BOSTON UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

Technical Report

Extending snBench to Support Hierarchical and Configurable Scheduling

by

GABRIEL PARMER

GEORGIOS ZERVAS

ANGSHUMAN BAGCHI

Submitted in partial fulfillment of the requirements of the course

CAS CS511 : Object Oriented Software Principles

Spring 2006

Abstract

It is useful in systems that must support multiple applications with various temporal requirements to allow application-specific policies to manage resources accordingly. However, there is a tension between this goal and the desire to control and police possibly malicious programs. The Java-based Sensor Execution Environment (SXE) in snBench presents a situation where such considerations add value to the system. Multiple applications can be run by multiple users with varied temporal requirements, some Real-Time and others best effort.

This paper outlines and documents an implementation of a hierarchical and configurable scheduling system with which different applications can be executed using application-specific scheduling policies. Concurrently the system administrator can define fairness policies between applications that are imposed upon the system. Additionally, to ensure forward progress of system execution in the face of malicious or malformed user programs, an infrastructure for execution using multiple threads is described.

Contents

List of Tables

List of Figures

List of Abbreviations

snBench	.....	Sensor Network Work Bench
SXE	.....	Sensor Execution Environment
STEP	.....	Sensor Task Execution Plan
QoS	.....	Quality of Service
JVM	.....	Java Virtual Machine
UML	.....	Unified Modeling Language
CPU	.....	Central Processing Unit
SRM	.....	Sensorium Resource Manager

Chapter 1

Introduction

The Sensor Execution Environment (SXE) includes a primitive method for scheduling that does not fully utilize computational resources and is susceptible to malicious or buggy opcodes. An ill-written opcode which does not return from its invocation can compromise the CPU resource, disallowing other opcodes access. Due to a single thread of execution, blocking on I/O causes poor processor utilization. Moreover, from the standpoint of the snBench application author [3], little or no flexibility is offered to ensure any Quality of Service (QoS) characteristics. Likewise, fairness constraints cannot be imposed on the execution of programs in the presence of other user's programs. These limitations are the driving motivators for the design decisions made for this group's software engineering project. These functional specifications have been discussed in detail in the following chapters.

The group management for this assignment was structured such that all members of the group were active in the discussion of the design of the code. After the design phase, Gabriel Parmer worked on the scheduler core code, Georgios Zervas worked on SXE integration code, and Angshuman Bagchi worked on the web page, and external documentation. Due to dependencies between the work of each member, everyone needed to have a stable codebase understanding so that parallel work could be completed. Thus, very early on in the design, three interfaces were chosen to be stable throughout the project, unless fundamental assumptions deemed them incorrect. Therefore the *IScheduler.java* interface for scheduler, the *IOpcode.java* interface for what constitutes an Opcode, and to a lesser extent, the *Task.java* abstract class were designed and the specifications produced very early on. These being the most pertinent interfaces to each member's duties, a large degree of work was accomplished in parallel.

Month	Task Outline
January	Research
February	Specification
March	Implementation
April	Testing
May	Final Submission

Table 1.1: Schedule of Tasks

We found this strategy of early isolation and freezing of interfaces to provide common-ground between members, to be quite satisfactory. An outline of the implementation schedule followed is given in Table 1.1. For a more detailed schedule, including weekly tasks the reader is referred to the the web page [1]. Documentation was done at every stage and the web page was updated on a weekly basis.

The remainder of this report is structured as follows: Chapter 2 outlines the functional description of the package implemented. It contains the Data Model of the system and provides a high level understanding of the design. The details of the design as well as the relevant Unified Modeling Language (UML) diagrams are provided in chapter 3. This is followed by an overview of how the source code of the package is organized in chapter 4. It also contains instructions to execute the package both as a stand-alone module as well as within the SXE framework. The next chapter, chapter 5, elucidates the testing methodology followed in this project. The report ends with a "post-mortem" or analysis of our project in chapter 6. Limitations and possible future extensions are discussed in chapter 6 as well.

It is pertinent to note here that the documentation of this project exceeds the length prescribed in the project guidelines. This is due to the nature of the application being developed. Although the package does not involve writing a lot of Java code, it affects the core SXE functionality. Therefore in this technical report we have attempted to document in detail all design and implementation decisions. This will help any future team of developers working on the SXE.

Chapter 2

Functional Description

A summary of the main functions and capabilities provided by our package follows:

Controlled Concurrent Opcode Execution: It is necessary to execute opcodes in separate execution contexts or threads as this minimizes the effects a malicious or malformed opcode can have on the entire SXE environment. If one opcode contains an infinite loop, other opcodes can still continue processing concurrently regardless. Further, if multiple opcodes can execute concurrently, then those that stall on I/O will have less of an effect on the overall CPU utilization as other threads will still run on the processor. However, to incorporate fairness or QoS, it is necessary that this concurrency be controlled by the Sensorium Resource Manager (SRM). A single snBench application should not be allowed to request a hundred threads of execution unless some trusted policy, installed by the SRM, allows it. Thus, albeit multi-threaded, the provision of computational isolation must be provided by the system. In our design, a $ThreadManager$ ensures this controlled concurrency.

