

Quintiq Internship Projects

March - August 2008

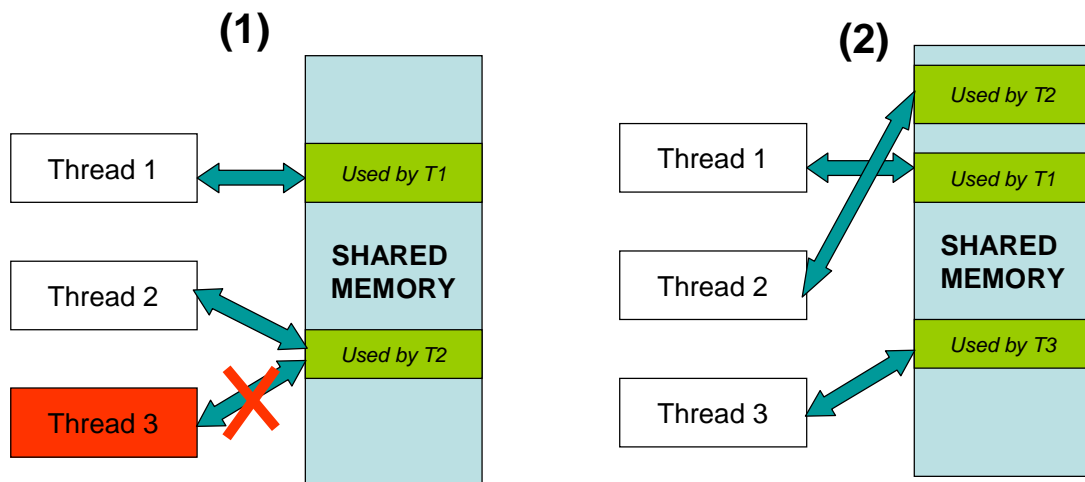
CONTENTS

Contents	i
1 Object-based software transactional memory	1
2 .NET query interface generator	2
3 Quill Virtual Machine	3
4 Generic distributed test driver	4

1 OBJECT-BASED SOFTWARE TRANSACTIONAL MEMORY

The software transactional memory is a lock-free concurrency model that can greatly improve the execution performance of concurrent threads. A transaction is defined as an atomic operation that reads from and writes to the threads' shared memory.

STM is optimistic, assuming that all the threads that are currently running are modifying different parts of the shared memory and thus non-conflicting. When conflicts occur, the first transaction to be committed to the shared memory wins; all subsequent transactions are reverted and must be run again. In order to detect the read/write overlaps each transaction must record a log of the reads and writes it wants to perform to the shared memory.



You can see an example in the picture above. At first, on the left side of the picture, thread 1 uses a piece of shared memory not in use by any other thread, and can thus commit without problems. Thread 2 uses the same piece of memory used also by thread 3, but finishes first, and is thus able to commit. Thread 3 finds out on commit that it is in an invalid situation and reverts its current transaction, rescheduling it for later. On the right side we see the subsequent moment. Threads 1 and 2 continue their execution unhindered. Thread 3, after retrying its reverted transaction, is finally able to commit.

As an internship project, we would like you to investigate this concurrency model: performance, implementation efforts, pitfalls etc. You will start with a literature survey on lock-free concurrency models in general and STM in particular. The next step of the internship program would be the implementation of a proof-of-concept, either using an available library or, if you prefer, your own transaction manager implementation. The language of choice for this implementation is C++.

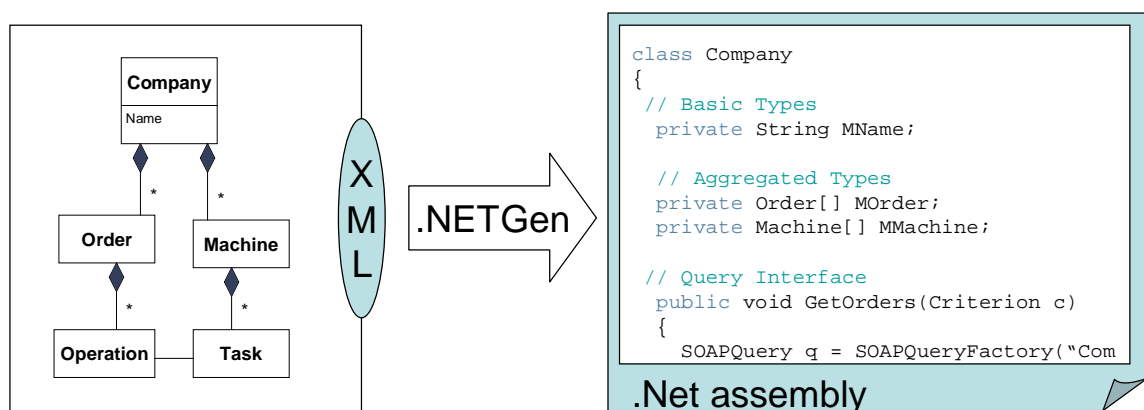
An additional point of interest for us in such an implementation is the implementation of undo-redo operations in the context of STM. More specifically, we would like to see a solution that takes advantage of the transaction logs that are already recorded by the transaction manager.

