ABSTRACT
This paper intends to talk about the various adaptive programming techniques used in optimizing the execution performance of a rule engine in Java.

Categories and Subject Descriptors
I.2.3 [Deduction and Theorem Proving]: Inference engine, execution algorithm and optimization techniques, Java programming.

General Terms
Algorithms, Performance, Design, Languages.

Keywords

1. INTRODUCTION
A rule engine executes the rules according to an algorithm. Our rule engine implements the RETE algorithm and is written in Java. A rule engine evaluates the tests specified in the rules against the objects, and if the tests are verified, actions will be executed.

Our rule engine has the following characteristics:
- It can work on any Java object model. It does not make any assumptions or put any constraints on how the object model is organized.
- The tests expressed on the rules are frequently evaluated during the evaluation phase.
- When the tests are verified, the actions are applied. The execution of the actions consumes little time compared to the condition part.

The rules are described using a specific Java-like rule language: the ILOG Rule Language. The execution algorithm relies on Java introspection to interpret the constructs of this language.

Introspection is a powerful feature of the Java platform; many Java applications rely on it. It brings adaptive capability to permit Java tools and frameworks to be abstracted from their target application’s object model: adapted processing will be defined at runtime. Its drawback, however, is slow performance and higher memory consumption compared to direct calls to the Java members. To improve the performance of the rule engine, several techniques have been used.

1.1 Dynamic code generation
The rule engine spends most of its time evaluating the condition parts of the rules. The rule condition parts access the objects (by introspection) and compare the field and method values against some other values. To improve the performance of the engine, we decided to optimize this part and proceeded to generate Java bytecode dynamically to perform the tests. The generated bytecode accesses Java objects directly and bypasses Java introspection, allowing the tests to be evaluated much faster than with introspection.

1.2 Plug and play optimizers
The bytecode generation has improved the performance of a particular step in the algorithm: the test evaluation. This optimization is indeed very generic.

For more performance, more adapted optimizations should be enabled. As the generic optimizations often suffer by their generality, it is required to foresee a hook through which developers can add new optimizations.

A technique used in the rule engine is the plug and play optimizer. At a certain step in the algorithm, the rule engine uses the service of an optimizer object to perform the optimizations. Instead of calling the optimizer class directly, it is invoked through an interface that provides the same service. The interface has a default implementation provided in the rule engine, but the customer gains the possibility to implement a more specific class.

1.3 Generating the whole program dynamically
If we reduce the requirements set on the rule engine, and this makes sense for some specific cases where the rules are simple validation rules, it is possible to generate a whole rule engine dynamically in Java bytecode and invoke it directly.

This approach has the strong advantage of allowing the user’s objects to be accessed using direct invocation with no introspection. Also the rules that are active for a specific problem are generated in the bytecode.

Of course such an approach raises many engineering issues, including the robustness and the scalability of code generation when the number of rules reaches tens of thousands and beyond. Those issues are to be addressed by an industrial organization of the project.