Polymorphic Scheduling Policies: When designing a system, it is necessary to realize that all application usage patterns cannot be anticipated. Hence policies must be extensible to accommodate decisions concerning resource usage made by a domain expert. Further, to ensure some notion of QoS for applications, reservations for "administrator" users, for instance, policies specific to these concerns must be installed. An approach which will require such functionality is flow-types, where application-specific scheduling requirements can be specified. In view of the above requirements, polymorphic scheduling policies are implemented in our architecture.

Hierarchical Scheduling Model: It is not enough to allow a single scheduling policy to be polymorphic. Conflicts of interest will occur naturally in a system with different applica-

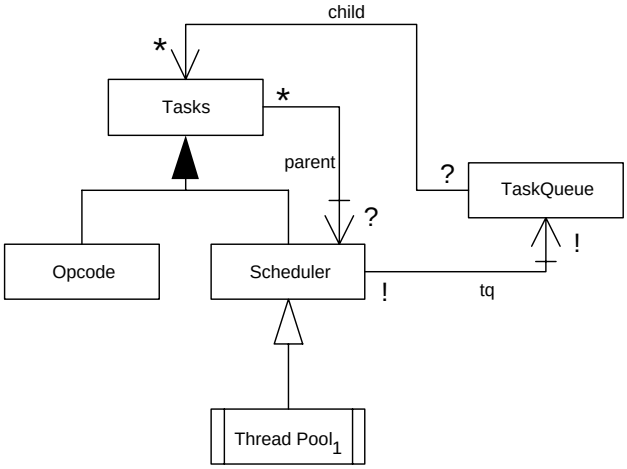

Figure 2·1: Data Model of the Scheduling System

tions having different requirements. For instance, an application may require a deadline-based scheduler while another might demand simple round-robin. To concurrently satisfy both requests, hierarchical scheduling must be employed. This is an active area of research in the System's and Real-Time community, and we simply provide the mechanisms with which hierarchies of schedulers can be constructed. In such a model, schedulers schedule "Tasks" while "Tasks" may be either other schedulers or opcodes.

The methods we used to implement this hierarchy are illustrated in the Data Model in Figure 2·1. This Data Model is subject to the following constraint:
There exists a t in $Tasks$ such that,

[t.parent = $\emptyset$ and for all t' in $Tasks$, t'.parent = $\emptyset \Rightarrow$ t'= t] and t *in ThreadPool*]

In other words the above constraint states that there is a task that has no parents, of which there is only one, and that task is the sole occupant of the $ThreadPool$. It should be noted that our project is not well suited to Data Models as there is very little data or data structures involved.

The package consists of schedulers and opcodes whereby a tree is formed with all internal nodes being scheduler tasks and all leafs being opcode tasks. Each scheduler task's children are either other scheduler tasks or opcode tasks, and the root scheduler has only one ancestor, the $ThreadManager$. The $ThreadManager's$ only child is the root scheduler task. This invariant structure is demonstrated in Figure 2·2. It is to be noted over here that the above

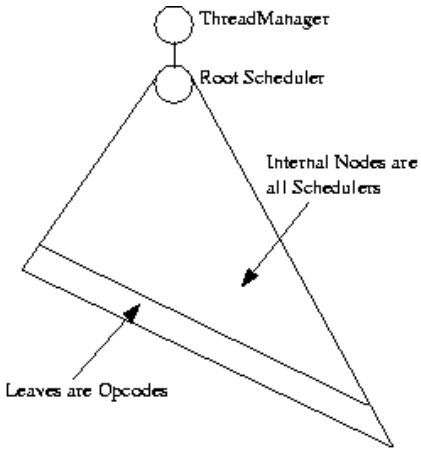

Figure 2·2: The Hierarchical Structure

invariant is maintained by the Java type system.

Of primary importance is the control flow between the *ThreadManager* to the opcodes. This is the method for execution of an opcode and includes a traversal of schedulers between the *ThreadManager* and the opcode. The schedulers, of course, have the volition to choose which opcodes to execute when they are invoked. This flow of control is illustrated in Figure 2·3.

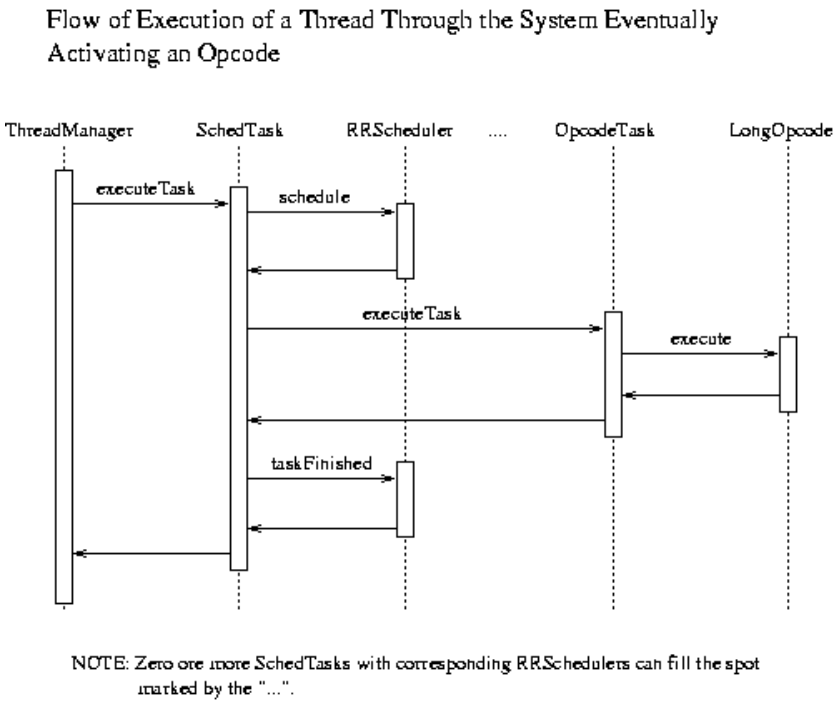

Figure 2·3: The Sequence Diagram

Chapter 3

System Design

The code base for hierarchical schedulers exists in the "edu.bu.cs511.p5" package, i.e. in "/src/edu/bu/cs111/p5/" in the directory tree. This code base includes the functional and logical source code providing the hierarchical scheduling framework. These are in the files:

