Extending snBench to Provide Concurrency Support in the Sensorium Execution Environment (SXE)

Londono, Jorge
Boston University Computer Science Department

http://hdl.handle.net/2144/1874
Boston University
Extending the snBench to Provide Concurrent Execution in the Sensorium Execution Environment (SXE)

Jorge Londoño  
jmlon@cs.bu.edu

Sowmya Manjanatha  
sowmya@cs.bu.edu

Zhinan Jenny Han  
jennyhan@cs.bu.edu

Boston University CS Department

May 3, 2006

Abstract

The snBench is a general-purpose programming environment and run-time system targeted towards a variety of Sensor applications such as environmental sensing, location sensing, video sensing, etc. In its current structure, the run-time engine of the snBench namely, the Sensorium Execution Environment (SXE) processes the entities of execution in a single thread of operation. In order to effectively support applications that are time-sensitive and need priority, it is imperative to process the tasks discretely so that specific policies can be applied at a much granular level. The goal of this project was to modify the SXE to enable efficient use of system resources by way of multi-tasking the individual components. Additionally, the transformed SXE offers the ability to classify and employ different schemes of processing to the individual tasks.

1 Introduction

snBench[1] is a programming and run-time infrastructure intended to provide a general-purpose computing environment for sensing applications. For instance, motion detection, temperature sensing and wireless device location tracking are all applications with disparate mission yet all are necessary in a single facility such as an airport, and snBench is designed to support these needs. snBench comprises of stand-alone components analogous to traditional computing systems for e.g., a high level strongly typed functional language, an interim abstract representation: Sensorium Task Execution Plan (STEP), Sensorium Execution Environment (SXE), and a program or a representation dispatcher called Sensorium Service Dispatcher (SSD).

The SXE is the core of the snBench infrastructure where all the STEP programs are executed. Efficiency in processing the STEP programs is important especially for the SXE to be able to support real-time applications such as voice and video. The current implementation of SXE processes the input STEP program serially by iterating through all the individual STEP components namely STEP Nodes. If in the process of execution, a STEP
Node blocks waiting for a System Resource, then execution halts until the Resource is available. The goal of this project was to modify the SXE to enable execution of a set of STEP programs parallely, and, offer flexibility of specific scheduling policies as needed by the client applications. Concurrency is implemented using Java Threads and a set of Custom Schedulers.

The Schedulers are responsible for ordering the nodes to be executed based on a certain criteria for e.g., priority, Round-Robin etc. In reality, application requirements may vary considerably for example certain applications may require fairness in sharing the resources in which case a Round Robin scheduler is a better fit. Some other kinds of applications may require lower delay where Priority Scheduler is a better option. The new SXE engine is designed keeping in mind the possibility of extension of SXE to support newer applications. The design and implementation of the Scheduler and all other components of the new SXE will be discussed in greater detail in Sections 3 and 4.

This report is organized as follows. Section 2 describes the Requirements Specification for this project. Section 3 explains the design of the modified SXE engine. Section 4 provides details of the new software modules and any specific techniques. Section 5 discusses the testing methodology and results of testing. Section 6 presents the conclusions and future work on the project.

2 Requirements Specification

This paper focuses on the redesign of the node execution subsystem of the SXE to support parallel evaluation of STEP nodes. In order to accomplish this goal, we replaced the GraphEvaluatorThread with a more general Scheduler module that coordinates the execution of one or more worker threads. The worker threads are solely responsible for requesting nodes that are ready for evaluation from the scheduler and executing them. Once the node has been executed, the worker thread returns to the pool of available threads and waits until it gets another node from the scheduler.

The scheduler is responsible for coordinating and controlling the execution of the nodes that are part of a STEP program. The scheduler is also responsible for ensuring the following:

1. The order of the evaluation of nodes not only depends on the scheduling policy i.e., HIFO or WRR etc, but also on whether a node is ready to be evaluated or not i.e. a node's value is required by a parent or not. The scheduler needs to accommodate the variations in applications because a node ready to be evaluated or not depends on which level of the tree it is in.

2. The scheduler must keep track of the nodes that are already enqueued for evaluation so that they are not resent. Some nodes may not be evaluated even if they are enqueued because of unresolved dependencies. The scheduler must wait until their dependencies are resolved, which happens when all its child nodes have been evaluated, and then move the node into the ready queue. A worker thread can pick up a node for execution only after it has been marked as dependencies resolved by the scheduler.