The STM implementation should be checked against a standard locking mechanism using different scenarios. A scenario should include the number of concurrent threads, the probability of using the same memory segment etc. The results of the benchmarks should be included and analyzed in the final report.

2 .NET QUERY INTERFACE GENERATOR

The power of the Quintiq software is that it allows for the construction of a model that is specifically tailored for one specific customer. This model is built using generic classes made available by the application, which are then subclassed as needed. The underlying platform understands these modeled types and creates instances at runtime, which are subsequently used to represent the customer's world.

An important part of any business process is the data mining and reporting based on the current state of the world (or a previously saved one). The Quintiq application currently understands SOAP queries. We would like to extend this by generating a .NET assembly that would closely mimic the Quintiq model and encapsulate the SOAP queries directly. This would allow for .NET applications to be built that could use this knowledge in a straightforward manner.

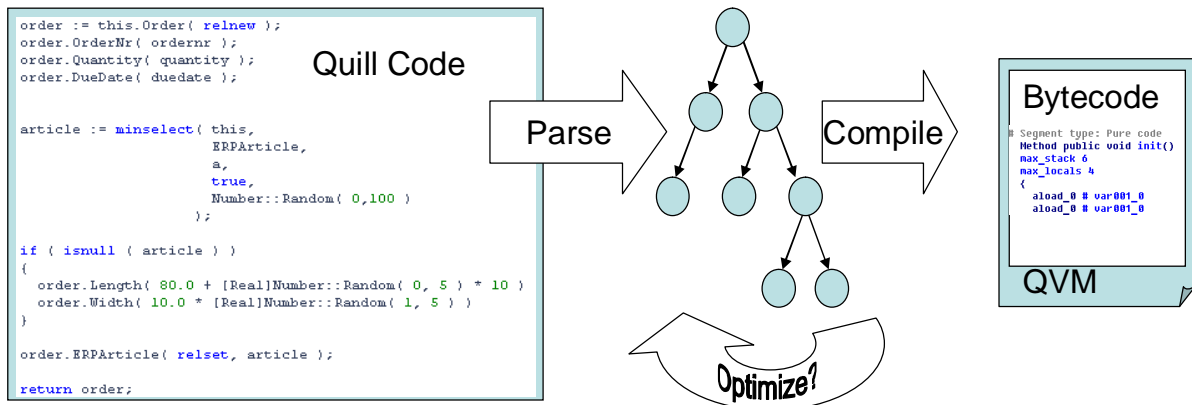


The Quintiq model can be currently exported to XML. This XML description should be subsequently used to obtain a .NET assembly containing the types plus their attributes and relations. We are not (yet) interested in a live feed (push) from the Quintiq application to the .NET assembly. Instead, we will pull the state of the model whenever reporting is required. The query functions that need to be generated will encapsulate SOAP calls to the Quintiq application.

The project also requires the build of a proof-of-concept query interface using a generated .NET assembly. Since a generic SOAP query interface is not yet available, you are only required to work with one small model, such as the one shown in the picture above (6-10 types, together with their attributes and relations). Quintiq DOES support model-specific SOAP queries, which are easy to specify and use, so the absence of a generic interface is not an obstacle.

3 QUILL VIRTUAL MACHINE

Quintiq uses its own modeling language, called Quill, to add both procedural and declarative logic to its models. This language is currently parsed via Yacc++ and the resulting abstract syntactic trees are stored in the server memory. We subsequently evaluate these ASTs within a given runtime context whenever necessary – a typical implementation of an interpreted language.