1) IOpcode.java
2) IScheduler.java
3) Task.java
4) SchedTask.java
5) OpcodeTask.java
6) ThreadManager.java
7) RRScheduler.java
8) ProportionalScheduler.java
9) FPScheduler.java
10) GenericScheduler.java
11) SchedData.java

In addition to the above files, some code which allows the framework to be tested independently are in:

1) Main.java

2) SchedHierarchy.java

3) LongOpcode.java

Finally, the code used to produce the final presentation's demo is in:

1) SchedulerDemo.java

2) DemoOpcode.java.

The SXE directory structure remains unchanged, and we discuss code changes later in this section. A UML diagram depicting the relationship of all classes in the framework can been seen in Figure 3·1.

3.1 The Scheduler Framework Design

The Design Patterns employed in the Scheduler Framework are:

1) *SchedulerHierarchy*: Implements the Factory design pattern described in page 371 of Liskov's book [4]. It returns Tasks corresponding to scheduler/opcode and parent scheduler arguments.

2) *ThreadManager*: Implements the Singleton design pattern described in page 378 of Liskov's book [4]. Only one thread creator must exist.

3) *SchedTask*: Implements the Strategy design pattern described in page 388 of Liskov's book [4]. Here tasks are an interface allowing the functional execution of a task from a parent, *SchedTask*.

4) *OpcodeTask*: Implements the Command design pattern described in page 388 of Liskov's book [4]. Opcode execution can consist of any opcode functionality thus no assumptions are made about the behavior or purpose of opcode execution.

The entire Scheduler hierarchy which is a hierarchy of "Tasks" follows a Composite Design pattern (page 390 of Liskov's book [4]) where "Component" nodes are of the type $SchedType$ and "leaf" nodes are of the type $OpcodeTask$. The invariant that schedulers are component nodes and opcodes are leafs is maintained by Java typing as the parent Task of all tasks must be a $SchedTask$. Each of these nodes is traversed using a Visitor design pattern (page 393 of Liskov's book [4]) where the visitor interface for $SchedTasks$ is defined by $IScheduler$ and the visitor for $OpcodeTasks$ is defined by $IOpcode$. This was the most important and useful design pattern employed.

3.2 Integration into the SXE

From the project's onset we envisioned the integration between our code and the snBench environment as a cross-cutting operation. That is to say, before we even began coding we identified the possible points of integration with the SXE. By studying and understanding the provided software package we then set out to design our scheduler with these constraints in mind.

This proved to be a wise choice as on one hand it allowed us to develop, document, verify and validate our code in isolation, a good software engineering practice. On the other hand when we had established enough confidence in our code we were able with minimal, precise incisions on the *snBench* code to replace the existing scheduler with our artifact. For problems that arose during the integration process, we were able to easily pinpoint the fault to the integration process itself, as the schedulers had already been tested.

The integration was performed within the $sxe.GraphEvaluatorThread$ class which was also previously responsible for scheduling opcodes. The first task was to wrap graph nodes, as defined by the $step.Node$ class, by the new class $SXEOpcode$ acting as an *Adapter* between the existing environment and our scheduler. This was required as our scheduler was designed to schedule objects that implement the $IOpcode$ interface and not $step.Node$ objects. As the rest of the *snBench* environment didn't need to be aware of this encapsulation, the $SXEOpcode$ class was defined privately withing $sxe.GraphEvaluatorThread$.

The second incision was made at the point where scheduling decisions are made, that is within the *run*() method of *sxe.GraphEvaluatorThread*. The old code called *doIteration*() to pick up an enabled opcode and execute it. We substituted this method with a *doSchedulerIteration* method whose responsibility is to instead invoke the scheduler which would then decide which opcode was to be executed next.

With these two simple, understandable, minimal amendments we were able to utilize the functionality of the scheduler from the SXE without exposing either package to the internals of the other: the scheduler didn't need to be aware of actual substance of the opcodes it is scheduling as long it implemented the *IOpcode* interface and the SXE environment didn't need to be aware of the implementation of the actual scheduling policies.

An outline UML diagram is provided if Figure 3·1. It provides a birds-eye-view of the modules implemented in the system and the point of integration of the package with the SXE.