3. The scheduler must implement scheduling disciplines such that runtime constraints given by an application are satisfied.
4. Mutual Exclusion between worker threads that share common resources must be guaranteed by the scheduling policies of all kinds and the scheduler must ensure that the integrity of the system is maintained at all times.

We have created a data model that as shown in the figure 1 based on the analysis performed above.

![Diagram of data model for scheduler integrated into SXE]

Figure 1: Data model for scheduler integrated into SXE

2.1 Domains

Server: The main SXE component. It contains all other components and handles HTTP communication

Graph: The representation of the STEP program as a directed acyclic graph

Node: A node represents an action within the STEP program

ReadyNodes: The subset of nodes ready for evaluation

BlockedNodes: The subset of nodes waiting for a dependency before they can be evaluated

Scheduler: Responsible for assigning and coordinating execution of nodes by worker threads

Workers: Set of threads responsible of node evaluation

2.2 Relations

LiveGraph: The graph the server contains. This graph is the result of merging (Grafting) all the STEP programs submitted to the SXE

scheduler: The single scheduler in the system. It is referenced by the server and the nodes

nodes: The set of nodes a graph contains

workers: The pool of threads a scheduler manages

blockedSet: The set of nodes waiting for dependencies in the scheduler

runnableQueue: The set of nodes ready for execution in the scheduler
nodeToEvaluate: The node a worker has been assigned to

2.3 Constraints

A node cannot be assigned to more than one worker thread
for all w1, w2: Workers [ w1.nodeToEvaluate != w2.nodeToEvaluate ]

A node scheduled for execution is either a ready node or a blocked node
for all n1: ReadyNodes, n2: BlockedNodes [ n1 != n2 ]

A node cannot appear more than once in the ready queue
for all n1, n2: ReadyNodes [ n1 != n2 ]

A node cannot appear more than once the set of blocked nodes
for all n1, n2: BlockedNodes [ n1 != n2 ]

3 Design

3.1 Scheduler architecture

The design of the scheduler is provided here in such a way that it can be easily extended to implement different scheduler models as per application needs. To enable such support, we developed a scheduling infrastructure that consists of an abstract Scheduler that defines the general interface for the scheduling engine. The real scheduling algorithms are implemented as subclasses of the parent scheduler.

![Scheduler Architecture Diagram]

Figure 2: Scheduler Architecture

The general architecture of the scheduling system is represented by the MDD in figure 2. Here the SchedulerFactory class implements a factory pattern providing the
means to create a particular instance of a scheduler. The particular scheduler and its operational parameters are determined by the scheduler.properties configuration file as described below. The SchedulerFactory has the only public way of creating instances of Scheduler and makes sure there is only one instance in the application, therefore implementing a Singleton pattern on the scheduler. The StateControlledThread interface imposes an interface for threads with the ability of keeping a state, observing this state and changing the state, which is used at the time of application shutdown to force all threads to exit. The WorkerThread instances are responsible for STEP node evaluation and the ThreadGroup is the standard thread grouping mechanism provided by the Java libraries, used in this case to control the worker threads as a group.

3.1.1 Scheduler factory

The runtime configuration of the scheduler is controlled by the scheduler.properties file, located at the root of the classpath. This file determines the specific implementation of Scheduler to use at runtime and contains the parameters required by this scheduler. If the scheduler.properties configuration is not present, the default action of the SchedulerFactory is to create a SimpleScheduler with a single working thread, and produce a warning in the standard error output. There are three schedulers available in our implementation:

1. SimpleScheduler: Provides a basic multithreaded scheduler with a single fifo queue and a given number of worker threads. A sample configuration for this kind of scheduler is:
   
   ```
   scheduler=sxe.scheduler.SimpleScheduler
   workers = 4
   
   Here, scheduler is the particular scheduler class to instantiate, and workers the number of worker threads to create.
   ```

2. HierarchicalScheduler: Provides a hierarchical scheduler composed of multiple queues, each one with its own queuing strategic (at present FIFO and PriorityQueue) and a given number of worker threads. A sample configuration for this scheduler is:
   
   ```
   scheduler=sxe.scheduler.HierarchicalScheduler
   workers = 4
   queues=4
   queue.0.type=fifo
   queue.1.type=fcfs
   queue.2.type=priority
   queue.3.type=pq
   ```

