifies how much time should have passed from the pa_start event to the corresponding pa_end event, it seems that this constraint is violated if the pa_end event never occurs. In essence, the mental model here is that the pa_end event occurred infinitely far in the future from the pa_start, so the constraint was violated. However, the situation is not as simple as it seems. Without execution of the pa_end event and certain pa_match or pa_set events the system may not have enough information to evaluate the assertion constraint formula. For example, if variable x were never assigned in our example **Code 1**, the constraint formula cannot be calculated and it is impossible to claim that it was violated even if the program ran for a very long time after executing the pa_start event.

This means that the system cannot just raise an assertion if pa_end does not occur within a certain time frame. On the other hand, in continuously executing software systems it is impossible to wait until the program terminates to raise the assertion "pa_end corresponding to pa_start(ASSERTION1) did not occur".

Another issue with time is the ordering of the events. Obviously pa_end events that occur before matching pa_start events should be reported, since this should only occur as an unintended consequence of misplaced pa_end or pa_start events. Out of order pa_set events that set variables used in assertions are more dangerous. They can lead to hard to analyze assertion failures or system failures without assertion violations (false positives or false negatives) when the out-of-date pa_set value is used to calculate the constraint formula. One way to debug this issue is to collect the complete log file and manually go through the events in it – a labor-intensive approach. Unfortunately such errors are similar to programming errors in that they cannot be completely prevented or automatically detected.

Performance assertion constraints involve time expressions. Time is usually measured in seconds and obtained from the operating system or hardware timer functions. It is also possible to have constraints expressed in CPU instructions or cycles; however, hardware support is usually needed for data collection for such constraints.

The time in constraints may refer to absolute time between events or the *process* time, where only the time spent in a specific process (task or thread) is counted. Using process time in constraints requires a service for measuring such time. Such services are available in some operating systems and are possible to implement in others.

4. PERFORMANCE ASSERTIONS FOR MOBILE DEVICES

The previous section discussed some issues about performance assertions, their semantics and specification, and presented some solutions to them. In this section, we propose a design that is applicable in Symbian based [15] mobile devices, easy to use for programmers, and avoids some of the implementation issues. It contains a lot of the ideas introduced in the previous section. We also present the implementation of this framework.

We propose to use the pa_start and pa_end events with static and dynamic IDs. When dynamic IDs are not needed, they can be left null. Additional matching is provided using pa_match events. For the purpose of simplicity, we assume that pa_match event can tie only a single ID of a preceding event (pa_start or pa_match) and a single ID of a following event (another pa_match or pa_end).

We need to be able to specify assertions across processes. At the same time, we want to keep assertions in the underlying programming language. Therefore we propose accessing data needed for assertions through wrapper functions.

We specify assertion formulas in a tool separate from the analyzed source code, so that changing of the assertion formula does not require recompilation and rebuilding of the whole software system.

We propose that the system does not raise assertions on the pa_start events that do not have a corresponding pa_end event. However, we propose a feature that allows a performance engineer to see the list of open pa_start events and make decisions whether these should have ended.

We implemented the assertion checker as a library of pa_start, pa_match, pa_set, pa_end and other helper functions. In our implementation these functions pass the data needed for assertion formula evaluation to the assertion checker inside a separate process. In this sense, our approach is similar to semi-online approach proposed by Perl et al [10]. However, it performs a fully online assertion processing.

In online performance assertion analysis all open pa_start and pa_set events and their data need to be kept for assertion evaluation. In the general case, pa_start events not closed by pa_end events would create uncontrolled memory increase. In our domain, we expect to test use cases that are no more than 10 minutes long. So even if we generously assume 1 event every 10 milliseconds, only 10 * 60 * 100 = 60000 events would accumulate. Assuming 10 bytes per event, this only needs about 600kB of RAM – an acceptable amount of memory on a device. In our experiments much fewer events were collected and remain open for extended periods of time. We expect that to be true in production use too. All performance constraints we encountered up till now had a millisecond or second range restrictions even for the worst values of parameters. These data lead us to believe that the unbounded growth of events in memory is not an issue in the mobile device domain.

We also plan to use a "worst case" upper bound that limits the length of time open performance assertion events should exist. If the performance constraints should be handled in a few seconds for most parameter values, the performance test engineer can use their domain knowledge to specify a relatively high upper bound, such as 30 seconds or 1 minute. Such upper bound should not lead to false positive assertion violations, but could decrease the overhead of retaining and checking old open events.

We have implemented the framework described above for Symbian based mobile devices. The architecture of the implementation is shown in Figure 1. Assertion functions are called in the application code. These functions communicate the events to the back-end process that contains the assertion event database and the assertion checker. Occurring pa_start events are added to the assertion event database; pa_set events set the variable values; pa_end events trigger corresponding pa_start event retrieval and assertion evaluation. The assertion UI communicates with the back end to enable/disable assertion tracking and to display the ongoing and failed assertions.