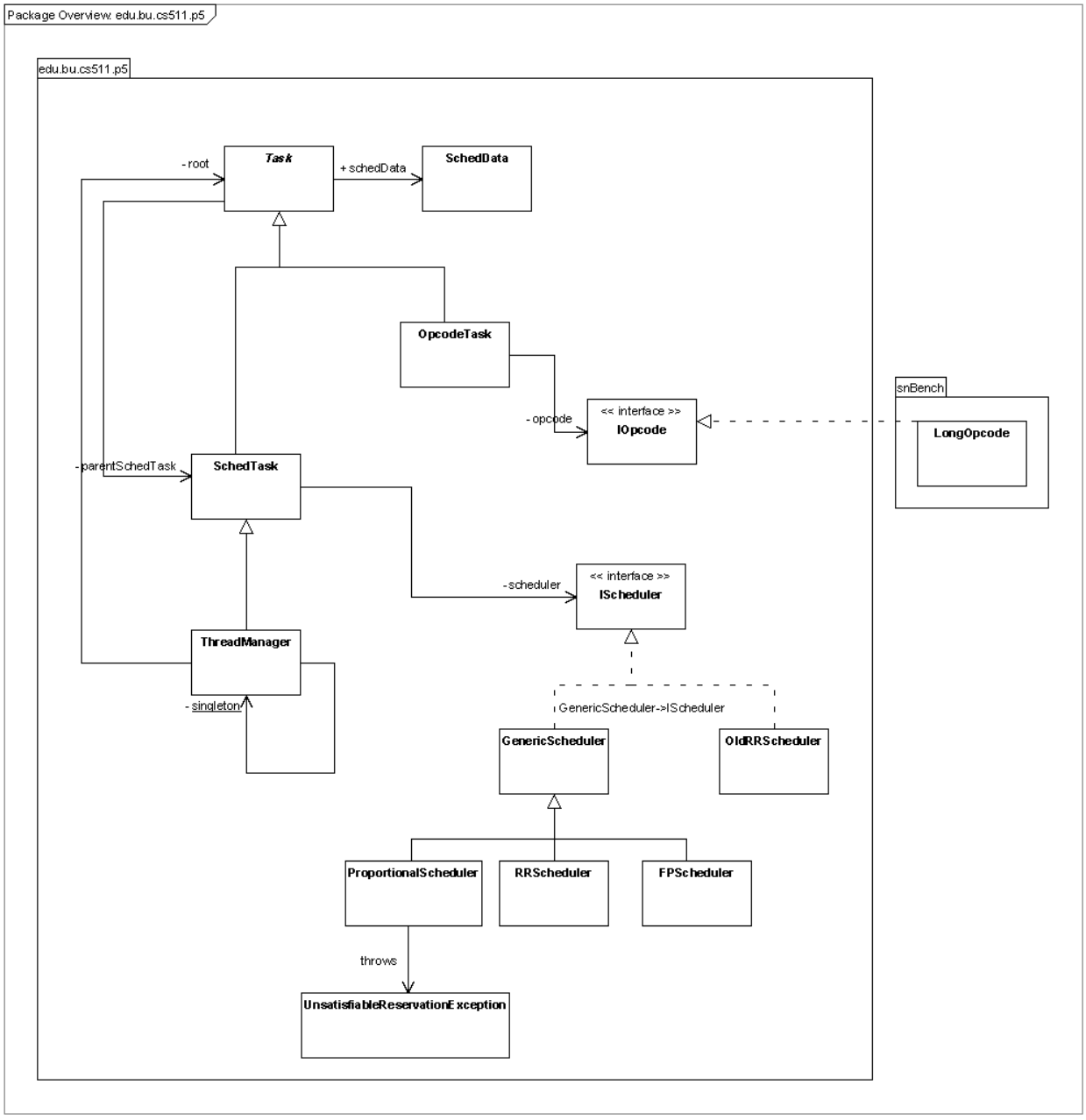

Figure 3·1: The UML Diagram of the Scheduling System

Chapter 4

Execution: Instructions and Examples

This chapter contains the instructions to compile and execute the scheduling framework. The package implemented by this team can be implemented as both a stand-alone module, as well as within the SXE. The details of these are enumerated in the following sections.

4.1 Executing the Scheduling Framework as a Stand Alone Module

This section describes how to compile and run the package as a stand alone module. To compile the package, make sure you are running in a UNIX-style prompt in the "src/" directory and issue the following instruction:

```
$javac edu/bu/cs511/p5/Main.java
```

The first test checks to make sure that basic operations for scheduling work fine. These are enabling, re-enabling, task removal and re-execution, and non-blocking execution. To execute this test, run:

```
$java edu.bu.cs511.p5.Main
```

An introduction to the program and a header specifying what the scheduler hierarchy is, will be displayed. Tasks are denoted by a pair of characters: First a number for the task number, and then a character for the scheduler that task runs under. A series of tests are carried out. When tasks are executed, outputs like the following are printed out:

```
$1a2a4b6c3a7c5b8c
```

The above display translates to "first task 1 which is under scheduler a runs, then task 2 under scheduler a, then task 4 under scheduler b, etc...".