   In this example a HierarchicalScheduler is specified with 4 worker threads and 4 queues. Queue 0 is set to FIFO, queue 1 is set to FCFS (synonym for FIFO), and queues 2 and 3 are set to be priority queues (pq is short for priority). This scheduler gives maximum priority to queue queues - 1 and minimum to the queue 0. For the purpose of assigning priority to nodes within the priority queues, flow-types are used. Please refer to 3.1.5 for details of flow-types.
3. Weighted Round Robin Scheduler (WRRScheduler): schedules the next node available for execution in a round robin fashion. Similar to Hierarchical Scheduler, this scheduler employs multiple queues. Each queue gets a weighted time slice based on the weights configured in the scheduler.properties file. The default weighting procedure provides same time slice to every queue. A sample configuration of scheduler.properties corresponding to WRRScheduler is:

```javascript
scheduler=sxe.scheduler.WRRScheduler
workers=4
queues=4
queue.0.weight=1
queue.1.weight=2
queue.2.weight=2
queue.3.weight=4
```

In the above example, the scheduler policy employed is weighted round robin. Four worker threads and four queues are created. Each queue gets one of the four weights configured above.

### 3.1.2 Design of the SimpleScheduler

The SimpleScheduler uses a single FCFS queue to enqueue nodes submitted for execution so that the worker threads in the thread pool can then dequeue a node in FIFO order and execute it. Figure 3 illustrates the operation of this scheduler. A node enters the scheduler when submitted via the scheduleNode method, and worker threads request nodes via the getNextNodeToRun method. These methods guarantee mutual exclusion when accessing the queue (the shared data structure). This is analogous to a typical producer-consumer problem [4], [5].

![SimpleScheduler Design](image)

**Figure 3: SimpleScheduler design**

### 3.1.3 Design of the HierarchicalScheduler

This scheduler provides a more general framework for executing STEP programs. This type of scheduler uses multiple queues to enable different scheduling policies between queues. This model could be handy for instance when implementing a real-time scheduling policy, where one queue could be dedicated to real-time processes (Nodes in our case), and other queue could provide a best-effort service. The function of this scheduler is illustrated in figure 4. The HierarchicalScheduler provides a configurable number
of queues (see 3.1.1), each representing a class of jobs. Incoming nodes (submitted via
the scheduleNode method) are then sent to one of these queues based on the node's
class. When a worker thread in the thread pool becomes available, it request a node via
the getNextNodeToRun method, and executes it. The getNextNodeToRun method is
responsible for implementing the hierarchical scheduling policy.

![Diagram of hierarchical scheduler]

Figure 4: HierarchicalScheduler design

This scheduler needs some extra information in order to classify and prioritize nodes
in the queuing system. To do this, we made use of the FlowTypes already in place in the
STEP syntax, so two new flowtype parameters were defined:

1. class: The class defines the queue a node should be assigned to.

2. priority: Which is the mechanism to define the order of the priority queue.

Notice that it is not mandatory to include FlowType information in a STEP program
and we did not want to change this. Instead, we defined that a node without class/priority
information should be assigned the class/priority of its parent, which makes sense in the
context that if the parent node was assigned a high priority, its dependent nodes should
have at least the same priority, otherwise, the parent would not get the expected quality of
service because anyway it will be blocked until its low priority child is evaluated. Notice
also that a STEP program is a DAG and therefore a node may have multiple parents. In
this case, the node should inherit the largest class/priority of its parents. If the parents of
a node do not have class/priority information, then the process will follow recursively up
to the root of the graph. If the root does not have class/priority information, then a default
value basically the lowest values of class 0 and priority 0 are assigned.

Section 3.1.5 describes the FlowTypes defined to assign class and priority to STEP
nodes.
3.1.4 Design of the WRRScheduler

The weighted round robin scheduler employs a set of queues and a set of weights to provide fairness to all types of applications. This scheduler must be employed when all applications are expected to execute within a reasonable time delay. There is a possibility of resource starvation of low priority applications with the use of Priority scheduler especially if the majority of applications submitted to the system are of higher priority. On the contrary, the WRRScheduler provides a kind of priority to some processes in addition to ensuring that all processes will run but at different rates. The difference in execution time is caused due to the weights applied to some processes over other. Another important difference with this scheduler is that weights are applied on a per-program level rather at the STEP node level.