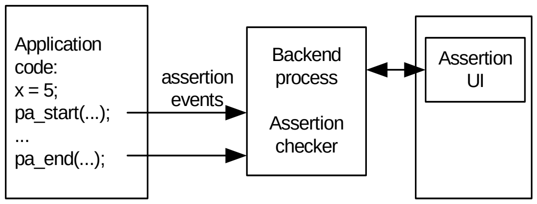

Figure 1. Architecture of the assertion framework

An assertion event takes about 3ms with more than a half of this time (1.7ms) used by the library function that obtains the time stamp of the event. To improve the efficiency of implementation, a more efficient way to obtain the time stamp is needed. The current performance assertion implementation can be used to validate high-level performance requirements that take at least 1 second – in such case pa_start and pa_end events will contribute 6% overhead. Requirements *(2, 6, 7, 8, 11, 12)* can be checked with the current implementation without high overhead.

5. PROGRAM REQUIREMENTS CHECKED VIA ASSERTIONS

We applied our performance assertion system to a number of performance requirements from the list of performance requirements mandated by mobile device project teams. This section describes three assertion examples in more detail.

"*Opening the calendar application should take less than 2 seconds plus 5 ms per each appointment in current month*" – this assertion belongs to a large class of performance assertions describing application startup. The application startup time is an important soft real-time constraint. Users expect the applications to start up immediately. However, for most applications the startup time cannot be characterized by a single constant, since it depends on various things that the application needs to process at the startup. Calendar is one of such applications, where the application starts in the current month view by default, showing all items in it. Therefore the assertion constraint depends on the number of appointments in the current month. To check this assertion, the pa_start event was added to the calendar initialization and pa_end event was inserted at the point where the display of the current month view is finished. Since the calendar application communicates with window server for display, the assertion had to be placed so that window server operations are finished before the pa_end event. A variable containing the number of items in the current month view had to be set via pa_set to transfer the number of appointments to the assertion evaluator.

"*Deleting contacts in the phone book application should take less than 1 second plus 5 ms per each deleted contact*" – another assertion that corresponds with a user-interface requirement. This assertion had to exclude the time taken by user confirmation of the delete command. The phone book application deletes contacts from a contact database. The database is managed in a different process; therefore, the placement of the assertion had to take into account such runtime architecture. The assertion had to obtain the number of contacts deleted via pa_set value.

"*File copy rate from internal flash memory disk to flash card should be at least* MinRate *kb/s*" – this assertion checks the requirement of the file transfer rate during a copy from one flash memory drive to another. To check this assertion pa_start and pa_end events were added to the File Manager process. The file size was obtained and provided to the assertion checker via pa_set.

These three assertions demonstrate the variety of requirements from real mobile device software development programs that were checked using performance assertions. Other requirements are similar to the ones described and can be easily handled by our framework of performance assertions. Since we used the assertions on a "work in progress" software release, some of the assertions passed and some failed.

6. DISCUSSION

Performance assertions seem to be a natural fit for specifying and validating performance requirements in mobile and embedded devices. Performance engineers have a lot of performance requirements that are difficult to specify in the software and to check during testing. Programmers need a simple, well-defined mechanism that is powerful enough to be able to specify various performance requirements and check them without a lot of testing work. Performance assertions offer such a mechanism. The proposal we presented resolves a lot of issues discovered in applying previous research results. It provides a needed performance assertion framework.

Performance assertions fit nicely into the quality assurance framework. Performance requirements may be created beforehand from requirements in the standards or from user-interface requirements. Such requirements can be directly translated into performance assertions. Additionally, software performance can be measured experimentally and the measurements can be used to create performance requirements (and therefore performance assertions) for future products. Such requirements for performance regression testing will have parameters to adjust them to new software and hardware platforms. These parameters can be inserted directly into assertions.

A large issue in applying performance assertions to all software independent of domain is that programmers may not care about their program performance if it is deemed "satisfactory". This highly subjective judgment may be sufficient for a lot of programmers. Even if the programmers want to use performance assertions, they may not have good performance requirements that could be used for constraints in the assertions. Finally, even if requirements exist – such as "*15 video frames per second for all MPEG movies*" – it may not be easy to decide where to place the assertions to validate the requirements. We believe from our experience that these issues do not apply to mobile device domain. In our domain, quantitative performance requirements are specified and checked in testing. However, these issues need to be resolved in the future to apply our work to the general software systems.

7. RELATED AND FUTURE WORK

Functional assertions have a long history, starting with a paper by Floyd [3] and continuing with a lot of seminal papers and books (Meyer [7], Rosenblum [13], etc.).

Performance assertions are less explored. Major work in this area includes Perl's thesis [9], Perl et al. paper [10], and a paper by Vetter and Worley [16].

Perl only analyzed offline assertion analysis. Perl's interval system is easily applicable in offline, post mortem performance analysis, since the whole performance event log file is available and can be analyzed and the relevant events matched. However, if performance assertions should be triggered online, during execution, the interval system is less attractive. Naive implementation of the interval system would have to keep all performance events from the very beginning of the program execution and process the whole log during each event. Even though there are techniques to optimize such processing, the overhead would likely to be prohibitive. Therefore a simpler matching approach, like the one presented in this paper, is needed.