The second test is used to demonstrate that the schedulers are working properly and allocating appropriate amounts of CPU time to each task. To run this test, type:

```
$java edu.bu.cs511.p5.Main test_allocations > output
```

After an amount of time (the longer the time, the more accurate the results) press *CTRL-C* to stop the program. A file called "output" has been created that contains a sequence of executed opcodes. Compile the "TestTaskPercentages.java" file:

```
$javac edu/bu/cs511/p5/TestTaskPercentages.java
```

Execute this program with the trace file of task percentages:

```
$java edu.bu.cs511.p5.TestTaskPercentages output
```

Something similar to the following will be printed out:

```
Unique characters are: 1a24b6c7583
Their frequency is:
1: 1967 (0.3456334563345633)
a: 2817 (0.49499209277807066)
2: 425 (0.07467931822175365)
4: 587 (0.1031453171674574)
b: 1173 (0.20611491829204007)
6: 1195 (0.20998067123528377)
c: 1701 (0.2988929889298893)
7: 253 (0.04445615884730276)
5: 586 (0.10296960112458267)
8: 253 (0.04445615884730276)
3: 425 (0.07467931822175365)
Total amount of characters in output: 5691
```

It can be confirmed that these values for the fractions of CPU allocations made to tasks, match the given scheduling hierarchy and the execution plan of opcodes through the system is correct. For instance, it can be seen by running "java edu.bu.cs511.p5.Main" and observing the scheduler hierarchy, that the "a" scheduler is allocated using the proportional scheduler 50% of the CPU, and that the test confirms that it received 49.49%. The expected values are perhaps slightly deceptive when multiple threads come into play: Scheduler "a" is a Fixed priority scheduler where task "1" has a higher priority than "2" and "3". Thus one would assume that "1" would always receive the CPU when scheduler "a" chooses a task to run. However, results show that "2" and "3" receive about 7.46% of the CPU each. This is due to the fact that "a" can request up to two concurrent threads. If it requests the threads at the same time, then the first thread will execute "1", but the second will choose between "2" and "3". Regardless of this , analyzing the results yields the fact that the scheduler's do schedule correctly.

An interesting conclusion can be drawn. In single-threaded cases, it is simple for schedulers to make the correct allocations, but as the number of threads increases in the system, the guarantees of the schedulers weaken due to the fact that two threads cannot concurrently execute the same opcode. Additionally, if the reader wishes, she can run the demo seen with the final presentation by executing:

```
$javac edu/bu/cs511/p5/SchedulerDemo.java
```

```
$java edu.bu.cs511.p5.SchedulerDemo
```

Pressing any key will step through the stages of the demo.

4.2 Executing the Package within the SXE

Instructions to compile and run the package within the SXE are provided in this section. Here "BASEDIR" refers to the directory where the project package has been extracted. We assume that $Java1.5$ and $ApacheAnt1.6$ are installed on the machine where the project will be compiled.

Compilation takes place in two parts. First, the scheduler has to be compiled and packaged in a "jar" file. This is performed by running the following command in "BASEDIR":

```
$ant -f cs511.xml clean
$ant -f cs511.xml all
```

The output of this process is the file "cs511.jar" which contains the scheduler. Next, we need to compile snBench.

```
$cd snBench/src
$make clean
$make
```

This compiles the snBench environment with support for the new scheduler.

To run the SXE one needs to follow the standard instructions and just add the new jar file to the classpath. We assume that the reader is in "BASEDIR/snbench/src" before running the following commands.

```
$java -cp .:jmf.jar:../../cs511.jar sxe.Server http://localhost:8080 NO
```

Once a graph is posted, in this case graph number 3, using the standard mechanism execute,

```
$java sxe.Poster http://localhost:8080


COMMAND         Does...
post            Posts a STEP file to specifed SXE
delete          Deletes a STEP node from specifed SXE
quit            quits
:post
Got [post]
Please specify the STEP file to be uploaded.
[0]     ./testsuite/framegrabber.repeating.step.xml
```

```
[1]     ./testsuite/temperature.step.xml
[2]     ./testsuite/math_and_string_ops.step.xml
[3]     ./testsuite/counter.step.xml
[4]     ./testsuite/sxe_info.xml
[5]     ./testsuite/basic_trigger.step.xml
[6]     ./testsuite/sxe_10.0.0.5.xml
[7]     ./testsuite/framegrabber.step.xml
[8]     ./testsuite/level_trigger_2.step.xml
[9]     ./testsuite/sxe_poster_local
[10]    ./testsuite/tester.java
[11]    ./testsuite/Makefile
[12]    ./testsuite/sxe_localhost.xml
[13]    ./testsuite/post_math
[14]    ./testsuite/counter2.step.xml
[15]    ./testsuite/tester.class
[16]    ./testsuite/framegrabber_320.step.xml
[17]    ./testsuite/factorial2.step.xml
[18]    ./testsuite/socket_localhost-8080_startsecond.step
[19]    ./testsuite/fg_temp_repeat.step.xml
[20]    ./testsuite/factorial.step.xml
[21]    ./testsuite/socket_localhost-8081_startfirst.step
[22]    ./testsuite/pairtest1.step.xml
[23]    ./testsuite/CVS
Specify a file number:3
```

One should see the following output from the SXE on the command line which indicates proper operation.

```
Content-Length: [751]
Serving [/snbench/sxe/node/]
```

```
Checking [text/xml] and [text/*]...
Got exact match: [1.0]/1.0
Returning 1.0
[CS511 Debug] New opcode to be added to the run queue: TriggerHead
[CS511 Debug] New opcode to be added to the run queue: not
[CS511 Debug] New opcode to be added to the run queue: equals
[CS511 Debug] New opcode to be added to the run queue: 0
[CS511 Debug] New opcode to be added to the run queue: lte
[CS511 Debug] New opcode to be added to the run queue: cond
[CS511 Debug] New opcode to be added to the run queue: isnil
[CS511 Debug] New opcode to be added to the run queue: lte1
[CS511 Debug] New opcode to be added to the run queue: start_value
[CS511 Debug] New opcode to be added to the run queue: subtract
[CS511 Debug] New opcode to be added to the run queue: lte2
[CS511 Debug] New opcode to be added to the run queue: 1
class sxe.core.not
[CS511 Debug] execute lte2
[CS511 Debug] execute cond
Evaluated: cond
[CS511 Debug] execute lte
[CS511 Debug] execute start_value
Evaluated: start_value
...etc...
```