A set of queues are created at scheduler initialization time and each queue is given a different weight. A STEP program may contain a weight in the form of a flow type attached to its root node. All nodes of a STEP program are queued in the same queue. Therefore, if the weight flowtype is provided at the root node of the STEP program, then that weight is used for all nodes of that program. At a particular point in time, a queue from which node is obtained is iterated as follows:

- If a queue is not empty, and weight assigned to this queue is \( W \), and \( W \) threads are available, then \( W \) nodes are dequeued and assigned to \( W \) threads.

- If a queue is not empty, and weight assigned to this queue is \( W \), and less than \( W \) threads are available, then threads will cycle around this queue until \( W \) nodes have been evaluated.

- If a queue is empty, then a thread looks for a node to execute in the the next queue.

Figure 5 illustrates the time slicing policy among queues.

![Figure 5: WRRScheduler time slicing policy](image)

3.1.5 FlowTypes used by schedulers

The existing implementation of the SKE provided the basic classes to support the association of FlowTypes with STEP nodes. The MDD for this design is illustrated in figure 6.

Here a node may be assigned zero or more flowtypes. Each flowtype is identified by its label and contains a list of parameters. The following flowtype was defined for the HierarchicalScheduler:

\[
\text{FlowType: } \langle \text{label= scheduler, parameters= \{class, priority\}} \rangle
\]
For the purposes of the WRRScheduler, one flowtype called weight is defined:

FlowType: <label=scheduler, parameters={weight} >

Where the class parameter will hold the number of the queue and the priority of the
the node. Here is a small step fragment illustrating its use:

<exp id="isnil"  opcode="sxe.core.isnil">
   <last_trigger_eval id="lte0" target="TriggerHead"/>
   <flowtype label="scheduler">
      <param name="class" value="1" />
      <param name="priority" value="2" />
   </flowtype>
</exp>

<exp id="isnil"  opcode="sxe.core.isnil">
   <last_trigger_eval id="lte0" target="TriggerHead"/>
   <flowtype label="scheduler">
      <param name="weight" value="2" />
   </flowtype>
</exp>

In this example, the exp node is assigned to queue class 1 and it is given priority 2.
When no flowtype information is given for a node, it is assumed that the node will go
to the class 0 queue and will have priority 0.

3.2 Trace and Monitoring

The goal of this project is to enable concurrency in the SXE engine and thereby improve
the performance of execution of certain applications. Characterizing the performance of a
system such as the SXE is a complex task and we pondered upon characterization using the following techniques:

1. Characterization using Operating System tools such as top on Linux and netmon on Windows.

2. Characterization by Instrumenting the Code within the engine.

The tools provided by operating systems above monitor system behavior at a very high level and do not provide granularity on the delay introduced by various components of the program. We decided to use the second approach in order to be able to obtain greater depth of information and flexible data gathering methods. The design and introduction of the Trace module started as a necessity for performance monitoring. However, it turned out to be very useful for logging and debugging purposes as well. The design below indicates how the module can be utilized for adding traces that enable unit testing as well as obtaining performance data.

As part of the analysis of the design for this module, we investigated the integration of the open source software called Log4j[2]. Log4j provides several logging routines and also facilities to log to an output file or to a network element using the protocol Syslog. These facilities are attractive but can turn out to be very expensive for a program like SXE for the following reasons:

- Log4j includes classes that contains the routines similar to the ones provided by Trace module. These classes are part of a series of hierarchies. The methods that these classes collectively provide significantly exceeds the needs of the simple Trace facility provided here. This adds complexity and cost since a class overloaded with methods must be called for a simple need.

- The memory footprint of Log4j package itself exceeds the current memory footprint of the SXE engine. As discussed before, the amount of tools utilized from this package is far less compared to the cost that it adds to the SXE engine. Hence, the simple trace module was written to satisfy the current needs of the SXE engine.

In the future however, the Trace Objects called within the SXE can be easily replaced with any other similar logging utility. What follows is a description of the simple Trace module.

The Trace Module is intended to provide several utility routines that aid debugging, logging and performance monitoring tasks. These routines collect specific and generic information as explained below from Scheduler and StateControllerThread modules and log them to an Output File. In addition, the information level is controllable through an input Properties file. At a high level, the Trace utilities can be divided into the following categories:

1. Customization: Customization with respect to the amount and kind of information output and the specific file to output the information is done through the Creator routines. The new SXE engine which comprises of the Scheduler component uses a special input data file namely the scheduler.properties file. This file can also include customizable data about the trace. Tracing can be completely turned off or turned on by one argument. In addition, the amount of tracing done can be changed using a level attribute.
2. Monitoring: The monitoring routines monitor time spent or the delay incurred within a specific method in the Scheduler or the Thread routines. Time spent can be measured at different granularity for e.g. milliseconds, microseconds, nanoseconds or seconds. Monitoring routines can be called from any routine in the code that require characterization.