Offline processing of an event log is powerful for checking statistical performance assertions. For example, assertions that use averages can easily be computed from a log, but are not easily defined for online validation. In case of offline computing the average is calculated over the whole log. In online analysis "the whole log" is not available until the system terminates. For systems that do not terminate and are continuously active, the average can be computed only for a fixed number of events in a sliding window. It is also possible to compute average (and other statistical metrics) for all events from the beginning of the system operation. This may result in the assertion violations early in the operation, since the log would contain very few events.

Perl et al. paper [10] presents a semi-online system that produces event logs, which are then analyzed on another computer with different log analysis tools. This approach still uses the offline log analysis algorithms. It is not a fully online performance assertion system and it does not work out the issues facing online performance assertion systems.

Vetter and Worley [16] only presented the idea of online assertions but did not consider any of the issues of applying them in complex large software systems.

Although a lot of performance issues are discovered using profilers, performance assertions have a different goal and reason. Comparing performance assertions to profilers is very similar to comparing functional assertions and debuggers. Both of them find issues in a program, but they use quite different approaches. Work by Reiss [12] is related to performance assertions, since it deals with response times of events in reactive systems.

In the future, the work by Andrews on log file analysis [1] possibly could be adapted to process performance logs. Aspect Oriented Programming [6] has been suggested as an ideal mechanism for implementing orthogonal concerns and could be applied to implement performance assertions at least on the platforms where AOP is available.

When considering a language for performance constraint specification, it might be worthwhile to consider prior work in Tquel [14]. Temporal logic [11] is another formalism that can be considered for inspiration on specifying performance constraints.

It may be worthwhile to explore the possibility to dynamically discover performance assertions similarly to functional and temporal invariant discovery done by Ernst and others [2] and Yang and Evans [17].

Our implementation could be improved by using low-overhead hardware timers to decrease the overhead of taking a timestamp. However, such timers are not always available. Timestamp operation is currently a major contributor to the overhead. After it is optimized, secondary contributors to the overhead might be exposed.

8. CONCLUSION

This paper explores the application of the performance assertions in large software systems, such as mobile device software. When applying performance assertions for such systems, we have identified and resolved a number of issues in assertion specification, matching, and evaluation.

We have described and implemented a concise, yet sufficiently powerful framework that allows specification and validation of real world performance requirements for mobile devices. We have applied this framework on a number of performance requirements from mobile device programs. We believe that such a framework will allow performance engineers to rapidly adopt performance assertions.

9. ACKNOWLEDGEMENTS

Authors want to thank Alexander Ran, Karel Driesen, and anonymous reviewers for comments on the earlier versions of this paper.

10. REFERENCES

[1] Andrews, J., Testing using Log File Analysis: Tools, Methods and Issues, *Proceedings of the 13th Annual*

International Conference on Automated Software Engineering (ASE'98), Honolulu, Hawaii, October 1998, pp. 157-166.

[2] Ernst, M. D., Cockrell, j., Griswold W. G., Notkin D., ``Dynamically discovering likely program invariants to support program evolution", IEEE Transactions on Software Engineering, vol. 27, no. 2, pp. 1-25, Feb. 2001.

[3] Floyd, R.W., Assigning Meanings to Programs, Proceedings of the Symposium in Applied Mathematics, Vol XIX, pp 19-32, American Mathematical Society, April 1967.

[4] GSM Association, http://www.gsmworld.com/, 2005.

[5] Kernighan B.W., Ritchie, D.M., *The C Programming Language*, 2nd edition, Prentice Hall , NJ, 1988.

[6] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.-M., and Irwin, J, Aspect-Oriented Programming, *Proceedings of the European Conference on Object-Oriented Programming,* 1997.

[7] Meyer B., *Object-Oriented Software Construction*, Prentice Hall, 1988.

[8] Metz, E., Lencevicius, R., Efficient Instrumentation for Performance Profiling, *Proceedings of the 1st Workshop on Dynamic Analysis,* 2003, pp. 143–148

[9] Perl, S. E., Performance Assertion Checking, *Ph.D. Thesis*, MIT, 1992.

[10] Perl, S. E., Weihl, W. E., Noble, B., Continuous Monitoring and Performance Specification, *DEC SRC Research Report 153*, 1998.

[11] Pnueli, A., "The temporal logic of programs", *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46-67, 1977.

[12] Reiss, S.P., "Event-based performance analysis", *Proceedings of the 11th International Workshop on Program Comprehension (IWPC-2003)*, pp. 74-83, 2003.

[13] Rosenblum, D. S., Towards a method of programming with assertions, Proceedings of the 14th international conference on Software Engineering, Melbourne, Australia, pp. 92 – 104, 1992

[14] Snodgrass, R., The temporal query language Tquel, *ACM Transactions on Database Systems (TODS)*, Volume 12 , Issue 2 (June 1987), pp. 247 – 298, 1987.

[15] Symbian OS, www.symbian.com, 2005.

[16] Vetter, J.; Worley, P.H., Asserting Performance Expectations, *Proceedings of the SC2002,* 2002.

[17] Yang, J; Evans, D., Automatically Inferring Temporal Properties for Program Evolution, *Fifteenth IEEE International Symposium on Software Reliability Engineering (ISSRE 2004)*, 2-5 November 2004.