Chapter 5

Quality Assurance

A majority of the project could not be checked with traditional Blackbox testing largely due to the fact that many of the complex aspects of scheduling occur in side-effects, such as the correct execution of the *next* task for a scheduler. Thus, where possible, strict black-box testing was used to make sure that just terminated tasks returned true to $isTerminated$. This analysis was limited by the nature of the project (schedulers correct functionality can only be measured by side-effects). Therefore to test the MODIFIES clauses of our methods we implemented the approach outlined in the following paragraphs.

To test the hierarchy itself, to see if schedulers can be added to schedulers, opcodes to schedulers, and the entire scheduler hierarchy to the $ThreadManager$, features of the *Composite* design pattern were utilized. Stub classes were used for the $Visitor$ class which defines how components and leafs are traversed and visited. These stubs were defined to be as simple as possible. The $IScheduler$ stub was a primitive RR scheduler which requested a concurrency level of the sum of its children's concurrency levels. Similarly, the $IOpcode$ visitors simply printed out their id and had a requested concurrency level of one.

The first aspect to test is to verify if the correct control flow through the hierarchy was maintained from the $ThreadManager$ to the *Opcodes*. When aligned in different configurations (some of which were demonstrated in the midterm presentation), one could ensure that the correct tasks were being visited at the correct times. By obtaining a sample trace of the execution of tasks, its valid constitution could be verified. In addition to typical cases, edge cases were tested. For example, schedulers with no tasks registered for them (thus they should request no threads and not execute) and a large number of children tasks. Small hierarchies where there was only a root scheduler and opcode children where tested as were

deep hierarchies with many schedulers under other schedulers.

The second aspect of the correct functionality was that, given requests from the opcodes percolating up the hierarchy to the *ThreadManager*, the correct requested concurrency levels were maintained while the correct number of threads were pushed into the system from the *ThreadManager*. For this, scheduler stubs were inserted into the system which always requested a given amount of concurrency and it was ensured that the *ThreadManager* spawned a correct amount of threads from its thread pool. Boundary cases tested were:

1) Schedulers which requested no threads.

2) Schedulers which requested many threads but that had parent schedulers that requested fewer (thus the child should receive that smaller number).

3) Schedulers that request more threads than the thread pool can sustain (thus the *ThreadManager* should limit the number of threads in the system on its own, independent of how much higher the request level is than their threshold).

Glass box testing was then carried out. There was surprisingly little at this phase as execution within the hierarchy was largely deterministic from a task level. Most of the conditions which would have to be tested were in Schedulers, described in the next paragraph. Black box testing was augmented mainly with tasks adversarial to the *ThreadManager* to ensure that no concurrency issues arose and that the correct number of threads were being released into the system given concurrency requests from the root scheduler. Many bugs were discovered here as concurrency is notoriously difficult to debug. Tests mainly included the root scheduler which would quickly oscillate between requesting no threads to requesting more than the thread pool would be willing to release, and testing all intermediary values for number of threads requested.

The main testing was carried out on schedulers. Initially, it was necessary to test on a very fine-grained basis to determine if a particular scheduler was choosing a task to execute correctly. To increase the amount of knowledge of the control flow through the system and to help in debugging, a textual representation of decisions made by schedulers, including

information regarding when tasks are executed and which schedulers are invoked was displayed as a human-readable **trace**. The results can be validated to be correct given semantic knowledge of specific schedulers i.e. one can check if the schedulers are acting correctly from the traces. These traces can be enabled by setting $debug = true$ in the $Task$ class. It was simple to observe the correct or incorrect functionality of schedulers based on which task they chose to execute and which they had chosen in the past. Again, typical cases where chosen as well as boundary cases such as:

1) Scheduling only one task

2) Scheduling many tasks

3) The immediate addition and deletion of tasks from scheduler queues

4) The increase of requested concurrency from children

5) The decrease to zero of requested concurrency.

Glass box testing followed. The amount of tasks asked to schedule, tested loops within schedulers as they routinely looped through all the tasks in the system. Conditionals were tested in two ways; oscillating the number of tasks schedulers must schedule and focusing on active insertions and deletions. This was important because many conditions are dependent on stale (already deleted) tasks still being in the data structures of schedulers and on there being no tasks in the data structures. Further, the concurrency requested by children was greatly oscillated to check for:

1) The correct requesting of threads from the scheduler's parent based on it's children's requests

2) The concurrent access of data structures maintaining concurrency levels to test for race-conditions.