The main reasoning behind modifying the SXE engine to a multi-tasked unit is to realize any performance benefits if possible. Determining whether there was any improvements is a difficult task because it is hard to synchronize the execution of the SXE threads with respect to an external time monitoring unit such as top in Unix and netmon in Windows. Some tasks performed by the threads could occur in milli or microseconds granularity and timing the monitoring modules to start exactly at the same time as the thread starts executing and stopping at the time as the Thread stops is never possible especially in a single processor system. The monitoring utilities themselves are competing for time with the threads themselves. Therefore, we found that the best way to enable accuracy in time measurement was making the monitoring utility part of the actual program itself.

3. Debugging: In the process of debugging, we found that adding System.out.println everywhere made the output verbose and also hard to control and decipher. Hence, the print messages are abstracted using the logging routines provided by the Trace module. The logging routines also take level parameters so that the log with a specific level is shut off when appropriate level is set from the scheduler.properties file. This enabled us to keep the code for logging debugging information but disable the output of this information when SXE engine is deployed in a production environment.

4. Logging: Some information that is output can be used even during the production phase for offline information tracking purposes. The output information contains some important information such as name of the STEP programs etc. Therefore, an offline processing can provide data regarding how many STEP programs were run using this engine etc.

![Figure 7: Trace Utility functions](image-url)
4 Implementation

Implementation implementations described in this section describe the code organization structure. There was a slight variation of the directory structure from the original mainly to support use of a few very flexible tools of Java. The main java tool that enabled efficient interaction with the team was integration of CVS with eclipse. Eclipse needed the structure to be as described in the Code Organization section below.

4.1 Code organization

We organized the project in Eclipse according with the following tree structure:

bin  Binary distribution of the application.
     We opted for keeping source and compiled classes separated.
doc  Javadoc documentation of the project. The file index.html
     is the home of the documentation tree.
junit-tests  Unit tests created with JUnit.
lib  JAR files required by the application.
src  Source code of the project
TestScripts  Shell scripts for regression testing,

All the new classes belong to the java package sxe.scheduler, therefore they are located at the src/sxe/scheduler directory. The classes are the following:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HierarchicalScheduler</td>
<td>Hierarchical scheduler implementation</td>
</tr>
<tr>
<td>SchedulerConfigurationException</td>
<td>Exception in scheduler configuration</td>
</tr>
<tr>
<td>SchedulerFactory</td>
<td>Scheduler factory class</td>
</tr>
<tr>
<td>Scheduler</td>
<td>Abstract scheduler superclass</td>
</tr>
<tr>
<td>SimpleScheduler</td>
<td>A simple single-queue fcfs scheduler</td>
</tr>
<tr>
<td>StateControlledThread</td>
<td>Thread with controlled state</td>
</tr>
<tr>
<td>Trace</td>
<td>Tracing facility</td>
</tr>
<tr>
<td>WorkerThread</td>
<td>Worker thread</td>
</tr>
<tr>
<td>WRRScheduler</td>
<td>Weighted Round Robin Scheduler</td>
</tr>
</tbody>
</table>

Detailed information about this classes is available as part of the internal documentation and in the Javadoc section of the website.

4.2 Integration with SXE

Very few components of the existing code base depend on the scheduler, so the modifications to the rest of the application where very few and in very localized places. In order to state clearly where these modifications occurred, we always included a comment of the form:

// SJJ - SMOD: Modified this...

In order to facilitate integration with the original SXE code, some modifications were needed in the already defined classes as described below. The changes in the code were
limited to method calls in the original modules. The methods themselves were mainly
defined in the new Scheduler classes except for those that were specific to the original
code such as a "getParents" function added in the Node class.

- **Server** Replaced LiveEvaluator (an instance of GraphEvaluatorThread) by an in-
  stance of the Scheduler singleton. This instance is initialized in the init method.
The serveGraph method, responsible of accepting STEP program submissions was
modified also to schedule the programs root nodes for the first time.

- **Graph** This class was modified to handle the concurrency control via the ReadWrite-
  Lock to implement the Reader/Writer exclusion on the graph.