We would like to examine another possibility, namely the execution of Quill code in a Quill Virtual Machine. We suspect that this would attract significant improvements, of up to or exceeding one level of magnitude (~10x faster). At the same time the compilation of ASTs into bytecode offers also opportunities for AST optimizations, a field which we have not yet considered for our standing implementation.

In this project you will be required to examine existing VM implementations and choose – or design – a suitable architecture and bytecode tailored for the Quill programming language. Then you should implement a proof-of-concept tool chain that would consist of the following components:

- Quill Parser – to parse a Quill program and create its corresponding AST
- Quill AST Compiler – to compile the AST from the previous step into bytecode
- Quill Virtual Machine – to run the bytecode

The grammar of Quill is already fully specified for YACC++. To parse the Quill language you may either adapt the current YACC++ implementation or create another (by using Antlr for instance).

The QVM should be then compared with the current interpreted implementation using different examples of Quill code. The benchmark results should be part of the final report.

There are two extra topics that we are interested in during the course of this project – they both deal with optimization, and you may choose either one and investigate it after you have a working prototype of the described tool chain. The first one is the investigation of possible AST optimizations. The second is examining Just-In-Time compilation opportunities in the generated bytecode.

You may implement the tool chain in either C++ or Java. We believe that QVM itself should be implemented in C++, but that can be changed, provided you have strong arguments in favor of another language.

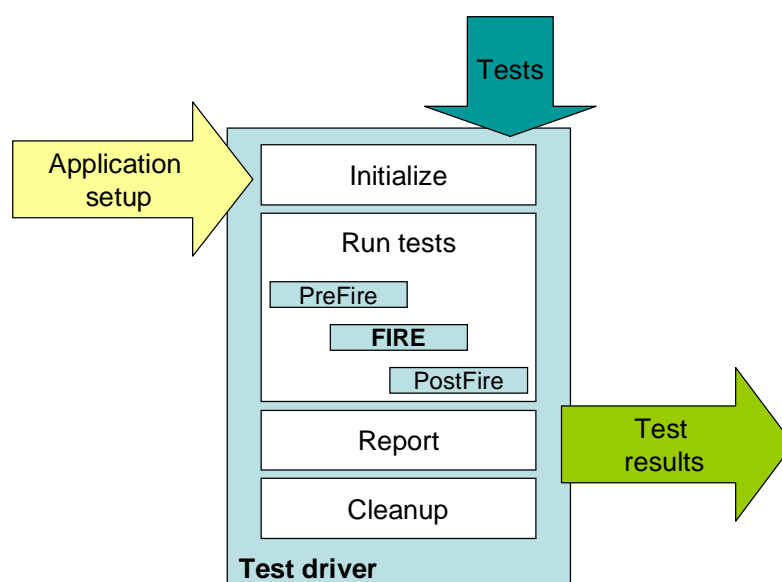
4 GENERIC DISTRIBUTED TEST DRIVER

A test driver or test harness is a tool that is able to perform functional tests for a given application. The typical job of a test harness can be described as follows:

```

InitializeTestEnvironment()
for each test in Tests do
    PreFireSequence()
    FireSequence()
    PostFireSequence()
CleanUpTestEnvironment()
    
```

Let us assume that we have a setup of a client-server application and a set of tests to perform. The *Initialize* operation would install the application on the machine or machines that participate in the test. Then, for each test that needs to be run, the test is set-up in the *PreFire* step, run in the *Fire* step and the results of the run will be evaluated in the *PostFire* step. The results of all the tests are then aggregated in the *Report* stage, and before the test driver exits it will also deinstall the application in the *Cleanup* stage.



The test driver is not application dependent, but generic enough to accommodate various applications or groups of applications. An additional level of complexity is added if the tests are required to run over a network. In this case an agent-based distributed test environment is required that would be able to accommodate at least the first of these two scenarios:

- Run Test1 on machine M1 and Test2 on machine M2 in parallel
- Run Test1 as follows: start a server on machine M1, start a client on machine M2, collect and aggregate results from both client and server

Another important aspect of such an application is test management. We should be able to specify, either from the command-line or other means (ini file, web interface etc.) which tests or suites of tests should be run for a particular test cycle. Command-line is preferred, since such an application is typically used as a part of a fully automated nightly build process.

You should design and implement the test driver application. Since such an application is not strictly Quintiq-related, we will consider together with you whether to release it under an open-source license. You may implement this in the language of your choice, as long as you have good arguments in its favor.