Although these tests provided us with reasonable confidence that the concurrency logic and the data structure behavior of schedulers were correct, it was difficult to determine if the

schedulers were actually choosing the correct tasks to execute at the correct times. This was partially due to the rather verbose output of the traces. Hence a different style of tests was devised. These tests measured the fraction of execution time given to tasks received in a scheduling hierarchy. Note, for this test we assumed that there would only be two levels of schedulers, the root scheduler and a scheduler for each snBench application which would contain all the opcodes for that application. Given that we could create the hierarchy and utilize all forms of schedulers viz. proportional share, fixed priority and round-robin, we could infer what percentage of the CPU time tasks should receive. For example, between two tasks of different fixed priority, the higher priority should get 100% of the CPU. In another case given three tasks requesting 20%, 30%, and 50% of the CPU respectively, under a proportional share scheduler, confirm that they get those percentages. Within this testing framework, we attempted many permutations of scheduler hierarchies with different numbers of opcodes in each. In every case the tests confirmed that the opcodes received the correct percentages of CPU time. These results give us reasonable confidence to presume that each scheduler does in fact schedule its tasks as defined by its specification.

Chapter 6

Post Mortem

After reading the preceding chapters we hope that the reader is convinced that our endeavors in this project were successful. Our design process proved robust enough to withstand initial faults in design as detailed in Chapter 3 and the following section. This phenomenon of revising the design given implementation constraints is faced in every software engineering scenario and is testament to our belief in the spiral model of development. The early definitions of important interfaces enabled a large degree of parallelism among the team members. This not only helped us focus on sub tasks independent of the whole architecture but also helped in testing components independently as mentioned in chapter 5.

The next section enumerates our initial preemptive approach of setting thread priorities in Java. Section 6.2 discusses some of the design revisions we had to do while implementing various schedulers. Section 6.3 gives a few pointers to enhance our package. The last section of this chapter concludes this technical report.

6.1 Initial Approach

Our initial approach to tackling the SXE scheduling problem was to attempt to design a time slice-based scheduler. That is, we wanted to implement a scheduler that could in principle allocate specific amounts of time to opcodes for execution. The actual policy with which time would be divided between competing opcodes viz. round-robin, RMS, EDF etc. would have been implemented as a specific strategy of a base scheduler object. Having said that, the policy of the scheduler doesn't impact the rest of the discussion. Clearly, for this preemptive form of scheduling to work, we would have needed to implement a dispatcher responsible

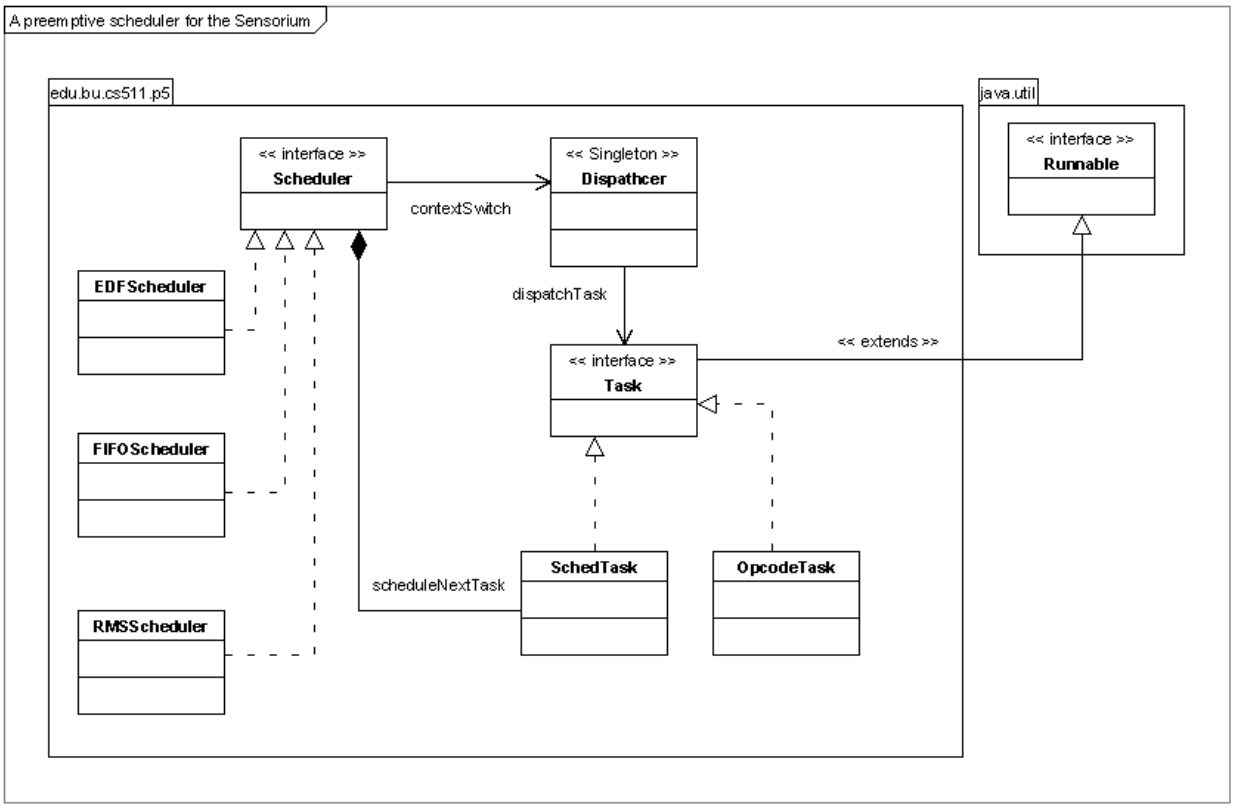

Figure 6·1: UML diagram of the initial Preemptive Scheduler

for stopping threads whose time slice had expired and moving ready threads from the ready queue to the run queue. A high level view of our initial conceptualization of the system is depicted in Figure 6·1 in UML notation.