- **Node** Added the interface to add FlowType information into the Node. Added a
  reference to the scheduler, needed for scheduling sleeping nodes when their inputs
  become available. Also added some observer methods, needed to get the list of chil-
  dren, of parents, flowtype info and check if a node is the only parent of this node.

- **FlowTypeParam** Its implementation was modified to handle the list of flowtypes in
  a hash structure, as lookups by name are going to be very frequent, and therefore
  \( O(1) \) access time is very desirable.

- **FlowTypeParam** It was missing the observer methods of its representation.

- **Parser** Added the code to parse flowtype information from the XML representation
  of the STEP program.

- **TriggerNode, LevelTriggerNode, EdgeTriggerNode** All triggers were modified,
as trigger evaluation goes throw a series of states and scheduling of the subtrees
  depend on the current trigger state. This way, only the nodes belonging to the subtree
  under evaluation are submitted to the scheduler.

### 4.3 Running the application

Running the application has not changed with respect to the original:

```java
java [VMFLAGS] [-cp CLASSPATH] sxe.Server SERVERURL <STEPSource> [STEP]
```

A typical example for our development environment, with assertions enabled (-ea),
binding to the localhost port 8080 and no Service Dispatcher (SD):

```java
```

To submit test programs, we use the standard post utility:

```java
```

### 4.4 Running test cases

#### 4.4.1 Configuring properties file

A sample scheduler.properties file is shown below:
scheduler=sxe.scheduler.WRRScheduler
#scheduler=sxe.scheduler.HierarchicalScheduler
workers = 4
queue=4
trace=1
tracelevel=5
#queue.0.type=fifo
#queue.1.type=fcfs
#queue.2.type=priority
#queue.3.type=pq
queue.1.weight=1
queue.2.weight=2
queue.0.weight=3
queue.3.weight=4

Some of the lines in the file start with a # that indicates that it is commented out. We use this here only to show all values that we employ. All the values that we use for the purposes of flexibility is shown above. The values are self explanatory. "trace=1" indicates that tracing is enabled. "tracelevel=5" indicates high level of tracing. Currently, in our code, we have used a tracelevel logging of up to 4. Hence, 5 will enable all tracing. Reducing the level to 1 will disable all tracing used in code that is above level 1. Setting "trace=0" completely disables tracing. Some printouts are still enabled to provide some indication that the program is executing.

5 Testing

Testing the SXE Modification Engine was the most complicated phase. The majority of testing with the new SXE Engine involved testing the new dynamics of scheduling with threads. As recommended by software engineering practices, testing was divided into

- Unit Testing: Individual units for e.g., schedulers, trace module and factory were tested independently.

- Integration Testing: One or more modules for e.g., factory with schedulers or trace with schedulers were integrated and tested to verify their interoperability.

- Functional Testing: Testing in this phase was mostly focused on validating the scheduling policies, mutual exclusion guarantee during individual posts, etc.

- Stress Testing: involved use of a couple of scripts in quickly running the tests and comparing already tested entities after any new change done to the code.

The sections to follow explain the longest phases namely unit and functional testing in greater detail.

5.1 Unit testing individual modules

Unit testing the scheduler implementations turned out to be very difficult. Standard unit testing facilities like JUnit do not provide support for multithreaded applications, which are more difficult because testing must take into consideration aspects such as:
• Sharing information between testing thread and application threads: The whole idea of unit testing is to be able to show that the postconditions of individual methods hold after its invocation. However, there could be other threads accessing the representation invariant of the objects being tested and therefore one cannot make guarantees about the representation not being modified after returning from the method call. This also raises race hazards, whose solution would require modifying the application to guarantee mutual exclusion with the testing thread, a change not trivial, and that could itself introduce a lot more bugs and have negative repercussions in the application performance, source code legibility and maintainability. We found some work [3] on how to deal with this problem, but we did not have chance to try that method.

• Synchronization mechanisms rely on the ability to block individual threads. If the testing thread is blocked, the test may never end. We actually fall into this pitfall when trying the dequeueing function of the scheduler. If the queue is empty, the expected behavior is to block the thread until some object enters the queue. Having no other threads producing items, the testing thread blocks forever. As this is what actually happened, the test was successful.