The natural next step was to investigate the language, Java in our case, support for achieving this task. The first and most obvious possibility was using the $Thread.suspend()$ and $Thread.resume()$. However, as we found these methods are potentially dangerous to use and in fact they have been deprecated in the latest release of Java. Having decided, for reasons beyond our control, against using the aforementioned method we had to look for alternatives. What we came up with was the idea of manipulating the $Thread.setPriority(int)$ method to achieve the desired effect, possibly sacrificing a bit of time accuracy but preserving the ability to preempt through the dispatcher. According to the official Java documentation for the Thread class [2] *"Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority."* This gave us the following idea:
Utilize three different priorities, let's call them:

1) STOPPED(0)

2) RUNNING(9)

3) PRIVILEGED(10)

These would respectively correspond to opcodes in the ready queue, opcodes that are currently executing and the dispatcher itself. Scheduling would be achieved by first starting the dispatcher at the highest, PRIVILEGED priority. The dispatcher would in turn Spawn a number of threads with RUNNING priority and then $wait()$. Whenever an opcode would finish its work it would $notify()$ the dispatcher which would then become the ready thread with the highest priority. This would prompt the JVM scheduler to run our dispatcher being the ready thread with the absolute highest priority. The dispatcher would in turn dispatch another opcode. Clearly, in this model the loss of time accuracy comes from the fact that not all opcodes take exactly the same time to run but we deemed this a necessary sacrifice in order to avoid suspending and resuming threads.

Unfortunately, accuracy turned out to be the lesser of our problems. While conducting simple scheduling tests by starting a few threads and manipulating their priorities we were surprised to find inconsistent and erratic behavior which hardly resembled what we expected. It is rather hard to characterize the observed behavior but it definitely seemed that it would be impossible to guarantee any sort of predictable execution sequence. Matters were complicated even further when we tested our scheduling code on different platforms for example, Linux and Windows. We got completely different results. Hence we decided to drop this approach as without any guarantees as to when the dispatcher would run, the whole point of designing a preemptive scheduler became moot. This failed approach reflects the spiral development model and signifies the fact that in reality the design needs to be revisited several times due to implementation issues.

We do not consider pursuing this approach as time wasted. It gave us a flavor of the project and an opportunity to get out fingers dirty. The insights gained from this strategy helped us in designing the system described in chapter 3.

6.2 The Design of Schedulers Revisited

The design of schedulers followed the spiral development model. The initial design which directly leveraged the *IScheduler* interface to define the code for a *RRScheduler* (round-robin scheduler which can now be viewed for posterity in *OldRRScheduler.java*) was sufficient. However it did not allow a great degree of code re-usability when other schedulers were introduced. In essence, the *RRScheduler* code had to be replicated across new schedulers and specific logic changed to accommodate the new scheduling policies. Given the large possibility for re-usable code that was not being leveraged, we decided to expand our type-hierarchy to contain specific families of schedulers. All the schedulers that we implemented, namely round-robin, proportional progress, and fixed priority require three queues:

1) A run queue, or tasks which are ready to execute, but aren't allocated their maximum requested concurrency allocation

2) A running queue which contains threads which are running at their maximum thread allocation

3) A waiting queue, or a queue containing Tasks that have not been enabled and currently request no concurrency.

Thus a *GenericScheduler* abstract class was made which abstracts any scheduler which uses these data-structures. All methods besides constructors, are defined in this class and can be overridden by subclasses. In this manner, the *GenericScheduler* is an instance of a *TemplateDesignPattern* described in page 369 of Liskov's book [4]. Each specific scheduler then implements only the methods which require specific functionality as defined by its policy. In this way, all three schedulers went from being approximately 2000 lines of code cumulatively, to being less than 1000 cumulatively.

6.3 Future Enhancements

Throughout the course of this project, we have developed a general and extensible framework for hierarchical scheduling. Additionally, we have provided the integration required to allow

opcodes in an SXE to be scheduled given this framework. However, because the SXE does not currently support the well-defined retrieval of flow-types for snBench applications, or for opcodes, it is not possible to configure the scheduler hierarchy in an arbitrary manner. Though the capability and ability exists in the scheduler hierarchy framework, additional work is required to create well-defined flow-type interfaces. This will enable the SXE to take advantage of the full generality afforded by our project. An addition of this code to the SXE would allow full realization of our code's potential in a snbench environment.

6.4 Conclusion

Our objective of taking this course was to get a flavor of software engineering and make contributions to an existing code base of relatively large size. Although this project does not entail a very large amount of coding, it does involve understanding the existing architecture and making fundamental structural changes. We believe that both understanding some one else's code and writing code that is understandable by others are challenging. We believe that we have achieved both these objectives. The preceding chapters have displayed our successful software engineering methods in tackling a large project. We hope that this project will be of use to a future team of developers and the snBench framework in general.

References

[1] http://cs-people.bu.edu/abagchi/oosp-project.htm.

[2] http://java.sun.com/j2se/1.5.0/docs/api/java/lang/thread.html.

[3] A. Kfoury A. Bestavros, A. Bradley and M. Ocean. snbench: A development and run-time platform for rapid deployment of sensor network applications.

[4] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification and Object-Oriented Design.* Addison Wesley, 2001.