• Order of execution of threads is not guaranteed, on the contrary, different executions may result in different execution. This turned out to be an issue when trying to test the queuing strategy, as inserting a sequence of nodes in the queue may result in other threads extracting nodes in different orders, for example if the testing thread inserts a node and then tries to extract it, sometimes a worker thread may picked it up before, or sometimes it may get it. A simple solution to this particular case is making the number of workers equal to zero, but this implies modifying the representation invariant (which calls for workers ≥ 1), which we thought its not the right thing to do, as would open the door for configuration errors when the application is deployed.

We explored using JUnit, all the test cases were created under the directory junit-test (separate from the main source code tree), and created tests for object creation, like verifying the creation fails and returns the right exception when the parameters do not satisfy the prerequisites, or making sure the singleton pattern hold when requesting a scheduler from the factory many times. It should be noted that the singleton pattern also caused problems for the test cases. More specifically, we wanted to test the factory with different combinations of parameters, some that were expected to fail, and some to pass. The singleton itself would forbid to create further instances after the first successful instantiation. To overcome this difficulty, we provided an extra static method to clear the singleton instance (Scheduler.resetSingleton()), a method that should only be called from the test case and never from the application itself. We provided a close approximation to this functionality by including an assert(false) within the method. So when running the application, out of the test case and with assertions enabled, the method will always produce an assertion failure if ever called.

5.2 Black box regression testing using traces

Our main method to verify the correct execution of step programs by our schedulers was mainly to generate traces of the detailed sequence of events during the STEP program
execution, as previously described in subsection 3.2. This method is however extremely cumbersome, error-prone and time consuming. For this reason we experimented with another approach to automate trace verification. Our method works as follows:

1. Run a reference STEP program, check the trace manually and save this trace to a file.
2. After making modifications to the program, run again the trace and compare it to the original trace.
3. If traces match, repeat 1 and 2 for various reference programs.
4. If traces match for all reference programs, then the regression test passes.
5. If at any step the traces do not match, the regression test fails.

The previous method has however a serious flaw: different correct executions of the same program will produce different traces. This is due mainly to the non-deterministic nature of the queuing systems involved in the schedulers themselves, and the presence of random functions in the program. In order to overcome this difficulty, we need to find some invariant in the trace that holds among executions. For the case that the STEP program has a single root (i.e. there is only one STEP program in the SXE Graph structure) and no random functions, the invariant are the evaluations of the root node, as this node will only be executed after all its children have been executed, and the result after consecutive iterations (in the case the root node is a trigger) with given inputs, will always be the same. This is our improved trace evaluation method:

1. Run a reference STEP program, check the trace manually and save record of root node evaluations to a file.
2. After making modifications to the program, run the trace again and compare results of root evaluation to the original trace.
3. If traces match, repeat 1 and 2 for various reference programs.
4. If traces match for all reference programs, then the regression test passes.
5. If at any step the traces do not match, the regression test fails.

Script testEvaluationTrace, located in the TestScripts or our source is code implements the first two steps of this procedure (to leave open the option on adding more test programs as needed).

This method however fails if the Graph contains multiple roots, because the sequence of evaluation of the root nodes is not deterministic and even worse, evaluation of one root can affect the value of another, as illustrated in the STEP graph of figure 8.

So, in this example, depending on the timing of execution, the add node may get the first value of the trigger0 (or null if trigger0 has not been evaluated yet, in which case add returns 1 as of the current implementation), and the value returned by trigger1 depends on the particular sequence.

We also developed a list of test cases that can most accurately validate the functionality of the schedulers. It is impossible to compose the entire raft of test cases that would satisfy all possible applications used. Instead, we picked a sample set of STEP programs and test cases that executes most of the strategies of a particular scheduler and executed them. We also created a few new STEP programs to test our schedulers better. Below is an illustration of test cases appropriate for each scheduler.
5.2.1 Sample test cases for Hierarchical Scheduler

Table 1 describes the test cases considered for the hierarchical scheduler.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>STEP Program Used</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Threads</td>
<td>counter-jml-flowtype.step.xml</td>
<td>Test node dispatching logic with 1, 2 and 4 threads</td>
</tr>
<tr>
<td></td>
<td>wordcount.step.xml</td>
<td>Test concurrent node evaluation</td>
</tr>
<tr>
<td>Test Queues</td>
<td>basic_trigger.hql1.xml and basic_trigger.hql2.xml</td>
<td>Hierarchy queuing with 2 fcfs queues</td>
</tr>
<tr>
<td></td>
<td>basic_trigger.hql1.xml and basic_trigger.hql2.xml</td>
<td>Hierarchy queuing with 2 priority queues</td>
</tr>
</tbody>
</table>

Table 1: Sample Functional Test Cases for Hierarchical Scheduler

5.2.2 Sample test cases for WRRScheduler

The WRRScheduler can contain one or more queues and a set of weights assigned to the queues.

![Diagram](image)

Figure 8: Root node interdependency
<table>
<thead>
<tr>
<th>Test Case</th>
<th>STEP Program Used</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Threads</td>
<td>basic_trigger.xml</td>
<td>Test the number of threads with default in properties file (emulate original SXE engine)</td>
</tr>
<tr>
<td></td>
<td>basic_trigger.xml</td>
<td>Test 1 thread creation</td>
</tr>
<tr>
<td></td>
<td>basic_trigger.xml</td>
<td>Test 2 threads creation</td>
</tr>
<tr>
<td>Test Weights</td>
<td>counter-sm-flowtype-weight.xml</td>
<td>Test by giving weight flowtype to root node</td>
</tr>
<tr>
<td>Test ordering</td>
<td>fakegrabber.step.xml</td>
<td>test ordering of programs with different weights</td>
</tr>
<tr>
<td></td>
<td>counter-sm-flowtype-weight.xml</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Sample Functional Test Cases for WRRScheduler

Therefore, when testing the WRRScheduler, it is important to cover test cases that test a set of boundary conditions with default values, 1 queue, 2 queues, etc., and with same weights assigned to all queues and different weights assigned, etc. In addition, each of the schedulers can be configured to create one or more threads and again boundary conditions on number of threads used to execute must be tested. In addition, we need to also ensure that the program gets the right set of weights by looking at traces to figure out if in one round of execution, the number of threads provided to one queue corresponds with its weights. Table 2 gives a sample set of test cases and the STEP programs used to test WRR functionality.

### 5.2.3 Sample test cases for SimpleScheduler

Table 3 describes the test cases considered for the simple scheduler.

### 6 Conclusions and Future Work

#### 6.1 Conclusions

Throughout this project we have had the opportunity to get first hand experience with many of the tasks involved in the development of large software projects and the different methodologies that around object oriented programming have been devised to carry out such projects. Among those experiences acquired during the realization of this project we have:
<table>
<thead>
<tr>
<th>Test Case</th>
<th>STEP Program Used</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Threads</td>
<td>counter-jml.step.xml</td>
<td>Test node dispatching logic with 1, 2 and 4 threads</td>
</tr>
<tr>
<td></td>
<td>wordcount.step.xml</td>
<td>Test concurrent node evaluation</td>
</tr>
</tbody>
</table>

Table 3: Sample Functional Test Cases for SimpleScheduler

- Requirement analysis, specification and design provide a solid foundation to the development of a complex software system. Implementation is much easier and it helps identify and plan for the required test cases.

- However, test cases themselves may impose additional requirements on the software system in order to facilitate the creation of the minimal environment to run the test themselves. It is therefore very important that requirement and design phases consider as well testing, as it may require additional considerations, and may even help conceive a better, more general design.

- Testing can be the most challenging part of complex software project. Even unit testing may demand considering a very large number of cases to test and to write a lot of code (also prone to errors) to conduct these tests.

- Additional difficulties in testing come from multithreading, as program execution may be scheduled differently between executions and determining if the results are correct is a very laborious and error-prone task.

6.2 Future work

There are several issues we think may be interesting for further development of this project:

- Standard modelling techniques have little or no consideration for concurrency issues. To do so, we went into writing an analysis of the issues that our project demanded, which were after all standard, well known techniques. We think this could be also modelled and included in the design, so developers are more aware of the implications and do not forget easy to avoid mistakes in the implementation phase, such as forgetting protecting a share object.

- Trigger constructs in STEP probe are very tricky to handle. Multiple triggers in a program can introduce non-determinism and this makes it difficult to define the accurate result. Given this problem, testing also turns out to be much more complex. In general we have done testing in simple programs with one one trigger which keeps things simple. Much more work is required to define the correct behavior of triggers in general and to implement test cases to verify the implementation does as expected.
• One of the possible areas of application of the SNBENCH infrastructure is in real-time systems, either for monitoring or control of complex network sensors. Our initial step towards a better scheduler system, falls pretty short in considering what the needs for such systems may be and how those needs could be implemented. This is left open for further development.
References


